ECE36800 Programming Assignment 3

Due Saturday, December 11, 2021, 11:59pm

This assignment covers learning objective 4: An ability to apply graph theoretic techniques, data structures, and algorithms for problem solving; learning objective 5: An ability to design and implement appropriate data structures and algorithms for engineering applications.

Grid traversal

This programming assignment is to be completed on your own. You will implement a program that involves finding a fastest way to travel from a location at the top of a grid and exit at a location at the bottom of the grid. Consider the following 4×5 (or 4-rows-by-5-rows) grid:

4	2	1	0	5
2	0	4	1	0
4	2	2	2	7
1	7	2	9	2

We label the rows from top to bottom 0 to 3 and the columns from left to right 0 to 4. Let the coordinates of location i be (r_i, c_i), where r_i is the index of the row that i is in, and c_i is the index of the column that i is in. The highlighted $(2,1)$-location of the grid, for example, stores the a non-negative value of 2 (stored as `short`).

The value at a location corresponds to the amount of time a visitor has to spend there once the visitor enters that location. After spending that amount of time at that location, the visitor moves to one of its adjacent locations. If a visitor enters the $(2,1)$-location at time t, the earliest the visitor can leave that location and enter a location adjacent to $(2,1)$ is at time $t+2$.

If a visitor enters the $(1,1)$-location, which stores the value 0, at time t, the visitor can immediately exit that location and enter a location adjacent to $(1,1)$ at time t.

Two locations, i of coordinates (r_i, c_i) and j of coordinates (r_j, c_j), are adjacent if $r_j = r_i - 1$ (j is above i), $r_j = r_i + 1$ (j is below i), $c_j = c_i + 1$ (j is to the right of i), or $c_j = c_i - 1$ (j is to the left of i). For the $(2,1)$-location, the adjacent neighbor above it is the $(1,1)$-location, the adjacent neighbor below it is the $(3,1)$-location, the adjacent neighbor to its right is the $(2,2)$-location, and the adjacent neighbor to its left is the $(2,0)$-location. For the boundary locations in the grid, they do not have four adjacent neighbors. They have only two adjacent neighbors (for the corner entries) or three adjacent neighbors (for the non-corner boundary entries).

For the given 4×5 grid, a visitor must enter the grid at one of the $(0,i)$-locations, $0 \le i \le 4$. It must exit the grid at one of the $(3,j)$-locations, $0 \le j \le 4$.

Suppose a visitor enters the grid at the $(0,0)$-location at time 0, the values stored in the following table show the respective earliest times when the visitor could enter the corresponding locations in the given grid:

0	4	6	7	7
4	6	6	7	8
6	6	8	8	8
10	8	10	10	15

For the entrance at the $(0,0)$-location at time 0, the highlighted path of $(0,0) \to (1,0) \to (2,0) \to (3,0)$ allows the visitor to exit the grid at the $(3,0)$-location the earliest in 11 time units. The visitor arrives at the $(3,0)$-location the earliest in 10 time units, as shown in the preceding table, spends 1 unit of time there before exiting. This path contains 4 locations in the grid.

Suppose a visitor enters the grid at the $(0,1)$-location at time 0, the values stored in the following table show the respective earliest times when the visitor could enter the corresponding locations in the given grid:

2	0	2	3	3
2	2	2	3	4
4	2	4	4	4
8	4	6	6	11

For the entrance at the $(0,1)$-location at time 0, the highlighted path of $(0,1) \rightarrow (1,1) \rightarrow (2,1) \rightarrow (2,2) \rightarrow (3,2)$ allows the visitor to exit the grid at the $(3,2)$-location the earliest in 8 time units.

Suppose a visitor enters the grid at the $(0,2)$-location at time 0, the values stored in the following table show the respective earliest times when the visitor could enter the corresponding locations in the given grid:

3	1	0	1	1
3	3	1	1	2
5	3	4	2	2
9	5	6	4	9

For the entrance at the $(0,2)$-location at time 0, the highlighted path of $(0,2) \rightarrow (0,3) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (2,2) \rightarrow (3,2)$ allows the visitor to exit the grid at the $(3,2)$ location the earliest in 8 time units.

The following table is obtained using the $(0,3)$-location as the entry point. The visitor could exit the grid in 7 time units at the $(3,2)$-location.

3	1	0	0	0
3	3	1	0	1
5	3	3	1	1
9	5	5	3	8

The following table is obtained using the $(0,4)$-location as the entry point. The visitor could exit the grid in 12 time units at the $(3,2)$-location.

8	6	5	5	0
8	8	6	5	5
10	8	8	6	5
14	10	10	8	12

There is another fastest path when the entry point is the $(0,4)$-location, as highlighted below:

8	6	5	5	0
8	8	6	5	5
10	8	8	6	5
14	10	10	8	12

Among all these entry points, the entry point at the $(0,3)$-location allows a visitor to enter the grid at the top and exit the grid at the bottom at the $(3,2)$-location the earliest in 7 time units. The corresponding path is $(0,3) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (2,2) \rightarrow (3,2)$, which contains 5 locations in the grid.

Deliverables

In this assignment, you are required to develop include file(s) and source file(s) that can be compiled with the following command:

```
gcc -std=c99 -pedantic -Wvla -Wall -Wshadow -O3 *.c -o pa3
```

It is recommended that while you are developing your program, you use the "`-g`" flag instead of the "`-O3`" flag for compilation so that you can use a debugger if necessary.

The executable `pa3` should be run as follows:

```
./pa3 binary_grid_file text_grid_file fastest_times_file fastest_path_file
```

Given a 2-dimensional grid stored in the (binary) input file `binary_grid_file`, the program produces three output files: `text_grid_file` stores the input grid in text form, `fastest_times_file` stores the fastest time to exit the grid at the bottom for each entry location at the top of the grid, and `fastest_path_file` stores the fastest path that allows a visitor to enter a location in the top row of the grid and exit at a location in the bottom row of the grid the earliest possible. Both `fastest_times_file` and `fastest_path_file` should be binary.

Binary grid file format

The `binary_grid_file argv[1]` is an input file in `binary` format. Given a grid of m rows and n columns, $0 < m,n \leq$ `SHRT_MAX`, the first two `short`'s stored in `binary_grid_file` are m and n, respectively. Here, `SHRT_MAX` is the largest `short` integer (see `limits.h`).

After those two `short`'s, the file is followed by a total of $m \times n$ `short`'s. All $m \times n$ `short`'s are within the range of $[0,$ `SHRT_MAX`$]$.

Out of these $m \times n$ `short`'s, the first n `short`'s are in row 0. Out of these n `short`'s, the first `short` is at column 0 and the last `short` is at column $n-1$. The next n `short`'s are in row 1. The last group of n `short`'s are in row $m-1$.

The total size of the file is $(2+mn) \times$ `sizeof(short)` bytes. You may assume that all input files are all correct.

In general, the sample files given to you are named as `m_n.b` where `m` is the number of rows and `n` is the number of columns. However, you should always determine the true dimensions of a given grid from the first two `short`'s of the input file. The file `4_5.b` provided to you for this assignment contains the example given earlier.

Suppose we run `pa3` with the following command

```
./pa3 4_5.b 4_5.t 4_5.f 4_5.p
```

The file `4_5.t` should store the 2-dimensional grid in text form. The file `4_5.f` should store the fastest times to exit the grid for all entry locations in binary form. The file `4_5.p` should store the fastest path in binary form. These files are also provided to you for this assignment.

Text grid file format

The first output file `argv[2]` is simply a conversion of the binary grid file to a text form. Again, we assume that there are m rows and n columns in the binary file. The first line is printed with the format `"%hd %hd\n"`, where the first `short` is the number of rows, m, and the second `short` is the number of columns, n.

After that, there should be m lines. Each line should print n `short`'s, where each `short` is printed with the format `"%hd"`. There should be a space (`' '`) character between a pair of consecutive `short`'s. There should be a newline character (`'\n'`) immediately after the last `short` in each line (row). There should not be a space character after the last `short` in a line.

Fastest times file format

The second output file `argv[3]` stores the fastest time to exit the grid for each entry location in the top row in binary form.

The file first stores the number of entry locations in the top row of the input grid. This should correspond to the number of columns in the given input grid. The number of columns should be stored as an `short`.

Let n be the number of columns in the given grid. The file next stores n `int`. Assuming that we label the `int` from 0 to $n-1$. The ith `int`, $0 \leq i \leq n-1$, is the fastest time to exit at the bottom of the grid when the entry point is the $(0,i)$-location.

The total size of the fastest-times file is $(\texttt{sizeof(short)} + n \times \texttt{sizeof(int)})$ bytes.

Fastest path file format

The third output file `argv[4]` stores the fastest path information in binary form.

The file first stores the fastest time it would take for a visitor to enter at a location in the top row of the input grid and exit at a location in the bottom row of the input grid. The fastest time should be stored as an `int`.

Then, we store in the file an `int` for the number of locations that constitute the path. Let l be the number of locations that constitute the path, the output file next contains coordinates of the l entries that form the path. Each location is stored as a pair of `shorts`, the (row, column) coordinates of the location, with the row coordinate followed by the column coordinate. The order in which the coordinates appear in the output file should follow the order in which the corresponding locations appear in the path. The path should start with a location in the top row of the grid and end with a location in the bottom row of the grid.

The total size of the path file is $(2 \times \texttt{sizeof(int)} + 2l \times \texttt{sizeof(short)})$ bytes.

Return value of `main` function

The `main` function should simply return `EXIT_FAILURE` if the argument count is incorrect.

If the given input file cannot be opened, your program should terminate and return `EXIT_FAILURE`.

Now, assume that the given input file can be opened. If in the process of determining the fastest path, your program encounters a failure in memory allocation or a failure in writing to the output files, your program should gracefully exit and return `EXIT_FAILURE`. We will test your program with valid input files of reasonable sizes. Therefore, it is unlikely that you will have to return `EXIT_FAILURE` when you could open an input file.

Of course, your program should return `EXIT_SUCCESS` otherwise.

Electronic Submission

The project requires the submission (electronically) of the C-code (source and include files) through Brightspace. You should create and submit a zip file called `pa3.zip`, which contains the `.h` and `.c` files. Your zip file should not contain a folder.

```
zip pa3.zip *.c *.h
```

You should submit `pa3.zip` to Brightspace.

If you want to use a `Makefile` for your assignment, please include the `Makefile` in the zip file. In that case, you can create the zip file as follows:

```
zip pa3.zip *.c *.h Makefile
```

If the zip file that you submit contains a `Makefile`, we use that file to make your executable (by typing "make pa3" at the command line to create the executable called `pa3`).

Grading

The assignment will be graded based on the two tasks performed by your program. The first output file accounts for 10%, the second output file accounts for 60%, and the third output file accounts for 30%.

It is important that all the files that have been opened are closed and all the memory that have been allocated are freed before the program exits. Memory leaks or memory errors reported by `valgrind` will result in a 50-point penalty.

Be aware that we set a time-limit for each test case based on the size of the test case. If your program does not complete its execution before the time limit for a test case, it is deemed to have failed the test case.

What you are given

You are given 4 sample binary grid files (`4_5.b`, `5_4.b`, `5_5.b`, `7_7.b`) and the corresponding text grid files (`4_5.t`, `5_4.t`, `5_5.t`, `7_7.t`), fastest times files (`4_5.f`, `5_4.f`, `5_5.f`, and `7_7.f`), and path files (`4_5.p`, `5_4.p`, `5_5.p`, and `7_7.p`). It is possible that a binary grid file may contain multiple paths that allow a visitor to enter and exit a grid in the shortest time possible. Your program should not try to match the path exactly. However, the shortest time that it took to enter and exit a grid should match.

To help you to understand the fastest times files and path files, we also provide the text version of these files (`5_4.ft` and `5_4.pt`, `4_5.ft` and `4_5.pt`, `5_5.ft` and `5_5.pt`, and `7_7.ft` and `7_7.pt`).

In each `.ft` file, the first line stores the number of columns in the given grid file (printed with the format `"%hd\n"`). Let n be the number of columns. The second line has n `int`, where each `int` is printed with the format `"%d"`. There is a space (`' '`) character between a pair of consecutive `int`'s. There is a newline character (`'\n'`) immediately after the last `int`. Assuming that we label the `int` from 0 to $n-1$. The ith `int`, $0 \leq i \leq n-1$, is the fastest time to exit at the bottom of the grid when the entry point is the $(0,i)$-location.

In each `.pt` file, the first line stores the (fastest) time (printed with the format `"%d\n"`). The second line stores the number of locations in the path (printed with the format `"%d\n"`). Each subsequent line stores the (row, column) coordinates of an entry (printed with the format `"%hd %hd\n"`).

Additional information

You may want to write a program that allows you to convert from a text grid file into a binary grid file. This allows you to easily create other examples to test your programs. Similarly, you may want to create programs to convert from a binary fastest times file (or binary path file) to a text fastest times file (or text path file) for testing.