The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 280– Intermediate Data Structures and Algorithms

Assignment 8

Date Due: April 2, 2020, 9:00pm

Total Marks: 61

1 Submission Instructions

- Assignments must be submitted using Moodle.
- Responses to written (non-programming) questions must be submitted in a PDF file, plain text file (.txt), Rich Text file (.rtf), or MS Word's .doc or .docx files. Digital images of handwritten pages are also acceptable, provided that they are **clearly** legible.
- Programs must be written in Java.
- If you are using IntelliJ (or similar development environment), do not submit the Module (project). Hand in only those files identified in Section 5. Export your .java source files from the workspace and submit only the .java files. **Compressed archives are not acceptable.**
- No late assignments will be accepted. See the course syllabus for the full late assignment policy for this class.

2 Background

In this section we present material required for Question 1.

2.1 Union-find ADT

A *union-find* ADT (also called a *disjoint-set* ADT) keeps track of a set of elements which are partitioned into disjoint subsets. It is useful for establishing equivalencies of groups of items in a set about which nothing is known initially. For example, suppose we have an initial set of cities:

Vancouver, Edmonton, Regina, Saskatoon, Winnipeg, Toronto, Montreal, Calgary

Let's then suppose that we decide that Vancouver and Edmonton are "equivalent" (this can be defined in any number of ways), that Regina, Saskatoon, and Winnipeg are equivalent, and that Montreal and Calgary are equivalent. Now we would have four subsets of equivalent elements of our overall set:

$$\{\text{Vancouver}, \text{Edmonton}\}, \{\text{Regina}, \text{Saskatoon}, \text{Winnipeg}\}, \{\text{Toronto}\}, \{\text{Montreal}, \text{Calgary}\}$$

Note that since Toronto was not deemed equivalent to anything, it is in its own subset by itself. Now, let's suppose we want to find out which set a particular city is in. This is done by choosing from each subset a *representative* (also called an *equivalence-class label*) which acts as the identifier for that set. Suppose for the sake of simplicity, that we choose the first item in each set as its representative (shown in bold):

$$\{\textbf{Vancouver}, \text{Edmonton}\}, \{\textbf{Regina}, \text{Saskatoon}, \text{Winnipeg}\}, \{\textbf{Toronto}\}, \{\textbf{Montreal}, \text{Calgary}\}$$

If we were to now ask which subset Winnipeg belongs to, the answer would be Regina. Asking which subset an element belongs to is called the *find* operation. The find operation applied to an element returns the representative of the set to which it belongs, for example, find(Winnipeg) = Regina, or find(Calgary) = Montreal, or find(Vancouver) = Vancouver. The find operation is one of the two main operations supported by the Union-Find ADT.

The Union-Find ADT unsurprisingly supports a second operation called *union*. The union operation takes two elements as arguments, and establishes them as being "equivalent", meaning, they should be in the same set. So union(Edmonton, Calgary) would place Calgary and Edmonton in the same subset. But if Edmonton and Calgary are equivalent, then by transitivity, everything in the subsets to which Edmonton and Calgary belong must also be equivalent, so the union operation actually merges two subsets into one — so this is just familiar set union operation!. Thus, union(Edmonton, Calgary) would alter our group of subsets so they look like this:

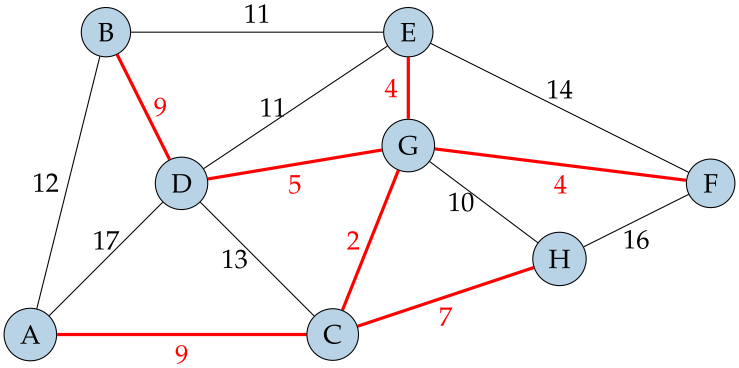
$$\{\textbf{Vancouver}, \text{Edmonton}, \text{Montreal}, \text{Calgary}\}, \{\textbf{Regina}, \text{Saskatoon}, \text{Winnipeg}\}, \{\textbf{Toronto}\}$$

So now Find(Calgary) would result in an answer of Vancouver. You may be wondering why we chose Vancouver as the representative element of the merged subset instead of Montreal. This is an implementation-level decision. In principle, either one could be chosen.

In summary, the Union-Find data structure keeps track of a set of disjoint subsets of a set of elements. It supports the operations find(X) (look up the name of the subset to which element X belongs) and union(X,Y) (merge the subsets containing X and Y). In this assignment we will implement the union-find ADT using a directed, unweighted graph.

2.2 Minimum Spanning Tree

Given a connected, weighted, undirected graph, its minimum spanning tree consists of the subset of the graph's edges of smallest total weight such that the graph remains connected. Such a set of edges always forms a tree because if it weren't a tree there would be a cycle, which implies that it wouldn't be the minimum cost set of edges that keeps the graph connected because you could remove one edge from the cycle and the graph would still be connected. Here is a weighed, undirected graph, and its minimum spanning tree (denoted by thicker, red edges):



No other set of edges that keeps the above graph connected has a smaller sun of weights.

The minimum spanning tree has many applications since many optimization problems can be reduced to a minimum spanning tree algorithm. Suppose you have identified several sites at which to build network routers and you know what it would cost to connect each pair of network routers by a physical wire. You would like to know what is the cheapest possible way to connect all your routers. This is an instance of the minimum spanning tree problem.

Finding the minimum spanning tree isn't as straightforward as it might seem. There are various algorithms for finding the minimum spanning tree. We will be using Kruskal's algorithm which, conveniently, can be implemented efficiently with a union-find ADT.

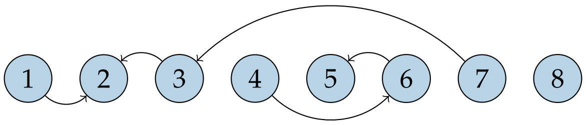
3 Your Tasks

Question 1 (16 points):

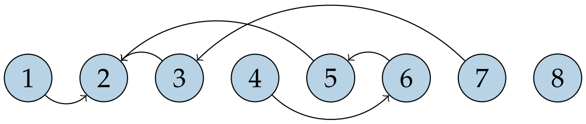
For this problem you will implement Kruskal's algorithm for finding the minimum spanning tree of an undirected weighted graph. Kruskal's algorithm uses a union-find data structure to keep track of subsets of vertices of the input graph G. Initially, every vertex of G is in a subset by itself. The intuition for Kruskal's algorithm is that the edges of the input graph G are sorted in ascending order of weight (smallest weights first), then each such edge (a, b) is examined in order, and if a and b are currently in different subsets we merge the two sets containing a and b and add (a, b) to the graph of the minimum spanning tree. This works because vertices in the same subset in the union-find structure are all connected. Once all of the vertices are in the same subset, we know that they are all connected. Since we always add the next smallest edge possible to the minimum spanning tree, the result is the smallest-cost set of edges that cause the graph to be completely connected, i.e. the minimum spanning tree! Here's Kruskal's algorithm, in pseudocode:

```
Algorithm minimumSpanningTreeKruskal(G)
G - A weighted, undirected graph.

minST = an undirected, weighted graph with the same node set as G,
        but no edges.

UF = a union-find data structure containing the node set of G in which
     each node is initially in its own subset.

Sort the edges of G in order from smallest to largest weight.

for each edge e=(a,b) in sorted order
    if UF.find(a) != UF.find(b)
        minST.addEdge(a,b)
        set the weight of (a,b) in minST to the weight of (a,b) in G
        UF.union(a,b)

return minST
```

In order to implement Kruskal's algorithm you will first need to implement a union-find ADT. We can implement union-find with a directed (unweighted) graph F. Initially the graph has a node for each item in the set, and no edges. This makes the union operation very easy. The operation union(a,b) can be completed simply by adding the edge $(\text{find}(a), \text{find}(b))$ to F, that is, we add an edge that connects the representative elements of the subsets containing a and b. The find(a) operation then works by checking node a to see if it has an outgoing edge, if it does, we follow it and check the node we get to to see if it has an outgoing edge. We continue going in this fashion until we find a node that does not have an outgoing edge. That node is the representative element of the subset that contains a, and we would return that node. Here's an example of a directed graph that represents a set of subsets of the elements 1 through 8:



If we were to call find(7) on this graph, we would see that 7 has an edge to 3, which has an edge to 2, but 2 has no outgoing edge, so find(7) = 2. Similarly if we called find(4), we would follow the edge to node 6, then its outgoing edge to node 5, and find that 5 has no outgoing edge, so find(4) = 5. Overall, this graph represents that 1, 2, 3, and 7 are in the same subset, which has 2 as its representative element; that 4, 5, and 6 are in the same subset with representative element 5, and 8 is in a subset by itself. Now, suppose we do union(6, 1). This causes an edge to be added from find(6)=5 to find(1)=2, that is an edge from 5 to 2:



This causes the subsets containing 6 and 1 to be merged, and the new merged subset has representative element 2. Convince yourself that if you call find() on any element except 8, you will get a result of 2 – follow the arrows from the starting node and you'll always end up at 2.

Here are the algorithms for the union and find operations using a graph as the underlying data structure:

```
Algorithm union(a, b)
a, b - elements whose subsets are to be merged

// If a and b are already in the same set, do nothing.
if find(a) == find(b)
   return

// Otherwise, merge the sets
add the edge (find(a), find(b)) to the union-find graph.



Algorithm find(a)
a - element for which we want to determine set membership

// Follow the chain of directed edges starting from a
x = a
while x has an outgoing edge (x,y) in the union-find graph
   x = y

// Since at this point x has no outgoing edge, it must be the
// representative element of the set to which a belongs, so...
return x
```

These are the simplest possible algorithms for union() and find(), and they don't result in the most efficient implementations. There are improvements that we could make, but to keep things simple, we won't bother with them. Eventually, I'll provide solutions that use these algorithms, as well as an improved, more efficient solution for those who are interested.

Well, that was a lot of stuff. Now we can finally get to what you actually have to do:

1. Import the project `Kruskal-Template` (provided) module into IntelliJ workspace. You may need to add the `lib280-asn8` project (also provided) as a module depdnency of the `Kruskal-Template` module (this process is covered in the self-guided tutorials on Moodle).
2. In the `UnionFind280` class in the `Kruskal-Template` project, complete the implementation of the methods `union()` and `find()`. Do not modify anything else. You may add a `main` method to the `UnionFind` class for testing purposes.

3. In Kruskal.java complete the implementation of the minSpanningTree method. Do not modify anything else.
4. Run the main program in Kruskal.java. The pre-programmed input graph is the same as the one shown in Section 2.2. The input graph and the minimum spanning tree as computed by the minSpanningTree() method are displayed as output. Check the output to see if the minimum spanning tree that is output matches the one in Section 2.2.

Implementation Hints

When implementing Kruskal's algorithm, you should be able to avoid having to write your own sorting algorithm, or putting the edges into an array to sort the edges by their weights. You can take advantage of ADTs already in lib280-asn8a. All you need is to put the edges in a dispenser which, when you remove an item, will always give you the edge with the smallest weight (hint: look in the lib280.tree package for ArrayedMinHeap280). Conveniently, WeightedEdge280 objects are Comparable based on their weight.

Question 2 (20 points):

For this question you will implement Dijkstra's algorithm. The implementation will be done within the `NonNegativeWeightedGraphAdjListRep280` class which you can find in the `lib280-asn8.graph` package. This class is an extension of `WeightedGraphAdjListRep280` which restricts the graph edges to have nonnegative weights. This works well for us since Dijkstra's algorithm can only be used on graphs with nonnegative weights.

1. Implement the `shortestPathDijkstra` method in `NonNegativeWeightedGraphAdjListRep280`. The method's javadoc comment explains the inputs and outputs of the method.
2. Implement the `extractPath` method in `NonNegativeWeightedGraphAdjListRep280`. The method's javadoc comment explains the inputs and outputs of the method.

The pseudocode for Dijkstra's algorithm is reproduced below.

```
Algorithm dijkstra(G, s)
G is a weighted graph with non-negative weights.
s is the start vertex.
Postcondition: v.tentativeDistance is the length of the
               shortest path from s to v.
               v.predecessorNode is the node that appears before v
               on the shortest path from s to v.

Let V be the set of vertices in G.

For each v in V
    v.tentativeDistance = infinity
    v.visited = false
    v.predecessorNode = null

s.tentativeDistance = 0

while there is an unvisited vertex
    cur = the unvisited vertex with the smallest tentative distance.
    cur.visited = true

    // update tentative distances for adjacent vertices if needed
    // note that w(i,j) is the cost of the edge from i to j.
    For each z adjacent to cur
        if (z is unvisited and z.tentativeDistance >
                               cur.tentativeDistance + w(cur,z) )
            z.tentativeDistance = cur.tentativeDistance + w(cur,z)
            z.predecessorNode = cur
```

Implementation Hints

Even though the pseudocode implies that `tentativeDistance`, `visited` and `predecessorNode` are properties of vertices and perhaps should be stored in vertex objects, it is easiest to just use a set of parallel arrays in the implementation of Dijstra's algorithm, much like the way we represented these as arrays during the in-class examples. E.g. an array `boolean visited[]` such that if `visited[i]` is true, it means that vertex `i` has been visited. This is quite easy to use since vertices are always numbered 1 through n.

Sample Output

If you have done things right, then you should get the following outputs for start vertices 1 and 9 respectively.

```
Enter the number of the start vertex:
1
The length of the shortest path from vertex 1 to vertex 1 is: 0.0
Not reachable.
The length of the shortest path from vertex 1 to vertex 2 is: 1.0
The path to 2 is: 1, 2
The length of the shortest path from vertex 1 to vertex 3 is: 3.0
The path to 3 is: 1, 3
The length of the shortest path from vertex 1 to vertex 4 is: 23.0
The path to 4 is: 1, 3, 5, 6, 4
The length of the shortest path from vertex 1 to vertex 5 is: 7.0
The path to 5 is: 1, 3, 5
The length of the shortest path from vertex 1 to vertex 6 is: 16.0
The path to 6 is: 1, 3, 5, 6
The length of the shortest path from vertex 1 to vertex 7 is: 42.0
The path to 7 is: 1, 3, 5, 6, 4, 8, 9, 7
The length of the shortest path from vertex 1 to vertex 8 is: 31.0
The path to 8 is: 1, 3, 5, 6, 4, 8
The length of the shortest path from vertex 1 to vertex 9 is: 36.0
The path to 9 is: 1, 3, 5, 6, 4, 8, 9


Enter the number of the start vertex:
9
The length of the shortest path from vertex 9 to vertex 1 is: 36.0
The path to 1 is: 9, 8, 4, 6, 5, 3, 1
The length of the shortest path from vertex 9 to vertex 2 is: 35.0
The path to 2 is: 9, 8, 4, 6, 5, 3, 2
The length of the shortest path from vertex 9 to vertex 3 is: 33.0
The path to 3 is: 9, 8, 4, 6, 5, 3
The length of the shortest path from vertex 9 to vertex 4 is: 13.0
The path to 4 is: 9, 8, 4
The length of the shortest path from vertex 9 to vertex 5 is: 29.0
The path to 5 is: 9, 8, 4, 6, 5
The length of the shortest path from vertex 9 to vertex 6 is: 20.0
The path to 6 is: 9, 8, 4, 6
The length of the shortest path from vertex 9 to vertex 7 is: 6.0
The path to 7 is: 9, 7
The length of the shortest path from vertex 9 to vertex 8 is: 5.0
The path to 8 is: 9, 8
The length of the shortest path from vertex 9 to vertex 9 is: 0.0
Not reachable.
```

Question 3 (25 points):

For this problem you will write a method (or methods) to sort an array of strings using the MSD Radix Sort. For purposes of this assignment, you may assume that strings contain only the uppercase letters A through Z.

You have been provided with an IntelliJ module `RadixSortMSD-Template` which includes a short main program that will load a data file containing strings to be sorted. There are several files provided named `words-XXXXXX.txt` where "`XXXXXX`" denotes the number of words in the file. The file format starts with the number of words in the file, followed by one word per line. There is also a file `words-basictest.txt` which is a good set of words to use to determine whether your sort is running correctly.

The pseudocode for MSD Radix Sort from the notes is duplicated on the next page for your convenience. **Note that we are removing the optimization of sorting short lists with insertion sort, as indicated by the strikethrough text.** You may just always recursively radix sort any list with more than one element on it.[1]

Complete the following tasks:

1. Write your sort method(s) within the `RadixSortMSD` class. It should accept an array of strings as input, and return nothing. When the method returns, the array that was passed in should be in lexicographic order (i.e. dictionary order).
2. Call your sort at the spot indicated in the main() function.
3. Record in a file called `a8q3.txt/doc/pdf` the time in milliseconds it takes to sort 50, 100, 500, 1000, 10000, 50000, and 235884 items (there are input files with each of these numbers of words provided). Include this file in your assignment submission.
4. When you hand in `RadixSortMSD.java`, leave the input file set at `words-basictest.txt` so that it is easy for the markers to run your program on this input file to see that it works.

Assessment

There will be marks allotted to the following aspects of your solution:

Correctness. As always, the solution has to work!

Design and speed of your implementation. Design and speed will be considered together because they influence each other — a poor design choice may result in a slower runtime. Design-wise, any reasonable design will be accepted; marks will be deducted for especially poor choices. Speed-wise, the bar to get full marks here will be fairly low, but if your sort is egregiously slow we will deduct some points.

Javadoc and inline commenting. As usual, include a javadoc comment with each header, and document meaningful blocks of code with inline comments to enhance understanding of the code.

For full details, consult the grading rubric on Moodle.

[1] You can, however, use the insertion-sort optimization if you want to, but you will probably have to implement your own insertion sort. If you choose to attempt this optional optimization, you are permitted to use resources from the internet to implement insertion sort, but **only** for the insertion sort, and **only** if you properly attribute any code you use to its original author or website.

Implementation Hints

- One of the most important decisions you have to make in your implementation is the choice of data structure to represent the array of lists used in the sortByDigit() helper method. Choose carefully! Your choice **will** have an impact on the speed of your sort, and the ease of implementing it. When considering which data structure to use, you may select from containers in either lib280 or the standard Java API. Case in point: on my first attempt, I made a bad decision that caused the sort of 10000 words to take several minutes. Now it takes 24ms (on my computer).
- Although runtimes will vary from machine to machine, on a decent machine you should be able to sort even the 235884-word input file in less than one second (1000ms). Even on slower machines if it is taking more than a few seconds, then you've done something particularly inefficient.
- Don't take the pseudocode below too literally. This is very high-level pseudocode which is intended to describe the operation of the algorithm, but intentionally glosses over a lot of details that become important at implementation-time (that's what pseudocode is **for**!). You need to fill in those details as you go. Don't be afraid to do what you need to do to get the job done, but that said, you should not need to write hundreds of lines of code (if you are, you should seek help and advice from Mark or a TA).

```
Algorithm MsdRadixSort(keys, R)
keys - keys to be sorted
R - the radix

sortByDigit(keys, R, 0)


Algorithm sortByDigit(keys, R, i)
keys - keys to be sorted
R - the radix
i - digit on which to partition -- i = 0 is the left-most digit

    for k = 0 to R-1
        list[k] = new list  // Make a new list for each digit

    for each key
        if the i-th digit of the key has value k add the key to list k

    for k = 0 to R-1
        if there is another digit to consider
            ~~if list[k] is small~~
                ~~use an insertion sort to sort the items in list[k]~~
            ~~else~~
                sortByDigit(list[k], i+1)

    keys = new list // empty the input list

    For k = 0 to R-1
        keys = keys append list[k]
```

4 Files Provided

lib280-asn8: A copy of lib280 which includes:

- solutions to assignment 7;
- graph classes necessary for questions 1 and 2.

GraphAdjListRep280 and WeightedGraphAdjListRep280 which you'll use in Question 1

Kruskal-template An IntelliJ module with templates template for question 1.

NonNegativeWeightedGraphAdjListRep280 class for Question 2.

RadixSortMSD-Template: The project template for question 3.

5 What to Hand In

UnionFind280.java Your completed union-find class from Question 1

Kruskal.java Your completed implementation of Kruskal's algorithm from Question 1.

NonNegativeWeightedGraphAdjListRep280.java Your completed implementation of Dijkstra's algorithm from Question 2.

RadixSortMSD.java: Your completed radix sort from question 3.

a8q3.txt/doc/pdf: Your timing observations from your sort in question 3.