Programming Assignment 7 (Trie, Dynamic Heap for string data with applications, and Dijkstra's algorithm)

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

1 Overview

We are essentially going to:

- **Implement Insertion and Search in a Trie**
- **Implement Dijkstra's Single Source Shortest Path algorithm**.
- **Implement Insertion and Deletion in a Dynamic Heap containing string data**
- **Implement Heap Sort** and **Find the k lexicographically largest strings** in an array

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

- `insert` and `match` methods in `Trie.h`/`Trie.java`
- `readWeightedGraph` method in `Graph.h`/`Graph.java`
- `executeDijkstra` method in `Dijkstra.h`/`Dijkstra.java`
- `insert` and `deleteMinimum` methods in `Heap.h`/`Heap.java`
- `heapSort` and `topK` methods in `HeapApplications.h`/`HeapApplications.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 Operations on Strings

Through out this assignment, you are going to be dealing with strings. You have already dealt with strings in PA6, but over there, you read characters and ordered characters based on string order. Here, you are going to be primarily do two things:

- Get hold of the character at index i of a string.

  If the string is str, use $str.at(i)$ in C++ and $str.charAt(i)$ in Java to get the character.

- Compare two strings $str1$ and $str2$ to get their relative lexicographic (dictionary) rank.

  Use $str1.compare(str2)$ in C++ and $str1.compareTo(str2)$ in Java to get their relative order.

  If the function returns a negative value, then $str1$ is smaller than $str2$. If the function returns a positive value, then $str1$ is larger than $str2$. If the function returns 0, then $str1$ equals $str2$.

3 Trie

The TrieNode class depicts each node in the trie, where each node is equipped with a hashtable. Say you are at a node u and v is a child of u, such that the edge label from u to v is the character C. Then the hashtable at u will hash C to the node v. TrieNode contains the following two functions. The `insertChild` function inserts a new child TrieNode with the edge label from this node to the new child node being c; it then returns the newly created child node. The `getChild` function returns the child node such that the edge label from this node to the child node is labeled by c; if there is no such child, then it will return null.

Implement the following functions in the Trie class: `insert` to insert a new string into the trie, and `match`, which returns true if the string has been completely matched, else returns false.

Inserting a String

- Set a TrieNode temporary variable *tmp* to the *root*. (In C++, *tmp* has to be a pointer.)
- for ($i = 0$ to $i < str.length()$), do the following:
  - Let c be the character at position i of str.
  - Use the *getChild* method on *tmp* with c as argument to get the child of *tmp*, such that the corresponding edge is labeled by c.
  - If the child is not null, set *tmp* to *child*, i.e., just move to the child.
  - Else, as long as $i < str.length()$, do the following:
    - Use the *insertChild* method on *tmp* with the character at position i of *str* as argument (to insert a new child of *tmp*).
    - The *insertChild* method returns the child; set *tmp* to the child.
    - Increment i by 1.

Matching a String

- Set a TrieNode temporary variable *tmp* to the *root*. (In C++, *tmp* has to be a pointer.)
- for ($i = 0$ to $i < str.length()$), do the following:
  - Let c be the character at position i of str.
  - Use the *getChild* method on *tmp* with c as argument to get the child of *tmp*, such that the corresponding edge is labeled by c.
  - If the child is null, then a mismatch has occurred; hence, return false.
  - Else, set *tmp* to the child.
- Once the for-loop is done, the entire string has matched. Hence, return *true*.

You are going to use a trie to build a spell checker. The idea is that you build a trie out of all the words in the dictionary by inserting them one-by-one. Now, given a sentence, you tokenize it to get all the words. Then, you match each word in the trie; if you get a mismatch that word does not exist in the dictionary, and hence, is an incorrect spelling.

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`.

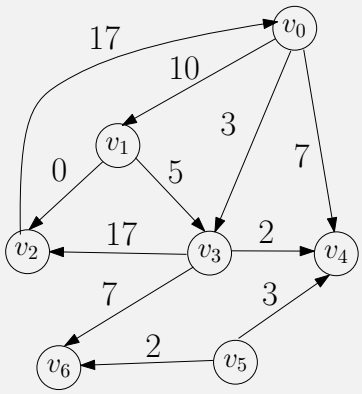
4.1 Data Description

To test the correctness, I have included two sample files (`dijkstra1.txt` and `dijkstra2.txt`). The corresponding graphs are shown in the next page.

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.

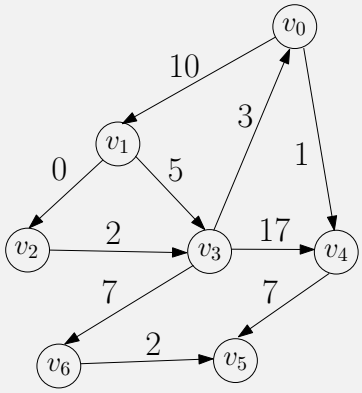
dijkstra1.txt and corresponding graph

```
7 11
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 10
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
```



4.2 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. Additionally, we use an array **outDegree** of length n, where $outDegree[i]$ = the number of outgoing edges of the vertex v_i.

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.

4.3 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 PA 6.

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 $numVertices$. Then, read the number of edges into the class-variable $numEdges$.
- Allocate $numVertices$ cells for $outDegree$. Allocate $numVertices$ rows for $adjList$.
- for ($i = 0$ to $i < numVertices$), do the following:
  - set $outDegree[i]$ to 0
  - add a blank row to $adjList$
- for ($i = 0$ to $i < numEdges$), 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 $adjList[src]$
  - increment $outDegree[src]$ by 1
- 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 are (i) how to keep track of open vertices, and (ii) how to find an open vertex with minimum distance label.

To address (i), we maintain a boolean array $closed[\ ]$, where $closed[v] = true$ indicates that vertex v is closed. Therefore, for a vertex v, if $closed[v]$ is $false$ and $distance[v] \neq \infty$, then we have

found a path to vertex v and v is not closed; hence, v is open. To address (ii), we start by guessing that $minVertex$ is -1 and $minDist = \infty$. Now, for every vertex v, we check if it is not closed and if $distance[v] < minDist$; if the criteria evaluates to true, then the vertex is open and has a smaller estimate than the current estimate; so, we update $minDist = distance[v]$ and $minVertex = v$. At the end, we will find the minimum vertex that needs to be expanded.

To keep track whether or not there is an open vertex, we use an integer variable *numOpen* that stores the number of open vertices. Whenever we relax an open vertex, we decrement *numOpen* (to indicate that a vertex has been closed); whenever we change the distance value of a vertex from ∞ to something finite, we increment *numOpen* (to indicate that a vertex has been opened).

Dijkstra's algorithm

- for ($i = 0$ to $i < numVertices$), do the following:
  - set $distance[i] = \infty$ (INT_MAX in C++ and Integer.MAX_VALUE in Java)
  - set $parent[i]$ to -1
  - set $closed[i]$ to $false$
- Set $distance[source]$ to 0
- Create an integer variable *numOpen* and set it to 1
- As long as ($numOpen > 0$), do the following:
  - Initialize a variable $minDist$ to ∞ and another variable $minVertex$ to -1. These will be used to expand the open vertex with the minimum distance label.
  - for ($i = 0$ to $i < numVertices$), do the following:
    - if ($closed[i]$ *is* $false$ and $distance[i] < minDist$), then:
      - set $minDist$ to $distance[i]$
      - set $minVertex$ to i
  - set $closed[minVertex]$ to $true$
  - decrement *numOpen* by one
  - for ($i = 0$ to $i < outDegree[minVertex]$), do the following:
    - let $adjEdge$ be i^{th} adjacent edge of $minVertex$
    - let $adjVertex$ be the destination of $adjEdge$
    - if ($adjVertex$ is not closed), do the following:
      - set $newDist$ to $distance[minVertex] + adjEdge's\ weight$
      - if $distance[adjVertex]$ equals ∞, then the vertex has not been visited; so, increment *numOpen* by 1
      - if ($newDist < distance[adjVertex]$), then {
        set $distance[adjVertex]$ to $newDist$
        set $parent[adjVertex]$ to $minVertex$
        }

5 Dynamic Heap

Use the notes posted on Heap to implement the `insert` and `deleteMinimum` methods in the `Heap.h`/ `Heap.java` files. Instead of implementing a standard heap (with a given maximum capacity), we shall use a dynamic array to implement a (dynamic) heap, i.e., we do not make any assumptions on the size of the heap. Therefore, at the class level, you have a *heapArray* variable, which is a dynamic array (vector in C++ and ArrayList in Java). Also, instead of integer heaps, we will implement a string heap, i.e., the heap will store strings.

To this end, you will need to make the following changes to the posted notes:

Accessing, Inserting, Deleting, and Swapping heap contents

- You can obtain the current size of the heap using the $size()$ method. The last value in the heap is at index $size() - 1$.
- To read a value of heapArray at index i use the following syntax:

  C++: heapArray.at(i)

  Java: heapArray.get(i)
- To insert a new value v use the following syntax:

  C++: heapArray.push_back(v)

  Java: heapArray.add(v)
- To delete the last value in the heap, use the following syntax:

  C++: heapArray.pop_back()

  Java: heapArray.remove(size() - 1)
- To update the value at index i to the value v, use the following syntax:

  C++: heapArray.at(i) = v

  Java: heapArray.set(i, v)
- To swap the contents of the heap at two indexes x and y, call the swap method with arguments x and y, i.e., `swap`(x, y)

Comparing heap items for swapping

Finally, remember that we are going to store strings in the heap; hence, you cannot compare the values using $<$ or $>$ symbol. Instead, you need to compare them as strings as discussed in the beginning of this draft.
More specifically, say we are comparing $leftKey$ and $rightKey$. To check if $leftKey$ is lexicographically smaller than $rightKey$, use the following syntax:

- **C++:** if $(leftKey.compare(rightKey) < 0)$
- **Java:** if $(leftKey.compareTo(rightKey) < 0)$

Use the same ideas to compare other values in the heap.

6 Heap Applications

Sorting Strings. Our first application of a heap is to sort an array of strings, i.e., order them lexicographically (dictionary order). Normally you could sort strings using a selection sort kind of approach. However, that would be bad (just like sorting numbers using them is bad). To see why that is the case, let's consider the scenario where we have m strings, each of length N characters. Suppose we want to sort them using selection sort. We know that will have a nested for-loop, one going over each string and other to find the minimum string. However, we will have another loop (either explicitly if you write it or implicit if you use the string compare method discussed before); this loop will run N times to find the order between two strings.[1] Therefore, the complexity is ultimately $O(m^2N)$; it would be the same in case of insertion sort.

Using a heap, we can reduce this complexity down to $O(mN \log m)$. Let's see why. You will have at most m nodes in the heap (one for each string), and to compare a string in a node with its parent or children, you will need $O(N)$ time. Therefore, *insert* and $deleteMin$ operations costs you $O(N \log m)$, whereas $getMinimum$ is still an $O(1)$ time operation. To sort, you will carry out m of these operations each. Hence, the overall cost is $O(mN \log m)$, which is a substantial improvement.

Following is the pseudo-code. Use it to complete the `heapSort` method of the `HeapApplications.h`/ `HeapApplications.java` file.

Heap Sort

- Create a heap via a constructor call to the `Heap` class.
- Insert all the strings of the array into the heap using a for loop.
- for ($i = 0$ to $i < arrayLen$), do the following:
  - set $array[i]$ to the smallest string in the heap (use $getMinimum()$ function of the heap to get the minimum string)
  - delete the minimum from the heap

Finding k-lexicographically largest strings. In many scenarios, we are interested to find the top-k numbers (i.e., the k highest numbers) from a given set of numbers. This is has numerous applications related to web searching (such as the 10 most relevant websites for a Google search, or the 20 most popular items for an Amazon search). The obvious way to do this would be to first sort the array, and then report the last k numbers in the sorted array. This, however, has a complexity of $O(n \log n)$, where the length of the array is n. Typically, k is much smaller than n; hence, we want to design an algorithm that is faster for smaller values of k (such as when k is a constant), but no slower than sorting for larger values of k (such as when k approaches n).

Specifically, our goal is to *design an algorithm that can find the k lexicographically largest strings in an array of m strings.* Our algorithm will achieve a complexity of $O(mN \log k)$ time, when N is the maximum length of a string (as opposed to $O(mN \log m)$ via normal heap sort).

The main idea is as follows. Note that the we need the k lexicographically largest strings, and the complexity has a $\log k$ factor; so, immediately, you can guess that the size of the heap should not exceed k. So, what you do is insert the first k strings in the array into the heap. Now, think

[1] Recall how you compare strings lexicographically. You match the strings one character at a time, until you find a mismatch (or exhaust one of the strings). The lexicographically smaller string is the one with the smaller mismatched character (or the shorter one if you exhaust one string).

of the next string in the array; if it is larger than the smallest in the heap then the smallest in the heap cannot be one of the k largest strings, whereas if the next string is smaller than the smallest in the heap, then the next string can never one of the k largest strings. Therefore, either way we can discard one of them – in the first case, remove the minimum and insert the next one, whereas in the second case, ignore the next one. Keep doing this for the remaining strings in the array and the current heap. At the end, the strings left in the heap form the k largest.

Use the following pseudo-code and complete the `topK` method of the `HeapApplications.h`/ `HeapApplications.java` file, which finds the k-largest strings and returns them in an array. Note that the output array must be correct, and you must achieve the claimed complexity for full credit.

topK

- If ($k > arrayLen$) then set $k = arrayLen$
- Create a heap via a constructor call to the `Heap` class.
- Insert the first k elements of the array into the heap using a loop.
- for ($i = k$ to $i < arrayLen$), do the following:
  - let $minString$ be the smallest string in the heap
  - if ($minString$ is smaller than $array[i]$), then:
    - delete the minimum from the heap
    - insert $array[i]$ into the heap
- Create a string array $topK[\,]$ of size k

  C++ programmers must use dynamic allocation. So, if you want to return a string array x of length 10, it must be declared as **string *x = new string[10];**
- Initialize an integer $pos = 0$
- while (size of heap > 0), do the following:
  - set $topK[pos]$ = the minimum string in the heap
  - increment pos by one
  - delete the minimum from the heap
- return $topK$