The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 280– Intermediate Data Structures and Algorithms

Assignment 7

Date Due: March 21, 2017, 10:00pm

Total Marks: 56

1 Submission Instructions

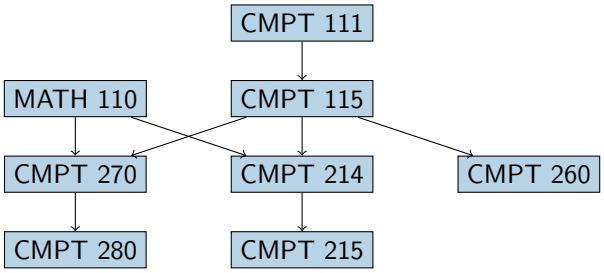
- Assignments must be submitted using Moodle.
- Responses to written (non-programming) questions must be submitted in a PDF file, plain text file (`.txt`), Rich Text file (`.rtf`), or MS Word's `.doc` or `.docx` files. Digital images of handwritten pages are also acceptable, provided that they are **clearly** 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

This assignment is meant to be completed over two weeks. Start early! You may run out of time if you leave everything until the second week.

2.1 Topological Sort of a Directed Graph

Imagine a directed graph is used to represent prerequisite relationships between university courses. There is a node in the graph for each course, and a directed edge from course A to course B if course A is a prerequisite for course B. Here's what such a graph would look like for some of our courses might look like:



Note that this is a graph and not a tree, because CMPT 270 has more than one "parent". But, there are no cycles in this graph.

If there are no cycles in a directed graph, we can perform an operation called a *topological sort*. A topological sort of a graph produces a linear ordering of the nodes such that if there is a directed edge from node A to node B, then node A appears before node B in the ordering. In the specific case of the course prerequisite graph, a topological sort produces an ordering of the nodes such that no course appears before any of its prerequisites – precisely the thing one needs to know in order to take courses in the right order!

Note that there is not necessarily a unique answer for topological sort. For example, in the graph pictured above, we could take MATH 110 and CMPT 111 in either order (because neither has prerequisites), so both of the following sequences would be valid topological sorts:

- 111, 110, 115, 270, 214, 260, 280, 215
- 110, 111, 115, 270, 214, 260, 280, 215

Also note that 280 and 215 could be taken in either order as long as they are taken after both 270 and 214, resulting in more possible topological orderings. The basic algorithm for topological sort, and a variation that is relevant to the specific problem we will want to solve, is presented with Question 1.

Finally, it is worth noting that you cannot perform a topological sort on a graph that has a cycle. If there is a cycle, then it is not possible to satisfy the prerequisites for any node in the cycle because each course in the cycle is a prerequisite of itself.

2.2 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 list 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:

{Vancouver, Edmonton}, {Regina, Saskatoon, Winnipeg}, {Toronto}, {Montreal, 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):

{**Vancouver**, Edmonton}, {**Regina**, Saskatoon, Winnipeg}, {**Toronto**}, {**Montreal**, 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.

It should not surprise you that the other operation is 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. Thus, union(Edmonton, Calgary) would alter our group of subsets so they look like this:

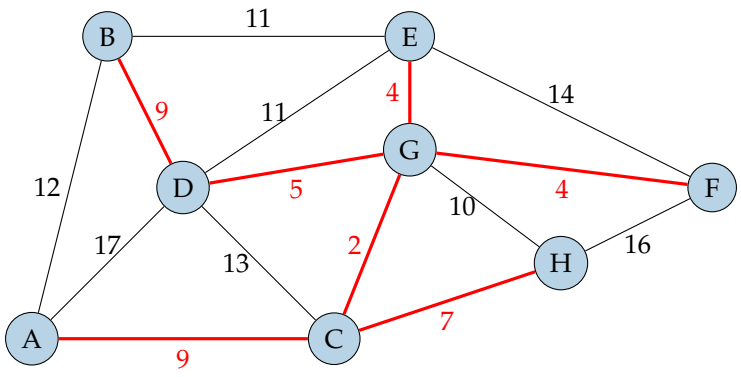
{**Vancouver**, Edmonton, Montreal, Calgary}, {**Regina**, Saskatoon, Winnipeg}, {**Toronto**}

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.

2.3 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 a 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 (15 points):

In this question we will once again be looking at a problem related to quests in video games. In many roleplaying video games, one completes quests to gain *experience points*. When one's charater gains enough experience points, their character advances and grows in power, or gains new abilities. Very often there are quests that can only be attempted after one or more prerequisite quests have been completed. We can represent this as a *quest prerequisite graph*, much like the course prerequisite graph in Section 2.1, in which each quest is represented by a node, and there is a directed edge from node A to node B if node A's quest is a prerequisite to node B's quest. In this question you will work with just such a graph, and implement a topological sort algorithm that will output an ordering in which all of the quests in the graph can be completed.

We can perform the topological sort of the graph using the following algorithm:

```
Algorithm TopologicalSort(G)
G is a directed graph.

L = Empty list that will contain the result of the topological sort
S = set of nodes in G with no incoming edges
while S is non-empty do
    remove a node n from S
    add n to the end of the list L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m now has no incoming edges then
            insert m into S

if the graph has any edges in it then
    throw exception (the graph had at least one cycle!!!)
else
    return L (a topologically sorted order)
```

Keeping in mind that each node of the graph `G` stores information about one quest, we can say that `S` stores the set of quests that are "doable" or "available" (those for which all prerequisites have already been completed) any any partcular moment. Being a set, it stores these quests in no particular order. But... we are greedy. When we remove a node from S at the top of the while loop, we always want to remove the quest in S that has the biggest experience point (XP) reward. In this way, we always do the available quest with the largest experience reward first so as to gain experience more quickly. But to do this we have to change this algorithm a little. Fortunately, it's a very easy change. All we have to do is **change S from a set of graph nodes to a heap of quests that are keyed on the quest's XP value**. Then we are guaranteed to always remove from S the available quest with the highest possible experience point reward (because it will always be at the top of the heap!) [1] The modified algorithm is:

[1] Note: although the modified algorithm will get us XP faster than if we used the original algorithm, it will NOT guarantee that we gain the **most** possible XP with the fewest number of quests. To illustrate this, suppose we have Quests 1, 2, 3, and 4 which are wroth 50, 500, 500, and 5000 XP respectively, but, quest 1 is a prerequisite to quest 4. Initially, the available quests are 1, 2, and 3 (4 cannot be done without doing 1 first). Following the algorithm with the "set" replaced by a heap, we would end up with quests 2 and 3 being first in the topological order, because they are both worth more experience than quest 1. Then we would have to do quest 1, and finally quest 4. If we did the quests in this order, over the first 3 quests we would gain 1050 experience. But we **could** have done quest 1 first, and **then** quest 4 immediately, giving us 5050 experience after just two quests. But we didn't realize this because we didn't look ahead... our algorithm is "greedy" in this respect and we took the **available** quest with the best reward first. This is fine and this is what we want to do here, I just don't want you to expect you will always get the "optimum" experience gain from this algorithm.

```
Algorithm TopologicalSort(G)
G is a directed graph.

L = Empty list that will contain the result of the topological sort
H = heap of quests (compared by experience value) whose corresponding
    nodes in G have no incoming edges,

while H is non-empty do
    remove a quest n from H
    add quest n to the end of the list L
    for each graph node m such that there is an edge e from n's graph node to m do
        remove edge e from the graph
        if m now has no incoming edges then
            insert m's quest into H

if the graph has any edges in it then
    throw exception (the graph had at least one cycle!!!)
else
    return L (a topologically sorted order)
```

You will be provided with the following files as part of an IntelliJ module called `QuestPrerequisites-Template`:

Quest.java A class for storing information about a quest. Note that it is different from the `QuestLogEntry` class from Assignment 4. It contains mostly accessors and mutators for protected instance variables. It also has a `toString()` method and a `compareTo` methods that allow quests to be compared by their experience values. You may not modify this class.

QuestVertex.java A class to be used as the vertex class in a graph. It defines a graph node that can store a reference to a quest. You can use the `quest()` method of this graph node to obtain the quest associated at this node. You may not modify this class.

QuestProgression.java This class contains a few static methods that together form a program for doing a topological sorting of quests in a "quest prerequisite graph". It contains the following static methods:

- `readQuestFile`**:** Reads a data file containing information about quests and quest prerequisites and builds a quest prerequisite graph. This method is finished, and you don't have to do anything to it. Note that the graph that is created has nodes that are of type `QuestVertex`.
- `hasNoIncomingEdges`**:** A method that determines whether a given node in a given graph has no incoming edges.
- `questProgression`**:** A method that performs a topological sort of a given quest prerequisite graph.
- `main`**:** The main program that loads the quest data, constructs the graph, performs the topological sort, and displays the result. This method is finished and you don't have to do anything to it.

quests16.txt A data file containing information on 16 quests and their prerequisites which is used by `main()` as the program input.

You must complete the implementation of the `hasNoIncomingEdges` **and** `questProgression` **methods.** It is up to you to inspect the methods available in the `GraphMatrixRep280` class (and it's superclasses!) and use them to solve the problem. Because you are not implementing methods within the graph class, you can only access and modify the graph via its public interface.

Hints

This program relies on a correspondence between graph nodes and quests. Note that in the graph ADTs, nodes are referred to by integers, starting from 1. Note also that each `Quest` instance contains an ID (you can see this in `Quest.java`). The quest prerequisite graph is constructed by the `readQuestFile` method such that a quest with a particular ID k is always associated with node k of the graph. Thus, if you have a quest object, you can find its corresponding graph node by looking up the node in the graph with the quest's ID. If you have a node ID, you can find its corresponding quest by looking up the node object (of type `QuestVertex`) using its ID, and call the `quest()` method of the resulting vertex object to obtain the quest stored there.

Sample output is shown below.

Sample Output

If you implement the unfinished methods correctly, the output of the program when given quests16.txt as input will be:

```
13, Discover Peppermint Butler's Secrets, Candy Kingdom, XP: 14550
7, Find the Ice King's Wizard Eye, Ice Kingdom, XP: 12000
12, Rescue Marceline from the Nightosphere, The Nightosphere, XP: 25000
10, Win Wizard Battle, Wizard Battle Arena, XP: 29000
9, Rescue Wildberry Princess from the Ice King, Ice Kingdom, XP: 4200
3, Defeat Goliad, Candy Kingdom, XP: 2578
5, Atone for Shoko's Sins, Finn's Treehouse, XP: 2700
4, Locate the Lich's Lair, Costal Wasteland, XP: 1000
8, Get some pickles from Prismo, Prismo's Home, XP: 150
6, Make an Amazing Sandwich, Finn's Treehouse, XP: 1900
16, Watch what Beemo Does When He Is Alone, Finn's Treehouse, XP: 70
1, Steal a Treat from the Donut Witch's Garden, Blue Plains, XP: 250
14, Defeat the Ice King's Penguin Army, Candy Kingdom, XP: 50000
11, Eat Marceline's Fries, Marceline's House, XP: 50
15, Discover the Ice King's Secret Past, The Past, XP: 3299
2, Recover the Stolen Items from the Door Lord, The Sandylands, XP: 700
```

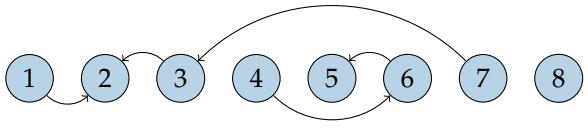
Another test you can do to check whether your program is working is to make a copy of `quests16.txt` and remove all of the ordered pairs starting on line 18 of the file. If you do this, then the topological sort should result in a list of quests sorted in descending order by their experience value.

Question 2 (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 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:

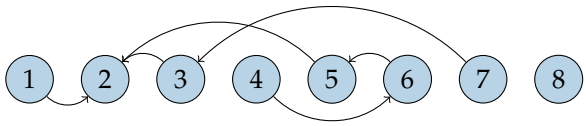
```
Algoirthm minimumSpanningTreeKruskal(G)
G - A weighted, undirected graph.

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

UF = a union-find data structure with 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-asn7` project (also provided) to the Java Build Path of the `Kruskal-Template` module.
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 2.3. 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 2.3.

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-asn7`. 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 3 (25 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-asn7.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.

```
Algoirthm 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 `visisted[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 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.
```

4 Files Provided

lib280-asn7: A copy of lib280 which includes:

- solutions to assignment 6;
- the `GraphMatrixRep280` which you'll use in question 1;
- the `GraphAdjListRep280` and `WeightedGraphAdjListRep280` classes which you'll use in Question 2; and
- the `NonNegativeWeightedGraphAdjListRep280` class for Question 3.

QuestPrerequisites-Template: The project template for question 1.

Kruskal-template: The project template for question 2.

5 What to Hand In

QuestProgression.java Your completed `QuestProgression` class from question 1.

UnionFind280.java Your completed union-find class from Question 2

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

NonNegativeWeightedGraphAdjListRep280.java Your completed implementation of Dijkstra's algorithm from Question 3.