Programming Assignment 7 (Priority Queue, and Dijkstra's algorithm)

Department of Computer Science, University of Wisconsin – Whitewater
Data Structure (CS 223)

1 Overview

We are essentially going to:

- **Merge k sorted arrays into a single sorted array using a priority queue**
- **Implement Dijkstra's Single Source Shortest Path algorithm**.

To this end, **your task is to implement the following methods**:

- `kWayMerge` method in `MergeSortedArrays.h`/`MergeSortedArrays.java`
- `readWeightedGraph` method in `Graph.h`/`Graph.java`
- `executeDijkstra` method in `Dijkstra.h`/`Dijkstra.java`

The project also contains additional files (which you do not need to modify).
Use `TestCorrectness.cpp`/`TestCorrectness.java` to test your code.
Output is provided separately in the `ExpectedOutput` file. Should you want, you can use `www.diffchecker.com` to tally the output.

2 Priority Queue

In this assignment you are going to use a priority queue. To this end, you will use the in-built class `priority_queue` in C++ or the `PriorityQueue` class in Java. The priority queue will contain objects of the `Element` class. Naturally, you will use the constructor of the `Element` class to create these objects; note that the constructor first accepts an item and then its priority. Although ideally one would use a generic item and possibly a generic priority as well, since it will suffice our purposes, to keep things simple, both items and priorities are integers in the implementation here.

To use the priority queue, we need to provide it with a comparator using which we can compare two items in the priority queue (i.e., compare the priorities of two items). Here, we will use the comparator described by the `ElementComparator` class. Let's now look at some of the syntaxes:

C++

- To create: `priority_queue<Element, vector<Element>, ElementComparator> pq;`
- To get the size: `pq.size();`

- To insert an object `ele` of the `Element` class: `pq.push(ele);`
- To extract the topmost/minimum element:

```
Element ele = heap.top();
heap.pop();
```

Java

- To create: `PriorityQueue<Element> pq = new PriorityQueue<Element>(new ElementComparator());`
- To get the size: `pq.size();`
- To insert an object `ele` of the `Element` class: `pq.add(ele);`
- To extract the topmost/minimum element: `Element ele = heap.remove();`

For usages, check `HeapSort.cpp` or `HeapSort.java` files that sorts an array using a priority queue. The code has been lightly commented to aid you. Notice how we create an object of the `Element` class and then insert the object into the priority queue.

3 Merge k-sorted Arrays

We are going to revisit the problem of merging k sorted arrays into a single sorted array. Recall that you used recursion to solve this problem in $O(N \log k)$ time, where N is the total length of all arrays and k is the number of arrays. However, it is a good idea in general to avoid recursion whenever possible. To this end, we will achieve the same complexity by using a priority queue.

Here the input *lists* is a dynamic array of integer dynamic arrays, which essentially represents a jagged array. Here, you will be required to to get the size of *lists*, the size of a particular row of *lists*, and the value at a particular row and column of *lists*. You will merge all the values and return another dynamic array containing these values. Refer to previous assignments on how to use dynamic arrays (both one-dimensional and jagged).

Since each array is sorted, note that the smallest is definitely one of the elements in cell 0 of one of the arrays, say this array is A. The next smallest is either element at cell 1 of array A or one of the elements in cell 0 of one of the arrays other than A. Thus, at each instance we are required to find the smallest from one of the k arrays leaving out the elements that have already been processed. We'll see how a priority queue can be used to this end.

Merge k-sorted Arrays

- Create a priority queue and a dynamic array.
- Insert the first number of each row of *lists* into the priority queue – the item is the row index and the priority is the number itself.
- Create an array *indexes* having the same length as *lists*. Fill the array with 1.
- As long as (the priority queue is not empty), do the following:
  - Extract the minimum element from the priority queue
  - Add the priority of the minimum element to the dynamic array.
  - Let *minItem* be the item of minimum element

- If $indexes[minItem]$ is less than the size of the row at index $minItem$, then
  - insert a new element into the priority queue – for this new element, the item is $minItem$ and priority is the value at row $minItem$ and column $indexes[minItem]$
  - increment $indexes[minItem]$

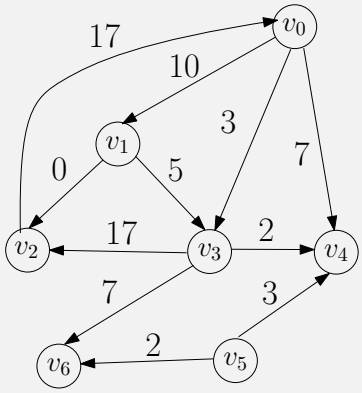
- Return the dynamic array.

4 Dijkstra's Algorithm

Implement the `executeDijkstra` method in `Dijkstra.h/Dijkstra.java`. Also, you have to implement the `readWeightedGraph` method in `Graph.h/Graph.java`. To test the correctness, I have included two sample files (`dijkstra1.txt` and `dijkstra2.txt`). The corresponding graphs are shown next. Each .txt file has the following format. First line is the number of vertices and edges respectively. Second line onwards are the edges in the graph; in particular, each line contains three entries: the source vertex, the destination vertex, and the weight of the edge.

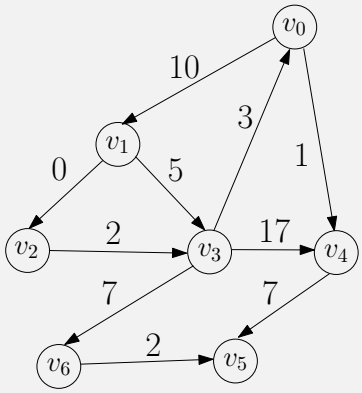
dijkstra1.txt and corresponding graph

```
7
0 1 10
0 3 3
0 4 7
1 2 0
1 3 5
2 0 17
3 2 17
3 4 2
3 6 7
5 4 3
5 6 2
```



dijkstra2.txt and corresponding graph

```
7
0 1 10
0 4 1
1 2 0
1 3 5
2 3 2
3 0 3
3 4 17
3 6 7
4 5 7
6 5 2
```



4.1 Adjacency List: Representing Graphs in Memory

The vertices in the graph are numbered 0 through $n-1$, where n is the number of vertices. We use a two-dimensional jagged array **adjList** (called *adjacency list*) to represent the graph. Specifically, row index i in the array corresponds to the vertex v_i, i.e., row 0 corresponds to v_0, row 1 corresponds to v_1, and so on. Each cell in row i stores an outgoing edge of the vertex v_i. Each edge has 3 properties – *src*, *dest*, and *weight*, which are respectively the vertex from which the edge originates, the vertex where the edge leads to, and the edge weight.

In a nutshell, Edge is a class which has three integer variables – *src*, *dest*, and *weight*. The adjacency list, therefore, is a jagged array, whose type is Edge. In C++, we implement *adjList* as a vector of Edge vectors. In Java, we implement *adjList* as an ArrayList of Edge ArrayLists.

4.2 Pseudo-code

Use the following to complete the `readWeightedGraph` method in `Graph.h/Graph.java` file. You will find similar (in fact, pretty much the same) syntax in previous assignments related to graph.

Reading graph file

- **C++:** Create an input file stream $fileReader$ on $filePath$.
  Java: Create a Scanner object $fileReader$ on $filePath$.
- Read the number of vertices into the class-variable $numVertices$.
- Allocate $numVertices$ rows for $adjList$.
- for ($i = 0$ to $i < numVertices$), do the following:
  - add a blank row to $adjList$
- Read remaining lines of the file one at a time, and do the following:
  - declare three integer variables *src*, *dest*, and *weight*
  - use $fileReader$ to read from the file into these 3 variables respectively
  - create an edge e by calling the Edge constructor with arguments *src*, *dest*, and *weight* respectively
  - add the edge e to the end of $adjList[src]$
- After the loop, close $fileReader$.

You have already seen Dijkstra's algorithm in lecture notes – the core idea is to pick an open vertex with the minimum distance label and relax its outgoing edges and close the vertex. The relaxation step is similar to what we did in acyclic graphs. So, the main questions is how to keep track of open vertices, and how to find an open vertex with minimum distance label.

To address this, we maintain *open* as a priority queue. We will also use a boolean array *closed*[ ] to keep tack of vertices that have been removed from *open*. Therefore, for a vertex v, v is closed if *closed*[v] is *true*. Thus, *closed*[v] is set to *true* when we have found a shortest path to v and it has been removed from *open*, else v is not closed.

You are going to use the built-in priority queues of C++/Java. The only catch is that in C++ or Java, you cannot update priority, i.e., you can only insert and extract. Although that may seem to be a bottleneck, there's any easy workaround.

What we will do is that whenever the distance estimate of a vertex goes down, we simply insert it (as the item) and its estimate (as the priority). Now, whenever a vertex is extracted from *open*, we simply check if it is already closed. If it is already closed, then we simply do not bother about it; otherwise, we close it and relax its outgoing edge.

The only downside is that priority queue size may be as big as the number of edges in the graph. We'll ignore this; the workaround is to implement the PriorityQueue from notes; it's similar to a heap, and I'll leave it as an exercise for the enthusiastic student.

Dijkstra's algorithm

- Set all cells of $distance[\,]$ array to ∞, all cells of $parent[\,]$ array to -1, and all cells of $closed[\,]$ array to $false$
- Set $distance[source]$ to 0
- Create a priority queue *open*
- Insert *source* (as item) into *open* with priority 0
- As long as (*open* is not empty), do the following:
  - Let *minElement* be the element obtained by extracting *open*
  - Let $minVertex$ be the item in *minElement*
  - If $minVertex$ is closed then this vertex has already been relaxed. So, continue and skip the remainder of the code in this loop.
  - Close $minVertex$
  - for (each outgoing edge $adjEdge$ of $minVertex$), do the following: [a]
    - Let $adjVertex$ be the destination of $adjEdge$
    - If ($adjVertex$ is not closed), do the following:
      - Let $newDist$ be $distance[minVertex] + adjEdge's\ weight$
      - Use Dijkstra's edge relaxation rule (i.e., when $newDist < distance[adjVertex]$) to update $distance[adjVertex]$, $parent[adjVertex]$, and insert $adjVertex$ (as item) into *open* with priority $newDist$.

[a] Refer to previous assignments on how to obtain the outgoing edges of a vertex. The only difference here is that you do not have the $outDegree[\,]$ array, but you can simulate that with $adjList$. Specifically, for a vertex v, $outDegree[v]$ is the number of outgoing edges of v, which is given by the length of row v of $adjList$.

Let V and E be the number of vertices and edges in the graph respectively. Whether or not a vertex is closed is detected in $O(1)$ time. Note that each vertex is added to *open* at most as many times as the number of incoming edges. Thus, the size of open is the sum of number of incoming edges over all the vertices, which is simply the same as the number of edges in the graph. For any graph, $E \leq V(V-1) < V^2$; thus, any operation on the priority queue costs $O(\log E) = O(\log V)$ time. Since each edge is relaxed at most once, the total number of insertions to *open* or updates in *dist*/*parent* arrays is at most E. Hence, the complexity is $O(E \log V)$.

Caution: Your code must use a priority queue and it must attain the complexity mentioned here to get any credits.