ARIZONA STATE UNIVERSITY

CSE 310 — **Data Structures and Algorithms** — Spring 2021

Instructors: Dr. G. Xue and Dr. Violet R. Syrotiuk

Project #3

Due Wednesday, 04/21/2021

This project involves algorithms on graphs represented using adjacency lists. The project has two parts:

1. An implementation of a variant of Dijkstra's algorithm with correctness evaluated using Gradescope.
2. A study of graph characteristics written up in a report.

Note: This project **must** be completed **individually**, *i.e.*, you must write all the code and report yourself.

Your implementation **must** use C/C++ and ultimately your code for a variant of Dijkstra's algorithm will be evaluated for correctness using Gradescope which runs on an `Ubuntu 18.04` Linux system. You will have unlimited access to evaluate your code yourself on Gradescope.

All dynamic memory allocation for the adjacency lists and min-heap **must** be done yourself, *i.e.*, using either `calloc()`, `malloc()`, or `new()`. They should be released using either `free()` or `delete()` when they are no longer needed. You **may not** use any external libraries to implement any part of this project, aside from the standard libraries for I/O and string functions (`stdio.h`, `stdlib.h`, `string.h`, and their equivalents in C++). If you are in doubt about what you may use, ask for permission.

By convention, your program should exit with a return value of zero to signal that all is well; various non-zero values signal abnormal situations.

You **must** use a version control system as you develop your solution to this project and commit to it frequently. Your code repository must be private to prevent anyone from plagiarizing your work. In this project, your submission will require a snap shot of the commits you have made to demonstrate your work.

The rest of this project description is organized as follows. In §1, the requirements of each of the three phases of the project are described. §3 describes a study of graph characteristics for you to compute and summarize in a report. §4 provides detailed submission instructions for this project.

1 A Variant of Dijkstra's Algorithm

Your program for this project has three parts:

1. First, your program must read in a weighted graph and represent it using adjacency lists.
2. Secondly, you need to design and implement a variant of Dijkstra's algorithm, as specified in §1.2.
3. Finally, your program must answer queries based on your variant of Dijkstra's algorithm.

Each of these parts are now described in the following subsections.

1.1 Format of the Graph

You are to read in an edge weighted graph $G = (V, E, w)$ from a text file whose name is specified as the first command line parameter. The type of graph is specified as the second command line parameter. Your executable named `dijkstra` is therefore invoked as:

```
./dijkstra <graph> <direction>
```

where `<graph>` is the name of the text file containing the input for the graph, and `<direction>` is either `directed` or `undirected`, denoting that the graph is directed or undirected, respectively.

The format of the input file containing the graph is as follows. The first line in the file contains two positive integers:

`n m`

where `n` is the number of vertices and `m` is the number of edges.

Each of the next `m` lines has four space separated fields of the form:

`<edge_ID> <vertex_u> <vertex_v> <weight>`

where `<edge_ID>` is an integer in the interval `[1,m]` denoting the identifier of an edge, `<vertex_u>` and `<vertex_v>` are two integers in the interval `[1,n]` denoting the vertices that are the endpoints of the edge, and `<weight>` is a non-negative floating point number denoting the edge weight. That is, $e_{ID} = (u, v) \in E$ with the given weight.

If the graph is directed, *i.e.*, parameter `<direction>` is `directed`, then each line in the file represents a directed edge from vertex `vertex_u` to vertex `vertex_v`, with weight `weight`. If the graph is undirected, *i.e.*, parameter `<direction>` is `undirected`, each line in the file represents an edge connecting vertex `vertex_u` and vertex `vertex_v`, with weight `weight`.

You must represent G using adjacency lists. Therefore, when you read the value of `n`, you should initialize an array of pointers of size `n` for the adjacency list for each vertex. As you read each edge, you should insert it into the appropriate adjacency list, at the front of the list. If the graph is directed, then the edge is inserted into vertex `vertex_u`'s adjacency list. If the graph is undirected, then the edge is inserted into both vertex `vertex_u`'s and vertex `vertex_v`'s adjacency list.

Once constructed, the adjacency lists do not change throughout the rest of the execution of your program.

1.2 The Design of a Variant of Dijkstra's Algorithm

In this section, we describe the variant of Dijkstra's algorithm that you need to design and implement. The input parameters for this algorithm contains the adjacency lists of $G = (V, E, w)$, a source node `source` $\in V$, an integer `destination`, and an integer `flag` $\in \{0, 1\}$.

If `destination` $\notin V$, your algorithm performs *single-source* shortest path computation from source vertex `source` to each vertex $v \in V \setminus \{$`source`$\}$ that is reachable from vertex `source`. If `destination` $\in V$, your algorithm performs *single-pair* shortest path computation (from vertex `source` to vertex `destination`) and stops when either vertex `destination` is extracted from the min-heap or the min-heap becomes empty, whichever comes first.

You need to modify Dijkstra's algorithm (from §24.3 of the textbook) as follows:

1. Instead of initializing a min-heap with all vertices $v \in V$, your algorithm should initialize the min-heap with only the source vertex `source`. Note that the total number of elements the heap may store, *i.e.*, its `capacity`, is set to $|V|$ while its current `size` is set to 1. The heap should not store elements with a key field equal to ∞.

2. During the relaxation process, if the key field of a vertex v that is not in the heap is reduced from ∞ to a real number, then v is inserted into the heap.

3. Instead of performing EXTRACT-MIN and relaxations until the heap is empty, your algorithm should terminate either when the heap is empty or the destination `destination` is extracted, whichever comes first. In case the destination `destination` is extracted from the heap, your algorithm should not perform any relaxations from it.

1.2.1 Outcomes

Depending the values of the source vertex `source`, and the destination vertex `destination`, a query may have one of the following four possible outcomes:

1. A shortest `source-destination` path is computed, and can be extracted using the `predecessor` field.
2. A `source-destination` path has been computed, but it is not known whether it is a shortest `source-destination` path.

3. No `source-destination` path has been computed, yet.
4. No `source-destination` path exists in the graph G.

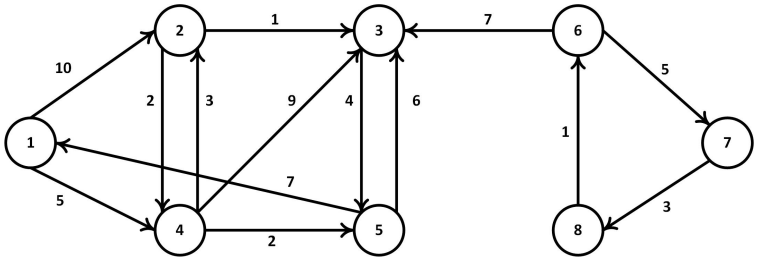

Figure 1: An edge weighted directed graph

We use the graph in Figure 1 to illustrate these possible outcomes.

Suppose that we have `source` $= 1$ and `destination` $= 4$. In this case, your program computes a *shortest path* `<1,4>` from vertex 1 to vertex 4 with a weight of 5 (outcome 1).

While relaxing the source vertex 1, the weight of vertex 2 drops from ∞ to 10 and it is inserted into the min-heap with a weight of 10. Hence, there is a *path* `<1,2>` from vertex 1 to vertex 2 with weight 10 but we do not know whether this is a shortest 1-2 path, with the current information. This is because vertex 2 has not been extracted from the min-heap (outcome 2).

We can tell that 1-v path has not been computed for $v \in \{3, 5, 6, 7, 8\}$ because each has a weight of ∞ (outcome 3).

Now assume that we have `source` $= 1$ and `destination` $\notin \{1, 2, 3, 4, 5, 6, 7, 8\}$. For any vertex $u \in \{2, 3, 4, 5\}$, a shortest `source`-u path is computed, but for any vertex $v \in \{6, 7, 8\}$, no `source-v` path exists in the given graph because these vertices are not reachable from vertex 1 (outcome 4).

1.3 Query Input and Output Format

After initializing the adjacency lists to represent the graph G as described in §1.1, your program reads an arbitrary number of queries from standard input (`stdin`). All output is to be written to standard output (`stdout`). (Of course, `stdin` can be redirected from a text file, and `stdout` can be redirected to a text file.) As described next, there are three kinds of queries to process. Be sure to echo each query prefixed by the string "Query: " before processing it.

1. `find <source> <destination> <flag>`

   Denote by `source`, `destination`, and `flag` the values of `<source>`, `<destination>`, and `<flag>`, respectively.

   This query is valid if `source` $\in V$, `destination` is an integer not equal to `source`, and `flag` $\in \{0, 1\}$.

   If the query is not valid, your program should write a message to `stdout` using the following format and wait for the next query.

   ```
   "Error: invalid find query\n"
   ```

   Now assume that the `find` query is valid. Your program should execute the variant of Dijkstra's algorithm as specified in §1.2, and wait for the next query.

   Note that important information will be computed during the execution of the algorithm. These include, for each vertex $v \in V$, the predecessor and the current distance from `source`. Your program may store other important information that can help to tell whether vertex v has been inserted into the min-heap or deleted from the min-heap. All of these information should be stored at vertex v, *without*

changing the adjacency lists for the given graph. These information will be needed for answering follow up `write` queries (to be described next), but will no-longer be needed when your program processes a new valid `find` query.

If `flag=1`, your program prints information about each insertion, deletion, and decrease-key operations in the min-heap as it computes the paths. In particular,

(a) when vertex v is inserted into the min-heap, print the information using the format:

`"Insert vertex %d, key=%12.4f\n"`

For example, when vertex 123 is inserted into the min-heap with (current) distance from `source` equal to 2.0, the corresponding information line should be

```
Insert vertex 123, key=      2.0000
```

(b) when vertex v is deleted from the min-heap, print the information using the format:

`"Delete vertex %d, key=%12.4f\n"`

(c) when the distance value of vertex v is decreased, print the information using the format:

`"Decrease key of vertex %d, from %12.4f to %12.4f\n"`

If `flag=0`, no min-heap operations are printed.

2. `write path <s> <d>`

The answer to this query is based on the computed information by the most recent valid `find` query:

`find <source> <destination> <flag>`

Denote by `s` and `d` the values of `<s>` and `<d>`, respectively, in this `write` query. Denote by `source` and `destination` the values of `<source>` and `<destination>`, respectively, in the most recent valid `find` query, if there has been one. Your program should act as follows.

(e1) If there has been no valid `find` query, your program should write a message to `stdout` using the following format and wait for the next query.

`"Error: no path computation done\n"`

(e2) If `s` $\neq$ `source` or `d` $\notin V \setminus \{s\}$, your program should write a message to `stdout` using the following format and wait for the next query.

`"Error: invalid source destination pair\n"`

Now Assume that neither (e1) nor (e2) happens. As described in section §1.2.1, there are four possible outcomes for this `write` query. Your program should perform the corresponding functions as outline below, and wait for the next query.

(a) A shortest `s`-`d` path is computed, and can be extracted using the `predecessor` field.
In this case, your output should include two lines, one the path itself, and the second its weight:

```
Shortest path: <s, s2, s3, ..., sk, d>
The path weight is: weight(s, s2, s3, ..., sk, d)
```

where `predecessor(s2)=s, predecessor(s3)=s2, ..., predecessor(d)=sk`, and `weight(s, s2, s3, ..., sk, d)` is a floating point value that is the weight of the path. Use `%12.4f` for printing a floating point weight.

(b) An `s`-`d` path has been computed, but it is not known whether it is a shortest `s`-`d` path.
In this case, your your output should include two lines, one the path itself, and the second its weight:

```
Path not known to be shortest: <s, s2, s3, ..., sk, d>
The path weight is: weight(s, s2, s3, ..., sk, d)
```

(c) No `s-d` path has been computed, yet.
In this case, your output should be, substituting the actual values of `s` and `d`:

```
No s-d path has been computed.
```

Here use `%d` for printing an integer.

(d) No `s-d` path exists in the graph G.
In this case, your output should be, substituting the actual values of `s` and `d`:

```
No s-d path exists.
```

3. `stop`
   Your program exits gracefully. This means that you should free all memory that you dynamically allocated before exiting the program.

You should also release dynamically allocated memory as soon as it is no longer needed. If one of your functions dynamically allocates memory but does not release it properly, your program may have a *memory leak*, which is a serious problem. Suppose that each call to a function leaks X bytes of memory. Then K calls to the function will leak KX bytes of memory. You may want to use `valgrind` to help you locate memory leaks in your program.

2 Modular Design

Your project must use a modular design, *i.e.*, you must provide at least the following source modules:

1. a main program, which coordinates all other modules;
2. a module that provides utility services including command line interpretation;
3. a module that implements the min-heap data structure and functions (this should be an update/extension of your heap module from an earlier project);
4. a module that implements the graph data structure and the variant of the single-source shortest path algorithm described in this project;
5. a `makefile` which compiles all modules and links them into the executable named `dijkstra`.

For each module other than the main program, you should have a **header file** which specifies the data structures and the prototypes of the functions in the module, and a **source file** which implements all of the functions specified in the header file.

3 A Study of Graph Characteristics

In this project we represented the graph using adjacency lists.

A *dense* graph is a graph $G = (V, E)$ in which the number of edges is close to the maximal number of edges. The opposite, a graph with only a few edges, is a *sparse* graph.

The graph density of simple graphs is defined to be the ratio of the number of edges $|E|$with respect to the maximum possible edges. For *undirected* simple graphs, the graph density D is:

$$D = \frac{|E|}{\binom{|V|}{2}} = \frac{2|E|}{|V|(|V|-1)}.$$

For `undirected` simple graphs the maximum possible edges is twice that of undirected graphs to account for the direction, so the density is:

$$D = \frac{|E|}{\binom{2|V|}{2}} = \frac{|E|}{|V|(|V|-1)}.$$

Compute the density of the graphs, directed and undirected, used in our test cases and summarize the results in your report. Are adjacency lists the better representation for the graphs used in this project, or would an adjacency matrix be a better choice? Briefly explain, using space complexity of the representation to justify your argument.

Compute a *depth-first search* (DFS) of each graph used in our test cases; see §22.3 of our textbook. Start the DFS at vertex 1, always visit vertices in numerical order, and count the number of trees added to the depth-first forest. Summarize the count obtained for each graph in your report.

Do you think a variant of Dijkstra's algorithm with source vertex `s` could take advantage of the results of a DFS starting from `s`? If so, describe how and under what circumstances. If not, why not? Briefly explain your answer.

4 Submission Instructions and Grading Policies

Submissions are always due before 11:59pm on the deadline date. Do not expect the clock on your machine to be synchronized with the one on Canvas! This project is due on Wednesday, 04/21/2021.
It is your responsibility to submit your project well before the time deadline!!! Late projects are not accepted.

An unlimited number of submissions are allowed. The last submission will be graded.

4.1 Requirements for Deadline

Two submissions are required before 11:59pm on Wednesday, 04/21/2021:

1. Submit your project to Gradescope using the submission link on Canvas. Detailed instructions for this will be provided on Canvas well before the due date.
2. Submit electronically, before 11:59pm on Wednesday, 04/21/2021 using the assignment submission link on Canvas, a zip[1] file named `yourFirstName-yourLastName.zip` containing the following:

   Project State (10%): In a folder (directory) named `State` provide a brief report (.pdf preferred) that addresses the following IN THIS ORDER:

   (a) Describe any problems encountered in your implementation for this project.
   (b) Describe any known bugs and/or incomplete query implementation for this project.
   (c) While this project is to be complete individually, describe any significant interactions with anyone (peers or otherwise) that may have occurred.
   (d) Cite any external books, and/or websites used or referenced.
   (e) A screen shot from your version control system showing commits over the development period of this project.

   Report (30%): In a folder (directory) named `Report` provide a brief report (.pdf preferred) that summarizes the results of your computations resulting from the study described in §3 and provides an answers to each question.

   Implementation and Correctness (60%): In a folder (directory) named `Code` provide:

   (a) Your well documented C/C++ source code files using a modular design as described in §2.
   (b) A file named `makefile` (with no file extension!) that compiles and links the individual executables into a single executable named `dijkstra`.
   Your program will be compiled and graded using Gradescope. If your program does not compile and execute on Gradescope, you will receive zero for correctness.

[1] **Do not** use any other archiving program except `zip`. **Do not** include spaces in your file name. **Do not** include any files in your zip that are not directly related to the project!

Assuming your submission compiles, the correctness of your program will be determined by running it with input that adheres to the specified format, some of which will be provided to you on Canvas prior to the deadline for testing purposes. Gradescope uses the Linux command `diff` to compare your output files to the expected output file.

42 points: You will earn 3 points for each of the 14 test cases posted in advance of the deadline that your program passes on Gradescope.

18 points: You will earn 3 points for each of the 6 test cases not posted in advance of the deadline that your program passes on Gradescope.

4.2 About the Graphs your Program will be Tested Against

Two graphs will be used in evaluating your program: `network01.txt` and `network02.txt`. `network01.txt` is the example graph shown in Figure 1. This small graph can help you to debug your program to make sure that your algorithm works properly, and your output is in the expected format.

`network02.txt` is huge undirected graph from **Real Datasets for Spatial Databases: Road Networks and Points of Interest**, available at `https://www.cs.utah.edu/ lifeifei/SpatialDataset.htm`. In particular, it is the **Road Network of North America (NA)**, with `175813` vertices and `179179` undirected edges. The file `network02.txt` was transformed from `NA.cedge` by

1. Adding the first line with `n` and `m`. This makes it easier for your program to dynamically allocate memory after reading in the first line.
2. Adding the vertex and edge identifiers by 1 so that the vertex identifiers are in the interval `[1,n]` and the edge identifiers are in the interval `[1,m]`.

4.3 About the Test Cases

Both `network01.txt` and `network02.txt` are available on Canvas. Each of the test cases includes the following:

- A bash script named `run.sh`. For the odd numbered unposted test cases, the script will be

  ```
  ./dijkstra network01.txt directed
  ```

  For the even numbered unposted test cases, the script will be

  ```
  ./dijkstra network02.txt undirected
  ```

- An input file `input.txt` containing commands and the corresponding output file `output.txt`. If you execute

  ```
  bash run.sh
  ```

  and your output file `myoutput.txt` is NOT identical to `output.txt`, your program does not pass the corresponding test case.