Programming Assignment Three: Priority Queue and its Application

Out: Oct. 28, 2018; Due: Nov. 11, 2018.

I. Motivation

This project will give you experience in implementing priority queues using C++. You will also empirically study the efficiency of different implementations.

II. Project Overview

In this project, you are given a rectangular grid of cells. Each cell has a number indicating its weight, which is the cost of passing through the cell. You can assume the weights are positive integers. The input will give you the starting coordinate and the ending coordinate. Your task is to use priority queue to find the shortest path from the source cell to the ending cell.

III. Input

You will read from the **standard input**. (For the ease of testing, you can write each test case in a file and then use Linux file redirection function "<" to read from the file.)

The format of the input is as follows:

```
<width m>
<height n>
<Start x> <Start y>
<End x> <End y>
<W(0,0)> <W(1,0)> … <W(m-1,0)>
…
<W(0,n-1)> <W(1,n-1)> … <W(m-1,n-1)>
```

The first and the second line give the width m and the height n of the grid, respectively. They are positive integers. The third and the fourth line give the starting coordinate and the ending coordinate, respectively. They are non-negative integers within the valid range. The upper left corner has the coordinate (0, 0). The x-coordinate increases from left to right and the y-coordinate increases from top to bottom. The remaining lines give the weights of the cells in the grid. They represent a two dimensional array of n rows and m columns, as shown above. $W(i,j)$ is the weight of the cell at coordinate (i,j) $(0 \le i \le m-1, 0 \le j \le n-1)$. The weights are all positive integers.

For example, we have an input:

```
4
```

```
3
0 0
3 1
5 1 2 3
2 1 5 6
7 1 1 1
```

It specifies a grid of width 4 and height 3. We want to find the shortest path form point (0, 0), which has weight 5, to the point (3, 1), which has weight 6.

You can assume that the testing cases are semantically and syntactically correct. Furthermore, you can assume that **the starting point and the ending point are not the same one**.

IV. Algorithm

In this project, we ask you to find the shortest path between two given cells in a grid. On a valid path, you can **only either go horizontally or vertically** from one cell to an adjacent cell; you **cannot go diagonally**. The length of a path is defined as the sum of the weights of all the cells on that path.

The algorithm we will use in this project is pretty similar to the Lee's wire routing algorithm that we previous mentioned in Ve280 (the slides of that algorithm can be found in the Progamming-Assignment-Three-Related-Files.zip). Recall that in that problem the grid cell is of unit weight. We apply a queue to find the shortest path between two points. In order to find the shortest path in a grid with non-uniform cell weight, we need to apply a min priority queue. For this project, for simplicity, we assume that there are no blocked cells in the grid. Below is the pseudo-code of the algorithm:

```
Let PQ be a priority queue;
start_point.pathcost = start_point.cellweight;
Mark start_point as reached;
PQ.enqueue(start_point);
while(PQ is not empty) {
    C = PQ.dequeueMin(); // The key is cell's pathcost
    for each neighbor N of C that has not been reached {
        N.pathcost = C.pathcost + N.cellweight;
        mark cell N as reached;
        mark the predecessor of N as C;
        // I.e., N is reached from C.
        if(end_point == N) {
            trace_back_path(); // Trace and print the path
                               // through predecessor info
            return;
        }
```

```
            else PQ.enqueue(N);
        }
}
```

The primary key of the min priority queue is the path cost of a cell, which is the minimal length from the starting point to the cell. (Updating the path cost using the above procedure will guarantee that the `pathcost` entry of a cell stores the minimal path length from the starting point to the cell.) The function `trace_back_path()` obtains and prints the final path.

In order for you to get the same result as ours, we require:

1. The visit of the neighbors starts form the right neighbor and then goes in the clockwise direction, i.e., right, down, left, up. For those cells on the boundary, they may not have a certain neighbor. Then you just skip it.
2. During a `dequeueMin()` operation, if multiple cells have the same smallest path cost, choose the cell with the smallest x-coordinate. Furthermore, if there are multiple cells with the same x-coordinate, choose the cell with the smallest y-coordinate.
3. Function `trace_back_path()` uses the predecessor information you have recorded during the search to recover the path. The predecessor of a cell N is the cell from which you reached N.

V. Command Line Input

Your program should be named **`main`**. It should take the following case-sensitive command-line options:

1. `-i, --implementation`: a required option. It changes the priority queue implementation at runtime. An argument should immediately follow that option, being BINARY, UNSORTED, or FIBONACCI to indicate the implementation (see Section VII Implementations of Priority Queues).
2. `-v, --verbose`: an optional flag that indicates the program should print additional outputs (see Section VI Output).

Examples of legal command lines:

- `./main --implementation BINARY < infile.txt`
- `./main --verbose -i UNSORTED < infile.txt > outfile.txt`
- `./main --verbose -i FIBONACCI`

Note that the first two calls read the input stored in the `infile.txt`. The third call reads from the standard input.

Examples of illegal command lines:

- `./main < infile.txt`

  No implementation is specified. Implementation is a required option.
- `./main --implementation BINARY infile.txt`

You are not using input redirection "<" to read from the file `infile.txt`.

We require you to realize the above requirement using the function **`getopt_long`**. See its usage and an example at
http://www.gnu.org/software/libc/manual/html_node/Getopt.html#Getopt

In testing your program, we will supply correct command-line inputs, but you are encouraged to detect and handle errors in the command-line inputs.

VI. Output

All outputs should be printed to the **standard output**.

Without 'verbose' option, the output of `main` should be as follows:

```
The shortest path from (<start x, start y>) to (<end x, end y>)
is <length of the shortest path>.
Path:
(<start x, start y>)
(<cell_1 x, cell_1 y>)
...
(<end x, end y>)
```

For the example input shown in Section III, the output should be:

```
The shortest path from (0, 0) to (3, 1) is 16.
Path:
(0, 0)
(1, 0)
(1, 1)
(1, 2)
(2, 2)
(3, 2)
(3, 1)
```

For 'verbose' mode, you should print out each step in detail:

```
Step <x>
Choose cell (<cell_0 x, cell_0 y>) with accumulated length <l_0>.
Cell (<cell_1 x, cell_1 y>) with accumulated length <l_1> is added
into the queue.
...
Cell (<cell_i x, cell_i y>) with accumulated length <l_i> is added
```

```
into the queue.
```

Until you arrive at the ending point, print:

```
Step <y>
Choose cell (<cell_0 x, cell_0 y>) with accumulated length <l_0>.
Cell (<cell_1 x, cell_1 y>) with accumulated length <l_1> is added
into the queue.
...
Cell (<cell_i x, cell_i y>) with accumulated length <l_i> is added
into the queue.
Cell (<cell_i+1 x, cell_i+1 y>) with accumulated length <l_i+1>
is the ending point.
The shortest path from (<start x, start y>) to (<end x, end y>)
is <length of the shortest path>.
Path:
(<start x, start y>)
(<cell_1 x, cell_1 y>)
...
(<end x, end y>)
```

Note that in the last step, before you arrive at the ending point (i.e., `cell_i+1`), you may first visit and enqueue some other cells (i.e., `cell_1, ..., cell_i`). You need to treat them in the normal way.

For the example input shown in Section III, the output should be:

```
Step 0
Choose cell (0, 0) with accumulated length 5.
Cell (1, 0) with accumulated length 6 is added into the queue.
Cell (0, 1) with accumulated length 7 is added into the queue.
Step 1
Choose cell (1, 0) with accumulated length 6.
Cell (2, 0) with accumulated length 8 is added into the queue.
Cell (1, 1) with accumulated length 7 is added into the queue.
Step 2
Choose cell (0, 1) with accumulated length 7.
Cell (0, 2) with accumulated length 14 is added into the queue.
Step 3
Choose cell (1, 1) with accumulated length 7.
Cell (2, 1) with accumulated length 12 is added into the queue.
Cell (1, 2) with accumulated length 8 is added into the queue.
Step 4
Choose cell (1, 2) with accumulated length 8.
Cell (2, 2) with accumulated length 9 is added into the queue.
```

```
Step 5
Choose cell (2, 0) with accumulated length 8.
Cell (3, 0) with accumulated length 11 is added into the queue.
Step 6
Choose cell (2, 2) with accumulated length 9.
Cell (3, 2) with accumulated length 10 is added into the queue.
Step 7
Choose cell (3, 2) with accumulated length 10.
Cell (3, 1) with accumulated length 16 is the ending point.
The shortest path from (0, 0) to (3, 1) is 16.
Path:
(0, 0)
(1, 0)
(1, 1)
(1, 2)
(2, 2)
(3, 2)
(3, 1)
```

Note that no space is attached after each line, and one space is attached after the comma in each coordinate. The step number starts from 0.

VII. Implementations of Priority Queues

We have provided a header file, `priority_queue.h`, which defines a templated abstract base class `priority_queue`. For this project, you are required to write three templated implementations of priority queue as the derived class of `priority_queue`: *binary heap*, *unsorted array-based heap*, and *Fibonacci heap*. To implement these priority queue variants, we have given you three separate files `binary_heap.h`, `unsorted_heap.h`, and `fib_heap.h` containing all the definitions for the functions declared in `priority_queue.h`. You will need to fill in the empty function definitions in these three header files. These files will also specify more information about each priority queue type, including runtime requirements and a general description of the implementation.

Fibonacci heap is an advanced data structure. You can study it by referring to the Fibonacci-Heap.pdf in Progamming-Assignment-Three-Related-Files.zip. We also show you the pseudo-code of Fibonacci heap in Section XIII Appendix.

You are **not** allowed to modify `priority_queue.h` in any way. Nor are you allowed to change the interface (names, parameters, and return types) of the functions that we give you in any of the provided headers. You are allowed to add your own private helper functions and variables as needed in `binary_heap.h`, `unsorted_heap.h` and `fib_heap.h`, so long as you still follow the requirements outlined in both the specifications and the comments in the provided files. You should not construct your program such that one priority queue

implementation's header file is dependent on another priority queue implementation's header file.

Note: We may compile your priority queue implementations with our own code to ensure that you have correctly and fully implemented them. To ensure that this is possible (and that you do not lose credit for these test cases), do not define your main function in one of the headers.

For the templated version of priority queue, it takes two template parameters `TYPE` and `COMP`, where `COMP` is a functor. An example of functor that can be used with the priority queue in this project is given to you in the file `test_heap.cpp` (which can be found in the `Priority-Queue-Test` folder within Progamming-Assignment-Three-Related-Files.zip). For more details, please read the example and the comments.

VIII. Implementation Requirements and Restrictions

Your code for solving the shortest path problem should call the priority queue implementations you write. Except some constraints on `priority_queue.h`, `binary_heap.h`, `unsorted_heap.h`, and `fib_heap.h` you are free to write your own `.cpp` and `.h` files to solve the shortest path problem.

You **are** allowed to `use std::vector, std::list, and std::deque`. You are **not** allowed to use other `STL` containers. Specifically, this means that the use of `std::stack`, `std::queue`, `std::priority_queue`, `std::set`, `std::map`, `std::unordered_set`, `std::unordered_map`, and the 'multi' variants of the aforementioned containers are forbidden.

You are **not** allowed to use `std::partition`, `std::partition_copy`, `std::stable_partition`, `std::make_heap`, `std::push_heap`, `std::pop_heap`, `std::sort_heap`, or `std::qsort`.

Furthermore, you may **not** use any `STL` component that trivializes the implementation of your priority queues. If you are not sure about a specific function, ask us.

IX. Programming Hints

1. To improve the performance of your program, we recommend you to add `-O2` option when compiling your programs.
2. For large test cases, I/O could be the runtime bottleneck. To reduce the I/O time, we recommend you to put the following two statements at the beginning of your main function:

```
std::ios::sync_with_stdio(false);
std::cin.tie(0);
```

X. Testing

We have provided you a `test_heap.cpp` to test your binary heap implementation. It can be found in the `Priority-Queue-Test` folder within Progamming-Assignment-Three-Related-Files.zip. To test, copy the original `prority_queue.h` and your modified `binary_heap.h` into the `Priority-Queue-Test` folder and type `make` to compile a program `test_heap`. The output of the program should be exactly like what is in the file `test_heap.out`. (Note: The `Makefile` in that folder also shows you how to compile codes involving templates.) You can modify `test_heap.cpp` to test your other implementations.

In the folder `Shortest-Path-Test` within Progamming-Assignment-Three-Related-Files.zip, we include two small test cases for the shortest path problem. They are `test-1.in` and `test-2.in`. You can use them as inputs by applying the Linux input redirection function "<". The outputs with and without the `-v` option are `*-verbose.out` and `*-brief.out`, respectively. Use `diff` to compare your output with these outputs.

These are the minimal amount of tests you should run to check your program. Those programs that do not pass these tests are not likely to receive much credit. You should also write other different test cases yourself to test your program extensively.

You should also check whether there is any memory leak using `valgrind` as we discussed before. The `Makefile` in the `Priority-Queue-Test` folder shows you a way to put `valgrind` command into a `Makefile`. To check whether `test_heap` program has memory leak, type `make memcheck`.

XI. Performance Comparison

We also ask you to compare the performance of the three priority queue implementations. To do this, you should first generate random inputs with different grid sizes. Then you should apply your implementations to these inputs. Finally, you should plot a figure showing the runtime of each implementation versus the grid size. For comparison purpose, you should plot three curves corresponding to the three implementations in the same figure. (You do not need to upload the source codes for this comparison program.)

Hint:

1. You may want to write another program that calls different implementations to do this study.
2. You can use mrand48() to generate integers uniformly distributed in the range $[-2^{31}, 2^{31}-1]$.
3. You can use clock() function to get the runtime. See http://www.cplusplus.com/reference/ctime/clock/

4. For simplicity, you can test on grid of which the width and the height are the same. With this, the x-axis in the plotted figure is the width. Choose the starting point to be the upper left corner of the grid and the ending point to be the lower right corner.
5. The runtime of your algorithm also depends on the detailed layout of the grid, not just its width. Thus, for each width you pick, you should generate a number of grids with that width and obtain the average runtime over all these grids. Also, for fair comparison, the same set of grids should be applied to all the three implementations.
6. You should try at least 5 representative widths.

XII. Submitting and Due Date

You should submit all the source code files, a `Makefile`, and a report of the runtime comparison. The `Makefile` compiles a program named `main`, which solves the short path problem. The report should be in pdf format. It should include a description of the experiment setup. At the end of your report, please attach your source code for the program that solves the short path problem. See announcement from the TAs for details about how to submit these files. The submission deadline is 11:59 pm on Nov. 11th, 2018.

XIII. Grading

Your program will be graded along five criteria:

1. Functional correctness
2. Implementation constraints
3. General style
4. Performance
5. Report on the performance study

Functional correctness is determined by running a variety of test cases against your program, checking your solution using our automatic testing program. We will grade Implementation constraints to see if you have met all of the implementation requirements and restrictions. General style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. Part of your grade will also be determined by the performance of your algorithm. We will test your program with some large test cases. If your program is not able to finish within a reasonable amount of time, you will lose the performance score for those test cases. Finally, we will also read your report and grade it based on the quality of your performance study.

XIV. Appendix

Pseudo-code for Fibonacci heap:

```
Make_Fibonacci_Heap() {
  H.n = 0; // n stores the number of elements in the heap
  H.min = NULL; // min refers to the minimal element in the heap
  return H;
}

Fibonacci_Heap_Get_Min(H) {
  return H.min;
}

Fibonacci_Heap_Enqueue(H, key) {
  create a new node x;
  x.degree = 0;
  x.parent = NULL;
  x.child = NULL;
  x.key = key;
  // x.mark = FALSE; // This statement is commented out, because you
                     // do not need it in this project.
  if(H.min == NULL) {
    create a root list for H only containing x;
    H.min = x;
  }
  else {
    insert x into the root list of H;
    if(x.key < H.min.key)
      H.min = x;
  }
  H.n = H.n + 1;
}

Fibonacci_Heap_Dequeue_Min(H) {
  z = H.min;
  if(z != NULL) {
    for(each child x of z) {
      add x to the root list of H;
      x.parent = NULL;
    }
    remove z from the root list of H;
    H.n = H.n - 1;
    if(H.n == 0) H.min = NULL;
```

```
    else Consolidate(H);
  }
  return z;
}

Consolidate(H) {
  let A[0..D(H.n)] be a new array; // D(N) is the maximum degree of
                                   // any node in a Fibonacci heap of N nodes
  for(i = 0 to D(H.n))
    A[i] = NULL;
  for(each node w in the root list of H) {
    x = w;
    d = x.degree;
    while(A[d] != NULL) {
      y = A[d];
      if(x.key > y.key) exchange x with y;
      Fibonacci_Heap_Link(H, y, x);
      A[d] = NULL;
      d = d + 1;
    }
    A[d] = x;
  }
  H.min = NULL;
  for(i = 0 to D(H.n)) {
    if(A[i] != NULL) {
      if(H.min == NULL) {
        create a root list for H only containing A[i];
        H.min = A[i];
      }
      else {
        add A[i] to the root list of H;
        if(A[i].key < H.min.key)
          H.min = A[i];
      }
    }
  }
}
Fibonacci_Heap_Link(H, y, x) {
  remove y from the root list of H;
  make y a child of x;
  x.degree = x.degree + 1;
  // y.mark = FALSE; // This statement is commented out, because you
                     // do not need it in this project.
}
```