CSc 352 (Fall 2023): Assignment 7

Due Date: 11:59PM Fri, Oct 27

The purpose of this assignment is to to work with linked lists, memory allocation, representing a graph in c, command line arguments, reading from a file, and using free().

General Requirements

1. Your C code should adhere to the coding standards for this class as listed in the Documents section on the Resources tab for Piazza. This includes protecting against buffer overflows whenever you read strings.
2. Your programs should indicate if they executed without any problems via their *exit status*, i.e., the value returned by the program when it terminates:

Execution	Exit Status
Normal, no problems	**0**
Error or problem encountered	**1**

3. Under *bash* you can check the exit status of a command or program ***cmd*** by typing the command "**echo $?**" immediately after the execution of ***cmd***. A program can exit with status *n* by executing "`exit(n)`" anywhere in the program, or by having `main()` execute the statement "`return(n)`".
4. Remember your code will be graded on lectura using a grading script. You should test your code on lectura using the `diff` command to compare your output to that of the example executable.
5. To get full points your code should compile without warnings or errors when the `-Wall` flag is set in `gcc`
6. Anytime you input a string you must protect against a buffer overflow. Review slides 70 – 75 of the basic_C deck.
7. You must check the return values to system calls that might fail due to not being able to allocate memory. (e.g. Check that malloc/calloc don't return NULL) getline() is an exception to this rule.
8. Your code must run without errors using valgrind meaning that valgrind shows the line "ERROR SUMMARY: 0 errors" for all test cases.
9. **New Requirement: You must free all your memory before exiting to get full credit.**

Testing

Example executables of the programs will be made available. You should copy and run these programs on lectura to test your program's output and to answer questions you might have about how the program is supposed to operate. Our class has a home directory on lectura which is:

`/home/cs352/fall23`

You all have access to this directory. The example programs will always be in the appropriate **`assignments/assg#/prob#`** subdirectory of this directory. They will have the same name as the assigned program with "ex" added to the start and the capitalization changed to maintain camelback. So, for example, if the assigned program is **`theBigProgram`**, then the example executable will be named **`exTheBigProgram`**. You should use the appropriate UNIX commands to copy these executables to your own directory.

Your programs will be graded by a script. This will include a timeout for all test cases. There must be a timeout or programs that don't terminate will cause the grading script to never finish. This time out will never be less than 10 times the time it takes the example executable to complete with that test input and will usually be much longer than that. If your program takes an exceedingly long time to complete compared to the example code, you may want to think about how to clean up your implementation.

Makefiles

You will be required to include a Makefile with each program. Running the command:

`make` ***`progName`***

should create the executable file *progName*, where *progName* is the program name listed for the problem. The gcc command in your Makefile must include the **`-Wall`** flag. Other than that, the command may have any flags you desire.

Submission Instructions

Your solutions are to be turned in on the host **lectura.cs.arizona.edu**. Since the assignment will be graded by a script, it is important you have the directory structure and the names of the files exact. Remember that UNIX is case sensitive, so make sure the capitalization is also correct. For all our assignments the directory structure should be as follows: The root directory will be named **assg#**, where # is the number of the current assignment. Inside that directory should be a subdirectory for each problem. These directories will be named **prob#** where # is the number of the problem within the assignment. Inside these directories should be any files required by the problem descriptions. For this assignment the directory structure should look like:

```
assg7
   prob1
      shortestPaths.c
      Makefile
```

To submit your solutions, go to the directory containing your assg7 directory and use the following command:

turnin cs352f23-assg7 assg7

Problems

prob1: shortestPaths

This problem can be thought of, conceptually, as an elaboration of a previous programming problem involving reachability in graphs, i.e., determining whether there is a path between two vertices in an undirected graph (however, the actual programming is quite different in many respects). In this case, we assume that the input graph is connected, that edges have weights ("distances") associated with them, and our goal is to compute the shortest distance between two cities.

Terminology:

> For the purposes of this problem, a *name* is a string, consisting of upper and lower case letters only, of length at most 64 (See `man isalpha`). A *query* is a pair of names separated by whitespace.

Write a C program in a file **shortestPaths.c** together with a **Makefile** that creates an executable **shortestPaths** which behaves as specified below.

- **Invocation:** Your program will be invoked with a single command-line argument:

  ```
  shortestPaths   inFile
  ```

  where "`inFile`" is the name of a file which contains the description of the undirected graph.

- **Behavior:** After reading the file giving distances between cities, your program will repeatedly perform the following computation:

  1. read in a query from **stdin**;
  2. compute the shortest distance between the two names specified in the query; and
  3. print the distance so computed to **stdout** according to the format specified below.

  until no further input can be read.

- **Input:**

    1. *Input File Format:* The input file consists of a series of lines where each line has the form

        `name`$_1$ **`name`**$_2$ **`dist`**$_{12}$

        where **name**$_1$ and **name**$_2$ are names and **dist**$_{12}$ is the distance between them. **dist**$_{12}$ should be a non-negative integer. These arguments are separated by white space.

        Note each line describes an edge between two vertices. Since this is an undirected graph, the order of the names doesn't matter.

    2. *Query Format:* Each query is on a separate line and is a pair of names separated by whitespace. (See above under "**Terminology**".)

- **Assumptions:** You can assume the following:

    1. Each name read in, either from the input file (as part of the distance database) or from **stdin** (as part of a query) has length at most 64. As always, you still need to protect against buffer overflows.
    2. The graph described by the input file is connected.

- **Output:** Output should be printed to **stdout** using the statement

    ```
    printf("%d\n", val)
    ```

    where `val` is the value computed as the shortest distance between a pair of cities.

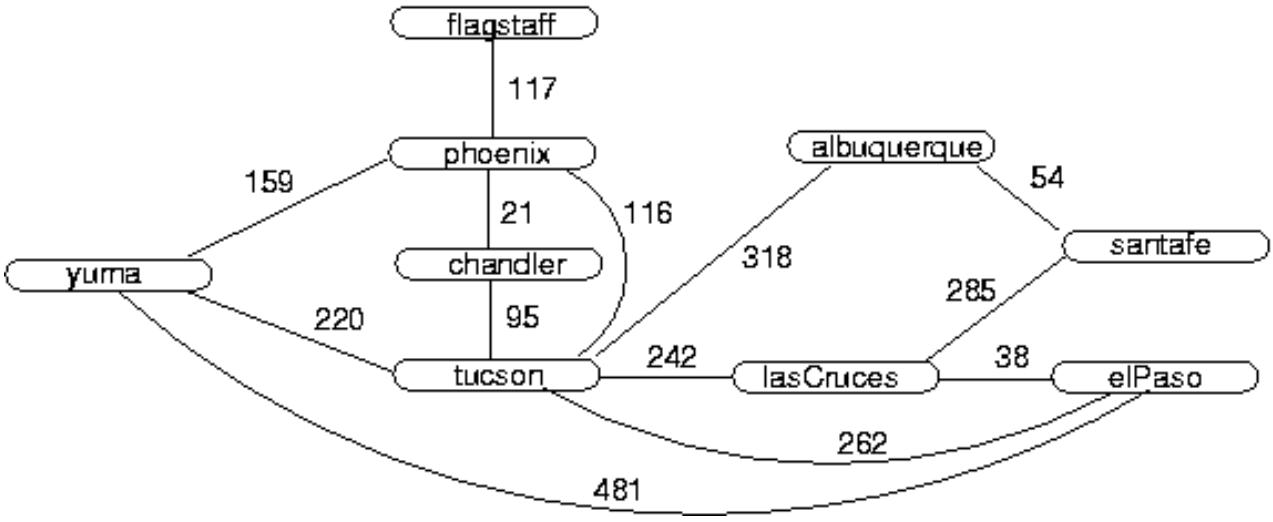
- **Error Conditions:**

    1. *Fatal errors:* Not enough command line arguments or input file cannot be opened for reading.
    2. *Non-fatal errors* Anything that does not meet the specification given above but is not a fatal error. Since the error is not fatal, your program should do something sensible (but give an appropriate error message) and continue execution.

        Examples (this is not an exhaustive list): too many command-line arguments (ignore extra arguments); input line not in the format specified (ignore the offending line); distance between some pair of cities is specified multiple times (ignore all but the first); city name(s) read from **stdin** not in the distance database read from the input file (ignore that input line). As always, if you wonder if something should be considered to be an error experiment with the example executable.

- **Example**: Shown below is an input file and a graphical portrayal of the distances it describes:

```
tucson      albuquerque 318
chandler    tucson      95
elPaso      tucson      262
tucson      phoenix     116
phoenix     flagstaff   117
yuma        elPaso      481
phoenix     yuma        159
tucson      yuma        220
lasCruces   tucson      242
lasCruces   elPaso      38
chandler    phoenix     21
lasCruces   santafe     285
santafe     albuquerque 54
```



A Shortest Path Algorithm

There are many different algorithms for computing shortest paths in graphs. The goal of a *single-source* shortest paths problem is to determine the shortest path from a given vertex in a graph to every other vertex. There are also *all-pairs* shortest paths algorithms that compute the shortest distance between each pair of vertices in the graph. We'll use the single-source version for simplicity. The one described here is known as *Dijkstra's single-source shortest path algorithm*: it assumes that the distances between cities (i.e., the weights on the edges of the graph) are all non-negative. Even though Dijkstra's algorithm is being described here, you're free to implement another algorithm (e.g. Prim's) if you so choose. I'm just describing this single algorithm for simplicity.

Suppose we have a graph ***G*** and a source vertex ***s***: we want to determine the shortest distance from the source ***s*** to the other vertices. (Strictly speaking, we want to know only the shortest distance between two given vertices. It turns out that, in order to do this, we need to compute the shortest distances to ``intermediate" vertices.)

The basic idea of the algorithm is to maintain a set of vertices for which we already know the shortest distance from the source ***s***; we'll *mark* these vertices ("marking" a vertex means setting a flag in the struct for that vertex, analogous to the way vertices were marked as "visited" in the reachability problem). As the algorithm executes, therefore, we have two classes of vertices:

- A set of *marked* vertices: for each such vertex ***a***, we know the exact value of the shortest distance from the source ***s*** to ***a***.
- The remaining vertices, which are unmarked: for each such vertex ***a***, we have an upper bound approximation ***d(a)*** to the shortest distance from the source ***s*** to ***a***. "Upper bound approximation" means that we may not know the exact shortest distance from ***s*** to ***a***, but we know that this distance is at most ***d(a)***.

The essential idea in the algorithm is to iterate over the vertices in the graph: in each iteration we are able to take an unmarked vertex and compute the exact shortest distance to it. The algorithm terminates when all vertices are marked (assuming that the graph is connected).

Data Structures

The data structure for this is in many ways similar to that for the graph reachability problem, with the following differences:

1. Each edge ***e*** now has an additional field ***dist***, an integer, which gives the value of the distance between the two endpoints of that edge.
2. Each vertex ***v*** now has an additional field ***minDist***, an integer that gives the shortest distance from the source to that vertex.

3. The graph is undirected, so every time you add an edge from A to B, add an identical edge from B to A.

Initialization

Initially, all we can say about the shortest distances is that the shortest distance from the source vertex to itself is **0**. For all the other vertices, we set the ***minDist*** field to *infinity* (Actually, to **`INT_MAX`** which is a macro set to the maximum int size. Include <limits.h> to use it)**.** Thus, this phase looks like:

```
for each vertex v {
  v.minDist = INT_MAX;
  unmark v;
}
src.minDist = 0;
```

Iteration

The main idea in the iteration is to find, at each iteration, a vertex ***u*** to which we already know the exact shortest distance from the source (as mentioned in the Initialization phase above, initially the only such vertex is the source itself), then use this to improve our estimate of the ***minDist*** field of all of the neighbors of ***u***.

Suppose that a vertex ***v*** is a neighbor of ***u***, where ***e*** is the edge between ***u*** and ***v***. Now ***v.minDist*** is our current approximation to the shortest distance from the source to ***v***. Suppose that we find that

`u.minDist + e.dist < v.minDist`

then (since by hypothesis we already know the shortest distance to ***u***) this means that we've managed to find a shorter path than before from the source vertex to ***v***, through the vertex ***u*** and then using the edge ***e***. This means that we now have a better approximation to the shortest distance from the source to ***v***. We accordingly set ***v.minDist*** to the new value.

As you can see, whenever a vertex ***v*** has its ***minDist*** field updated, the new value is obtained by adding an ***e.dist*** value to the ***minDist*** field of one of its neighbors. Under the assumption that all edges have nonnegative weights, i.e., ***e.dist*** **≥ 0** for all edges ***e***, this means that if ***w*** is an unmarked vertex with the smallest ***minDist*** value of all the unmarked vertices, then its value can't get any smaller. So we can mark the vertex ***w***, i.e., conclude that we've found the shortest distance from the source to ***w***. (Actually, during each round of iteration we do this step first, then update the ***minDist*** fields of ***w***'s neighbors: this is discussed below.)

The Overall Algorithm

The algorithm consists of the following steps:

1. Initialize, as discussed above.
2. **repeat** until no new vertex can be marked:
   - Let ***u*** be an unmarked vertex that has the smallest ***minDist*** value among all the unmarked vertices.
   - Mark the vertex ***u***.
   - For each vertex ***v*** that is a neighbor of ***u***:
     - let ***u*** and ***v*** be joined by the edge ***e***, where ***e.dist*** gives the distance between ***u*** and ***v***.
     - We improve our estimates of the field ***v.minDist*** as follows:

       n = ***u.minDist*** + ***e.dist***;
       if (*n* < ***v.minDist***) **then** ***v.minDist*** = *n*.

In your implementation, you should make sure that the initialization step is carried out for *each* query, i.e., each pair of cities for which we want to compute the shortest distance. Choose one of the two cities as the source (it doesn't matter which one), run the algorithm, and then print out the ***minDist*** field for the other one.