ARIZONA STATE UNIVERSITY

CSE 310 SLN 82170— **Data Structures and Algorithms** — Fall 2020

Instructor: Dr. Violet R. Syrotiuk

Project #3

No milestone submission; complete project due Tuesday, 11/24/2020

Problems on graphs are pervasive in computer science, and algorithms for working with them are fundamental. Hundreds of interesting computational problems are expressed in terms of graphs. This project examines the problem of how to compute shortest paths between vertices when each edge has an associated length or weight. Specifically, you will implement Dijkstra's algorithm to find the shortest paths from a given source vertex to all other vertices in a graph. An efficient implementation of Dijkstra's algorithm uses adjacency lists to represent the graph, and uses a min-priority queue of vertices, keyed by their current distance. This requires the implementation of a min-priority queue using a min-heap.

This project is an Accreditation Board for Engineering and Technology (ABET) required project as part of the accreditation process for our CS and CSE programs, and is not of my design.

Note: This project **must** be completed **individually**. Your implementation **must** use C/C++ and ultimately your code **must** run on the Linux machine `general.asu.edu`.

You must build all dynamic data structures, *i.e.*, the min-priority queue and the adjacency lists, by yourself from scratch. All memory management must be handled using only `malloc` and `free`, or `new` and `delete`. That is, you may not make use of any external libraries of any type for memory management! You may only use 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 me.

Remember that you **must** use a version control system as you develop your solution to this project. Your code repository must be private to prevent anyone from plagiarizing your work.

1 Implementation of Dijkstra's Algorithm

As already stated, an efficient implementation of Dijkstra's algorithm uses adjacency lists to represent the graph, and uses a min-priority queue of vertices, keyed by their current distance. Let's recall the definition of a min-priority queue.

1.1 Min-Priority Queue

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. Priority queues come in two forms: Max-priority queues and min-priority queues. Our focus is on a min-priority queue because Dijkstra's algorithm finds the shortest path to each vertex from a source, *i.e.*, it works to minimize the path length.

A min-priority queue supports four operations:

1. INSERT(S, x): Inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$.
2. MINIMUM(S): Returns the element of S with the smallest key.
3. EXTRACT-MIN(S): Removes and returns the element of S with the smallest key.
4. DECREASE-KEY(S, x, k): Decreases the value of element x's key to a new value k, where k is less than or equal to x's current key value.

Not surprisingly, we can use a min-heap to implement a min-priority queue. In a given application, elements of a priority queue correspond to objects in the application. For Dijkstra's algorithm, an element of a min-priority queue has three fields, corresponding to a vertex v, the predecessor of v, and the current known minimum distance to v. You are to build the priority queue keyed on the distance field of an element.

In this project, you must implement the functions of the min-priority queue using a min-heap. You should be able to adapt the functions of a max-heap discussed in class to implement a min-heap, and from them, the min-priority queue functions.

1.2 Dijkstra's Algorithm

As you know, Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are non-negative. Therefore, we assume $w(u, v) \geq 0$ for each directed edge $(u, v) \in E$.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.

You are to implement Dijkstra's algorithm using the psuedocode that follows, using a min-priority queue Q of vertices keyed by their distance values, and an adjacency list representation for G. The functions Initialize-Single-Source and Relax were provided and discussed in class.

Algorithm 1 Dijkstra(G, w, s)

```
Initialize-Single-Source$(G, s)$ {Initialize distance and predecessor of each vertex $v \in V$}
$S = \emptyset$ {$S$ is initially empty because no shortest-paths are determined yet}
$Q = V$ {Use Insert$(Q, v)$ to insert each vertex $v \in V$ into the min-priority queue $Q$}
while ( $Q \neq \emptyset$ ) do
  $u$ = Extract-Min$(Q)$ {Extract the vertex $u$ with current minimum distance from $Q$}
  $S = S \cup \{u\}$ {Add $u$ into the set $S$ of vertices whose final shortest-path has been determined}
  for each vertex $v$ adjacent to $u$ do
    Relax$(u, v, w)$ {If the distance to $v$ decreases to $d'$, then Decrease-Key$(Q, v, d')$}
  end for
end while
```

You are responsible for defining the structure (*i.e.*, `struct`) for the elements in the min-heap (and hence the min-priority queue), and the structure for the adjacency list representation of the graph. Be sure to place your definitions in appropriately named include files as described in §1.4.

1.3 Input and Output Format

First, your program is to read in a graph from `stdin` and construct its adjacency list representation. Recall that the adjacency list representation of a graph is an array (indexed by vertex), where the list for vertex v corresponds to a singly linked list of outgoing neighbours of v (the list of neighbours need not be ordered; you should be able to repurpose some of the you code for the hash table in Project 2).

In order to read the graph, the first line of input contains two integers n and m, which correspond to the number of vertices and the number of edges of the graph, respectively. This line is followed by m lines, where each line contains three integers: u, v, and w. These three integers indicate the information associated with an edge, *i.e.*, that there is a directed edge from u to v with weight w. Note that the vertices of the graph are indexed from 1 to n (and not from 0 to $n - 1$).

Following the input describing the graph, your program now processes commands, one per line. There are three commands:

- `stop`. On reading `stop`, your program must exit gracefully, with a return code of zero. Recall that, by convention, a return value of zero signals that all is well; non-zero values signal abnormal situations.

- `write`. On reading `write`, your program must write the graph to `stdout`. The output format of the `write` command is as follows: The first line should contain two integers, n and m, where n is the number of vertices and m is the number of edges in the graph, respectively. This line must be followed by n lines, one for each vertex. If vertex u has k neighbours (u, v_i) with weight w_i, $1 \leq i \leq k$, then for each vertex u, output:

  $u : (v_1; w_1); (v_2; w_2); \ldots ; (v_k; w_k)$

- `find s t flag`, where `flag` is either zero or one.

  On reading `find s t 1`, your program runs Dijkstra's shortest path algorithm to compute the shortest path from `s` to `t`, and prints out the length of this shortest path to `stdout`. The information output format is:

  `LENGTH:`ℓ

  where ℓ is the shortest path length.

  On reading `find s t 0`, your program runs Dijkstra's shortest path algorithm to compute a shortest path from `s` to `t`, and prints the actual path (sequence of vertices) to `stdout`. The information output format is:

  `PATH:`$s; v_1; v_2; \ldots ; v_k; t$

  where the sequence of vertices listed corresponds to the shortest path $(s, v_1); (v1, v_2) \ldots (v_k, t)$ computed of length k.

While your program reads in only one graph, it may be asked to compute s-t shortest paths for many different source-destination pairs s and t. Therefore, during the computation of the s-t shortest path, your program must not modify the given graph.

1.4 Modular Design

You should use modular design for this project. At the minimum, you must have:

- the `main` program in the file `main.cpp` (and corresponding include file `main.h`),
- the implementation of the min-priority queue using a min-heap in the file `heap.cpp` (and corresponding include file `heap.h`),
- the graph functions in the file `graph.cpp` (and corresponding include file `graph.h`),
- any utility functions in the file `util.cpp` (and corresponding include file `util.h`),
- a makefile named `makefile` that compiles and links the individual executables into a single executable named `dijkstra`, when the command `make dijkstra` is executed.

2 Submission Instructions

Submissions are always due before 11:59pm on the deadline date.

1. Due to the Provost's changes to the Fall 2020 semester, the time frame for this Project #3 was compressed. As a result, there is insufficient time to have a milestone graded. Nevertheless, I suggest that you aim to complete the min-heap implementation of the min-priority queue by the milestone deadline of Thursday, 11/12/2020.
2. The complete project is due on Tuesday, 11/24/2020. See §2.1 for requirements.

It is your responsibility to submit your project well before the time deadline!!! Late projects are not accepted. Do not expect the clock on your machine to be synchronized with the one on Canvas!

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

2.1 Requirements for the Full Project Deadline

Using the submission link on Canvas for the full Project #3, a zip[1] file named `yourFirstName-yourLastName.zip` containing the following:

Design and Implementation (40%):

1. Your zip file must unzip into, as a minimum, the files listed in §1.4 implementing the main program, the min-priority queue, graph operations, and any utility functions, and their associated include files. (10%)
2. The makefile named `makefile` must compile and link the individual executables into a single executable named `dijkstra`, when the command `make dijkstra` is executed on `general.asu.edu`. If your program does not pass this step, you will receive zero on this project. (10%)
3. Your code must be well documented, *i.e.*, provide sufficient comments for the variables and algorithms. (10%)
4. Reading the graph, and processing of commands. (10%).

Correctness (60%):

The correctness of your program will be determined by running your `dijkstra` program on the several data sets. Specifically, there are 30 test cases posted on Canvas and 30 test cases that are not posted.

1. Your program produces the correct output for a subset of the posted test cases. (30%)
2. Your program produces the correct output for an unposted set of test cases. (30%)

[1]**Do not** use any other archiving program except `zip`.