ECE 2230: Computer Systems Engineering **Fall 2018**
Machine Problem 6 **Due: 11:59 pm, Monday, November 19**

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 are required to use the form of Dijkstra's algorithm as provided in the file **`pseudo_dijkstra.c`**. 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 not 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.

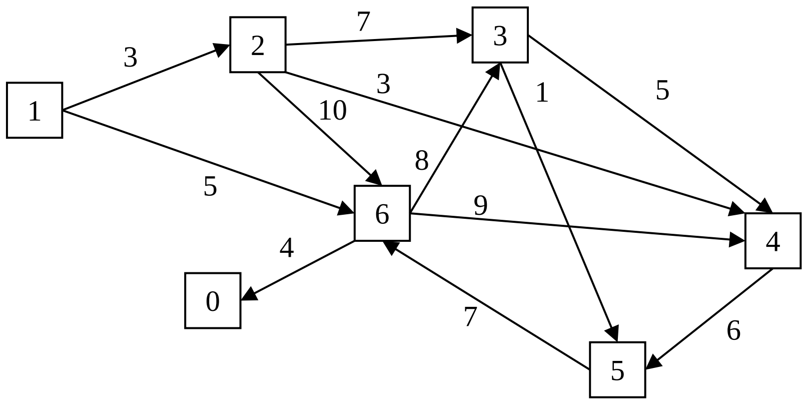
Interface specifications

Command line arguments must be used to modify 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` *R*	Approximate number of adjacent vertices $0 < R < N$
`-h` [1 | 2 | 3]	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

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

Notice that the edges in this graph are not symmetric, and there is an edge between every pair of vertices.

3. **Random graphs**, with *N* vertices and approximate number of adjacent neighbors *R* (**`-n`** *N* **`-a`** *R*). The vertices are numbered 0 through *N* – 1.
   - Vertex 0 is at location (0.0, 0.5)
   - Vertex *N* – 1 is at location (1.0, 0.5)
   - 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 $R = N\pi C^2$, 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 *R*, the radius (or normalized communication range), *C*, is

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

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

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

   - 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 undirected ($w_{i,j} = w_{j,i}$), 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, *R*, is smaller than approximately 10. With high probably the graph is connected if *R* is larger than approximately 20.

For random graphs you **must** print the following information after the graph is built (and before any changes to the graph are made): The average, maximum, and minimum number of adjacent vertices taken over all *N* vertices. For large graphs, the average number that you calculate will be slightly smaller than the value you specify with `-a R` because nodes near the area boundary will have fewer than an average number of neighbors.

Graph Operations (`-h` [1 | 2 | 3])

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 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 are `-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 2,172 and there are 11 vertices in the path (there are multiple paths with the same cost, and the path you find depends on how ties are broken). Your implementation of Dijkstra's algorithm must achieve performance that is no worse than $O(n^2)$ where *n* is the number of vertices. If you choose, you can make an implementation that can achieve $O(n \log n)$ for sparse graphs if you use adjacency lists and a priority queue with a heap representation, however, the more efficient operation is not required.

2. **Network diameter**. Find the vertex 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 there are one or more vertex pairs for which the cost is infinity, print one message stating that the graph contains at least one pair of vertices without a path. The output should print the value for the source-destination pair that has the largest cost less than infinity and the path. 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. Your implementation of this algorithm must achieve performance that is no worse than $O(n^3)$ where *n* is the number of vertices.

3. **Multiple link-disjoint paths**. 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 destination. Then, remove the edges in the path from the graph. Then, find the next shortest path between the same source and destination. Repeat until there is no longer a path between the source and destination.

   For example, if the parameters are `-g 1 -h 3 -s 1 -d 5`, there are two paths found when searching for shortest paths: `1 -- 2 -- 3 -- 5` with cost 11 is found first, and when these links are removed from the graph there is a second path `1 -- 6 -- 4 -- 5` with cost 20.

(Open question: for `-g 1 -h 3 -s 1 -d 5`, the total cost of these two paths is 11 + 20 = 31. However, if the objective is to minimize the sum of the costs of the link-disjoint paths, a better solution is (1, 2, 4, 5) with cost 12 and (1, 6, 3, 5) with cost 14, for a total cost of 26. However, the first path is not the shortest path. Our link-disjoint path approach is an example of a greedy algorithm that does not find the best solution when the goal is to minimize the sum of the costs of the disjoint paths. Can you find an algorithm that performs better than the greedy approach?)

Performance Evaluation

1. Computational complexity verification for Dijkstra'a algorithm. As we discussed in class, Dijkstra's algorithm requires time that is $O(n^2)$ in general, where *n* is the number of vertices. However, if a graph is sparse (i.e., the average number of neighbors is bounded and does not

depend on the size of the graph), then an implementation of Dijkstra's algorithm using adjacency lists and a priority queue with a heap representation can achieve $O(n \log n)$.

a. For graph type two (strongly-connected and directed) show that your implementation is $O(n^2)$. This graph is dense and represents a worst-case scenario because there is an edge between each pair of vertices. One way to demonstrate this is find a value for the number of vertices, **`N`**, that is large enough so that the run time is greater than one second for the options **`-g 2 -h 1 -n N -s 0 -d N-1`**. Measure the time to run Dijkstra's algorithm using the `clock()` function (only measure the time Dijkstra's algorithm requires and do not include the time to build the graph). Run the same experiment but with the number of vertices as ***`N`*`/2`**. Show that the ratio of the two run times is approximately 4. (Why does this verify that the performance is $O(n^2)$?)

b. Repeat the experiment for graph type three (random) with the input for the number of adjacent vertices equal to 20 (i.e., **`-a 20`**). Demonstrate if your implementation is $O(n^2)$ or $O(n \log n)$, and explain why.

2. Computational complexity verification for finding the network diameter. Using graph type three (random) with **`-a 20`**, demonstrate that your implementation is either is $O(n^3)$ or $O(n^2 \log n)$, and explain why.
3. Node density for random graphs. For the types of random graphs generated with the **`-g 3`** option, there is a relatively narrow range for the value of **`-a R`** (the approximate number of adjacent vertices) such that the graph changes from a low probability of being connected to a high probability. This range does not depend on the number of vertices. The graph operation **`–h 2`** (network diameter) is one way to determine if a graph is connected. Consider random graphs with **`-a 7`** and show that with the **`-g 3 -h 2 -n 100 -r seed -a 7`** that the graph is usually not connected (try with at least 10 different seeds for the random number generator). In a second experiment, show that if you take **`-a 20`**, then it is very rare that the graph is not connected (again show the output for some different trials with different values for the seed).
4. Multiple link-disjoint paths

a. For graph type two show that the number of paths is equal to **`N-1`**, where **`N`** is the number of vertices. For example, **`-g 2 –h 3 –n 15 –s 14 –d 0`** should show 14 paths.

b. For random graphs (type 3) consider graphs with different number of average neighbors (**`-a R)`** Use **`-g 3 -h 3 -n 1000 -s 0 -d 999 -a R`**. Compare the number of link-disjoint paths found, trying different values for *R* (such as 10, 20, 50, and 100).

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

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

3. Begin with a top-down design. Once you have a design then you start with bottom-up programming (never write more than ~10 lines of code before testing!). For example, you **might** decide to make a graph module that includes:

   ```
       graph_t * graph_construct(NumVertices)
       graph_destruct(G)
   ```

   At least two operations seem likely (but other designs are possible):

   ```
       graph_add_edge(G, link_src, link_dest, link_weight)
       graph_shortest_path(G, path_src)
   ```

   Every program we have done has had a debugging function to print internal details. So add

   ```
      graph_debug_print(G)
   ```

   Once you decide on a design for your module you might have something like:

   ```
   G = graph_construct(NumVertices)

   if GraphType is 1
       graph_add_edge(G, 1, 2, 3);
       graph_add_edge(G, 1, 6, 5);
       graph_add_edge(G, 2, 3, 7);
       graph_add_edge(G, 2, 4, 3);
       graph_add_edge(G, 2, 6, 10);
       graph_add_edge(G, 3, 4, 5);
       graph_add_edge(G, 3, 5, 1);
       graph_add_edge(G, 4, 5, 6);
       graph_add_edge(G, 5, 6, 7);
       graph_add_edge(G, 6, 0, 4);
       graph_add_edge(G, 6, 3, 8);
       graph_add_edge(G, 6, 4, 9);

   else if GraphTYpe is 2
     for(link_src = 0 ; link_src < NumVertices; link_src++)
       for(link_dest = 0 ; link_dest < NumVertices; link_dest++) {
         if (link_src == link_dest) continue;
         graph_add_edge(G, link_src, link_dest, weight(link_src, link_dest));
       }
   ```

4. **You must develop your own implementation of Dijkstra's algorithm following the pseudo-code from the textbook and as studied in class. Other variations on Dijkstra's algorithm are not permitted for this assignment**.

   See the file **`pseudo_dijkstra.c`** for the code from the textbook. This version includes a few additional notes. Furthermore, there are notes about the required prints for the **`Verbose`** mode.

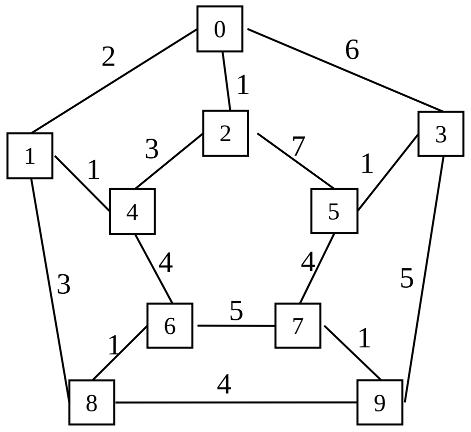
See the ECE 223 Programming Guide for additional requirements that apply to all programming assignments. All code and documentation files must be turned in by email to ece_assign@clemson.edu. Use as subject header ECE223-1,#6.

Work must be completed by each individual student, and see the course syllabus for additional policies.

Optional Extra Graph

Here is another graph for testing. This is the graph from homework set 7. With this graph, the **`Verbose`** print when the source vertex is 0 should give the order in the table. See the **`pseudo_dijkstra.c`** file for the details on how the **`Verbose`** print should be used.

Vertex	Cost	Predecessor
0	-	∅
2	1	0
1	2	0
4	3	1
8	5	1
3	6	0
6	6	8
5	7	3
9	9	8
7	10	9



```
if GraphType is 4

    graph_add_edge(G, 0, 1, 2);
    graph_add_edge(G, 0, 2, 1);
    graph_add_edge(G, 0, 3, 6);
    graph_add_edge(G, 1, 0, 2);
    graph_add_edge(G, 1, 4, 1);
    graph_add_edge(G, 1, 8, 3);
    graph_add_edge(G, 2, 0, 1);
    graph_add_edge(G, 2, 4, 3);
    graph_add_edge(G, 2, 5, 7);
    graph_add_edge(G, 3, 0, 6);
    graph_add_edge(G, 3, 5, 1);
    graph_add_edge(G, 3, 9, 5);
    graph_add_edge(G, 4, 1, 1);
    graph_add_edge(G, 4, 2, 3);
    graph_add_edge(G, 4, 6, 4);
    graph_add_edge(G, 5, 2, 7);
    graph_add_edge(G, 5, 3, 1);
    graph_add_edge(G, 5, 7, 4);
    graph_add_edge(G, 6, 4, 4);
    graph_add_edge(G, 6, 7, 5);
    graph_add_edge(G, 6, 8, 1);
    graph_add_edge(G, 7, 5, 4);
    graph_add_edge(G, 7, 6, 5);
    graph_add_edge(G, 7, 9, 1);
    graph_add_edge(G, 8, 1, 3);
    graph_add_edge(G, 8, 6, 1);
    graph_add_edge(G, 8, 9, 4);
    graph_add_edge(G, 9, 3, 5);
    graph_add_edge(G, 9, 7, 1);
    graph_add_edge(G, 9, 8, 4);
```