# CS 580U, Fall 2022
# Assignment 3

October 3, 2022

## 1 Instructions

### 1.1 Deadline

**Due: Wed Oct 12 11:30:00 AM EDT 2022** (i.e. at the start of class), as measured by the Brightspace timestamp.

### 1.2 Before You Begin

All programs will be tested on the Classroom machines in the N01 classroom and/or on Remote. If your code does not run on the system in this lab, it is considered non-functioning *even if it runs on your personal computer*. You can write your code anywhere, but always check that your code runs on the lab machines before submitting.

### 1.3 Testing Your Code

As with previous assignments, we will be testing your function implementations using a CUnit test suite. **Makefile** includes a target `test`, which will build the testing executable, which you then run without arguments.

### 1.4 Submitting Your Program

The file **grading.make** includes a target `make submit` which will create a .tar file named "USERNAME.tar", where USERNAME is your LDAP username (e.g. mcole8). Make sure you have saved all files, run `make clean && make check`, and observe that a directory has been created (which you can later delete by running `make clean` again), as well as a .tar file with the appropriate name. Upload this tar file this assignment on Brightspace by the deadline.

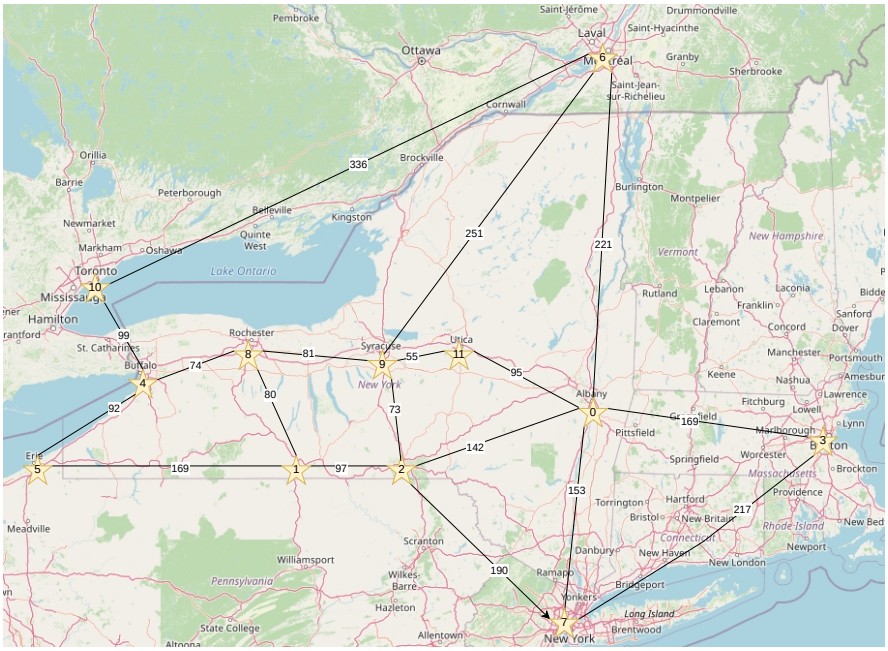


Figure 1: Highway mileages between cities in the Northeast.

## 1.5 Grading

The file **grading.make** includes a target `make check` which will build the test executable and run it. Results will be both printed to the screen by the executable, and recorded to a file called **feedback.log** by the Makefile. You will receive points according to the following rubric:

| Task | Points | Description |
|---|---|---|
| testSAFEADD | 15 | 3 points per test case |
| testMIN | 10 | 2 points per test case |
| testREADTABLE | 25 | 5 points for sz, 20 points for table |
| testDIJKSTRA | 40 | $3\frac{1}{3}$ points per test case |
| Coding Standards | 10 | At grader discretion |

Table 1: Grading rubric for Assignment 3.

The coding standards document is located at Brightspace.

# 2 Questions

This assignment is a running exercise, where we will implement Dijkstra's algorithm to determine the fastest route between cities in the Northeast using the interstate highway mileage (see Figure 1). We model our our highway map as an graph, where each city has been assigned a numbered vertex, and each highway segment is an undirected edge with a length equal to that highway segment's published mileage. Furthermore, we encode this graph as an adjacency list in **mileage.txt**. The first line of this file contains the number of vertices. Subsequent lines of the file contain three integers capturing an edge: the first two are the endpoint vertices, and the third is the length of the edge. Again, because our edges are undirected, **mileage.txt** contains a line `0 2 142`, but not a line `2 0 142` even though one can drive the highway between Binghamton (#2) and Albany (#0) in either direction. Also note that some vertices are not connected to each other, which means that their directly connecting edge has an infinite length that cannot be traveled. This represents that one cannot travel directly some cities to others (e.g. one cannot travel from Binghamton (#2) directly to Boston(#3), they must travel through either Albany (#0) or New York (#7)). Since C does not have an integral infinity value, we represent this infinite distance as `UINT_MAX` (4294967295, as defined in **limits.h**), which means we are limited to total travel distances and edge lengths of `UINT_MAX` minus one. In practice, this isn't a problem – `UINT_MAX` can contain a distance of over 4 billion miles, or 46 times the distance between the Earth and the Sun.

Note that when your implementations are graded, the input file may be different than that of **mileages.txt**: it may contain more or fewer cities, and/or mileages could be different for the existing edges. As a result, you cannot hardcode the size variable, nor the mileage table. You must also allocate (and free) the space for the table on the heap since you don't know its size at compile time.

Implement the following functions in **functions.c** using prototypes located in **functions.h**:

1. `unsigned int safe_add(unsigned int n1, unsigned int n2);`
2. `unsigned int min(unsigned int x, unsigned int y);`
3. `unsigned int **read_table(const char *filename, size_t *sz);`
4. `unsigned int dijkstra(unsigned int src, unsigned int dst, unsigned int **table, size_t`

## 2.1 Safe Addition

Dijkstra's algorithm requires a step called "relaxation", where the tentative minimum distance to a city is compared to the sum of the minimum distance from a visited city to the next city in question. Since we use `UINT_MAX` as a placeholder for infinite distance, we need to define a special addition function that prevents integer overflow:

$$safeadd(x, y) = \begin{cases} UINT\_MAX & \text{if } (x = UINT\_MAX) \text{ or } (y = UINT\_MAX) \\ UINT\_MAX & \text{if } (x + y) \geq UINT\_MAX \\ (x + y) & \text{otherwise} \end{cases}$$

Since we cannot add x and y then compare against `UINT_MAX` because the overflow will have already occurred, you will need to come up with a way to predict that an overflow would occur without actually causing the overflow.

| `n1` | `n2` | `safe_add()` |
|---|---|---|
| 4294967295 | 1 | 4294967295 (case 1) |
| 1 | 4294967295 | 4294967295 (case 1) |
| 2100000001 | 2194967295 | 4294967295 (case 2) |
| 2100000001 | 2194967293 | 4294967294 (case 3) |

Table 2: Example inputs and outputs for `unsigned int safe_add(unsigned int, unsigned int)`

## 2.2 Minimum

Likewise, relaxation requires us to select a minimum value from two values, so we will need to define a minimum function:

$$min(x, y) = \begin{cases} x & \text{if } x \geq y \\ y & \text{if } x < y \end{cases}$$

Note that if either x or y is `UINT_MAX`, then it follows that the return value would also be `UINT_MAX` since we cannot encode a larger value. Hint: this function is easiest to implement as a ternary statement.

| `x` | `y` | `min()` |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 2 | 2 | 2 |

Table 3: Example inputs and outputs for `unsigned int min(unsigned int, unsigned int)`

## 2.3 Read Table

This function must read a file encoding an adjacency list allocate space for a mileage table and populate the table, which is a two dimensional array:

1. The argument `filename` is a string containing the path to a file containing the adjacency list. The `read_table()` function will open a file descriptor to that path in read mode.

2. Read the size of the table (which is both the number of rows and number of columns) and set the pointer variable `sz` so that other functions can access this size.

3. Allocate a two-dimensional array (see Listing 1), and initialize its values to either zero (for self-edges) or `UINT_MAX` (all other edges).

4. Reads the edges line by line and set the cells in the two-dimensional array. Continue reading until you reach the end of the file. Hint: On success, `fscanf()` returns the number of input items successfully matched and assigned; this can be fewer than provided for, or even zero, in the event of an early matching failure. When there is no more input file to be read, the return value should then be zero.

5. Finally, the function closes the file descriptor and returns a pointer to the allocated table. Note that it is the responsibility of the *caller*, not the callee (this function) to free the allocated table (see Listing 2).

Listing 1: Code snippet demonstrating allocation of a two-dimensional array of 3 rows, 4 columns.

```
#include <stdio.h>
#include <stdlib.h>

int r = 3, c = 4, i, j, count;

int** arr = (int**)malloc(r * sizeof(int*));
  for (i = 0; i < r; i++)
    arr[i] = (int*)malloc(c * sizeof(int));


  count = 0;
  for (i = 0; i < r; i++)
    for (j = 0; j < c; j++)
      arr[i][j] = ++count;

  for (i = 0; i < r; i++)
    for (j = 0; j < c; j++)
      printf("%d ", arr[i][j]);
```

Listing 2: Code snippet demonstrating freeing a two-dimensional array.

```
for (int i = 0; i < r; i++)
  free(arr[i]);

free(arr);
```

## 2.4 Dijkstra's Algorithm

Dijkstra's algorithm is a single-source shortest path-finding algorithm. It proceeds as follows:

1. Mark all vertices unvisited. Create an array storing the vertices' visited/unvisited state.

2. Assign to every vertex a tentative distance value: set it to zero for our source (initial) vertex and to infinity for all other vertices. Store these distance values in an array, where the vertex numbering is mapped to an index into the array. During the run of the algorithm, the tentative distance of a vertex v is the length of the shortest path discovered so far between the vertex v and the starting vertex. Since initially no path is known to any other vertex than the source itself (which is a path of length zero), all other tentative distances are initially set to infinity. Set the source vertex as the current vertex.

3. For the current vertex, consider all of its reachable, unvisited neighbors and calculate their tentative distances through the current vertex. Compare the newly calculated tentative distance to the one currently assigned to the neighbor and assign it the smaller one (*relaxation*). For example, if the current vertex A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B through A will be $6 + 2 = 8$. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, the current value will be kept.

4. When we are done considering all of the unvisited neighbors of the current vertex, mark the current vertex as visited in the visited array. A visited vertex will never be checked again.

5. If the destination vertex has been marked visited (when planning a route between two specific vertex) or if the smallest tentative distance among the unvisited vertices in the visted array is infinity, then stop. The algorithm has finished.

6. Otherwise, select the unvisited vertex that is marked with the smallest tentative distance, set it as the new current node, and go back to step 3.

7. Once the destination vertex is marked as visited, free any memory this function has allocated, and return the entry in the distance array corresponding to the destination vertex. You should *not* deallocate the table memory because this function did not call `read_table()`, this is the responsibility of the `read_table()` caller's responsibility.

| src | dst | dijkstra() |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 239 |
| 0 | 5 | 397 |
| 1 | 5 | 169 |

Table 4: Example inputs and outputs for `dijkstra()`, assuming the table is the one created by `read_table()` having read mileages.txt.