CMSC 132 Project #4 Fall 2012

Date due: Wednesday, November 28, 10:00:00 p.m.

1 Introduction

This project has two parts: creating a representation of a graph, and using it to implement two graph algorithms. One is Dijkstra's algorithm for shortest paths, which we discussed in class, and the other is a variation of depth–first search that tests whether a vertex is part of a graph that contains a cycle. The purpose of the project is to figure out the specific details of representing a graph, which we talked about only in general terms in class, and being able to use it to implement practical algorithms for graphs. You are encouraged to read Chapter 10 in the text carefully, including Section 10.3 that covers graph representations.

Dijkstra's algorithm is one of the more difficult algorithms we have studied, in that even once you understand its steps they're still a bit abstract, and implementing them involves more detail that you have to figure out. It's likely that about half of your project code is going to be just your implementation of Dijkstra's algorithm. Be sure to trace and understand the algorithm before beginning, and start early on the project so there's enough time.

Be sure to carefully read Section 4 below, so you know what you can do, what you can't do, and what you have to do, before starting to code.

2 Graphs in this project

Graphs in this project will be directed. Of course an undirected graph can be simulated using a directed graph by just having two edges, one in each direction, between every pair of vertices that have an edge in the corresponding undirected graph. Edges will also have weights, which will be integers. Note that the algorithm to determine whether a vertex is part of a graph that contains a cycle doesn't use edge weights, and is concerned only with the presence or absence of edges in a graph, while Dijkstra's algorithm does use edge weights.

Some graph operations you have to implement are structural, meaning ones that construct or modify graphs or return information about them, such as adding vertices and edges, or returning the cost of an edge. Besides the structural operations you need to implement the two algorithms mentioned above.

Each vertex of a graph will have some type of associated data; in particular, the graph will be a generic type (which must be instantiated with some type of object, not a primitive type), and its type parameter represents the type of data for each vertex. A graph can be created storing characters (e.g., `Graph<Character>`), or strings (`Graph<String>`), or integers (`Graph<Integer>`), or of hamsters, if a suitable class storing the data of a hamster has been defined (`Graph<Hamster>`). The only requirement is that the class that the type parameter is instantiated with must implement the `Comparable` interface, so it has a natural ordering based on its `compareTo()` method. You may assume that any class that the `Graph` class is instantiated with will be a well–designed Java class, so it will have overridden the `toString()` method and it will follow the hash code contract, and you know that it will implement the `Comparable` interface due to the generic type parameter bound.

You need to choose a representation for graphs, with the restriction that you **cannot use an adjacency matrix** (you also cannot use an incidence matrix, a representation that we didn't cover). You may use one of the other representations we talked about: namely an adjacency list, set, or map, or variants of these. However, you **cannot use Java library collection classes**. You have to create your own data structures, either classes you have written either specially for this project or for earlier projects.

The algorithm below for detecting if a vertex is part of a graph that contains a cycle, and Dijkstra's algorithm, both involve (in different ways) keeping track of which vertices have been processed so far. In covering the graph traversal algorithms we saw in class (breadth–first search and depth–first) two methods of doing this were discussed– either using a visited set, or adding tags to vertices. You may use either of these you like. Notice that vertices in your project are a generic type and any kind of object may be used to instantiate them, so you don't have any ability to modify whatever class the `Graph` class generic type parameter `V` is instantiated with, meaning that you can't add a tag directly to it. However, if you want to use tags, you can create a class of your own that contains a field of type `V`, as well as a tag (or different tags if needed), and use objects of that class as vertices in your graph. Or you could use a visited set instead (if you implement a set data struture).

3 The `Graph` class, and methods you have to write

The methods that you must write for the `Graph` class are described below. Some of the methods should throw exceptions in certain situations; it doesn't matter what string or message these exceptions are constructed with, or even if they have a message string at all. In any situation where more than one error condition is applicable, so more than one exception could be thrown, it doesn't matter which one your code throws. In cases where exceptions are thrown the current object graph should not be modified by the method before throwing the exception.

3.1 `Graph()`

The constructor should create and return a new `Graph` object. Exactly what it does depends of course on how you implement graphs. Note that it should be possible for a program to create and manipulate multiple `Graph` objects simultaneously, and their data should not conflict.

3.2 `void addVertex(V vertex) throws IllegalArgumentException`

This method should add a new vertex with data `vertex` to its current object graph. Exactly what it does depends on how you implement graphs.

If there is already a vertex with data `vertex` present in the graph the method should throw an `IllegalArgumentException` and not modify its current object graph. Note that since this is an unchecked exception (one of Java's library exceptions) it is not necessary that the method declare that it could be thrown, however we added it to the method's signature anyway just to be explicit.

3.3 `void addEdge(V source, V dest, int cost) throws IllegalArgumentException`

This method should add a new edge to its current object graph that goes from vertex `source` to vertex `dest` (a directed edge). Note that since the graph is directed this does not imply that the graph has an edge from `dest` to `source`, although one may also be separately added. The edge added should have weight `cost`.

If either `source` or `dest` are not already present in the graph they should be added by this method. If there is already an edge in the graph between `source` and `dest` the method should **replace** it with the edge being added, with cost of `cost`. As a result, in this project it's not possible for graphs to have multiple edges from a source vertex to the same destination vertex. In this project graphs cannot have negative weight edges, so if `cost` is negative the method should throw an `IllegalArgumentException` and not modify the graph. A edge with a cost of zero is not a problem.

3.4 `int getEdge(V source, V dest)`

This method should return the cost of the edge in its current object graph between the vertices `source` and `dest`. If either `source` or `dest` are not present in the graph then there obviously can't be an edge between them, in which case the method should return -1. Since the `addEdge()` method ensures that negative–weight edges can never be added to a graph it is known that no real edge will have a cost of -1. If `source` and `dest` are both in the graph but there is no edge between them, the method should also return -1.

3.5 `Iterable<V> getNeighbors(V vertex) throws IllegalArgumentException`

This method should return some type of class (that you have to write) implementing Java's `Iterable` interface, containing all of the neighbors of the vertex `vertex` in its current object graph. "Neighbors" means all the vertices that are successors of `vertex`, so if there is an edge from another vertex to `vertex` but not one the other way around, that other vertex should not be included in the result object. The returned object's iterator (returned by calling its `iterator()` method) should return each neighbor of `vertex` in the graph (there may be zero or more neighbors) once and only once before its `hasNext()` method returns false.

When calling the `iterator()` method on the object returned, the caller of this method cannot assume that the vertices will be iterated over in any particular order, only that the object returned has an `iterator()` method that can be called. To see what vertices are in the object returned the caller of course may iterate across it, or, while iterating over it they can store its elements in some type of Java collection that does have an ordering if they want to see the neighbors in some sort of predictable order.

If `vertex` is not present in the graph the method should throw an `IllegalArgumentException`. If `vertex` is present but has no neighbors an `Iterable` object should be returned that will have nothing in its iteration (`hasNext()` will initially be false when called on the iterator returned by its `iterator()` method).

3.6 `void removeEdge(V source, V dest) throws NoSuchElementException`

This method should remove the edge going from `source` to `dest` in its current object graph. If either `source` or `dest` are not present in the graph, or there is no edge between them, the method should throw a `NoSuchElementException` (also a Java library exception) and not modify the graph.

3.7 `void removeVertex(V vertex) throws NoSuchElementException`

This method should remove the vertex `vertex` from its current object graph. If `vertex` is not in the graph it should throw a `NoSuchElementException` and not modify the graph.

Removing an edge is straightforward, because vertices can exist even if they have no associated edges, but edges cannot exist unless both their source and destination vertices are present in the graph. Therefore, in the process of removing `vertex` (assuming it was present), all of its **outgoing and incoming** edges in the graph must also be removed; there may be zero or more outgoing edges from it or incoming edges to it.

3.8 `Iterator<V> iterator()`

This method should return an iterator over all of the vertices in the graph. It should return each vertex in the graph (there may be zero or more) once and only once before its `hasNext()` method returns false. There is no required order that the vertices should be returned in, but a user should be able to determine whether a vertex is present in a graph by successively calling the `next()` method and comparing the vertex being searched for against what it returns (recall that the vertices are comparable because the generic type parameter `V` implements the `Comparable` interface).

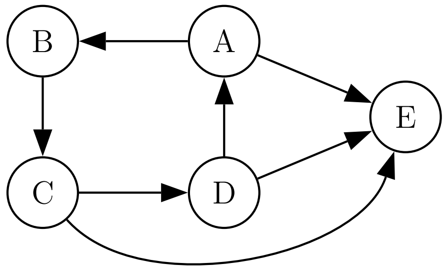
(If you use this method in your other methods, keep in mind that things may not work right if an object is modified while it is being iterated over.)

3.9 `boolean isInCycle(V vertex) throws IllegalArgumentException`

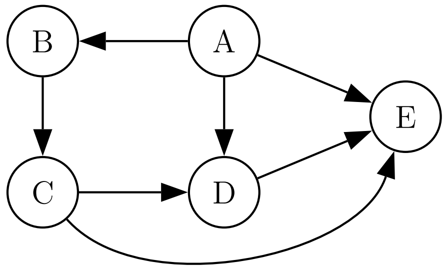
This method should return true if `vertex` is a part of the current object graph that contains a cycle and false otherwise. In other words, it returns true if the current object graph contains a cycle that is reachable from `vertex` (note that `vertex` itself may not be part of the cycle), and false otherwise. If `vertex` is not present in the current object graph the method should throw an `IllegalArgumentException`.

One problem involving graphs that might need to be solved is to determine whether a graph contains a cycle somewhere. This method is to solve a slightly different problem, which is to detect whether a graph has a cycle that is reachable from its parameter vertex. (Cycle is defined here to be a path consisting of one or more edges in which the start vertex is the same as the ending vertex; of course any vertex on a cycle could be chosen to be the start vertex and a path around the cycle would end at it.) A graph could have cycles not reachable from `vertex`, but unless some cycle is reachable from it the method should return false.

The algorithm is a variation of depth–first search. It would seem that simply performing a traversal of a graph (either depth–first or breadth–first), starting from a particular vertex, would reveal that there was a cycle if in the process of traversing a vertex was encountered that had been seen before (either using tags to indicate which vertices had been seen, or a visited set). For example, in the first graph below, when starting any type of traversal at vertex A, the traversal would eventually reach A again, and there is a cycle reachable from A.



However, consider the second graph below it, which is the same as the one above it, with just the edge from A to D reversed. If a traversal– suppose the recursive version of depth–first search– (assuming that vertices are processed in alphabetical order) is begun at A it will call itself upon and visit B, C, D, and E, then the recursive call will return and the next two recursive calls on the other neighbors of A (D and E) will encounter vertices that have already been seen, however, there is no cycle in this graph.



The difference in these two situations is that in the first graph a vertex that had been seen before would be encountered later in the process of performing the **same** recursive call (in the process of making another recursive call directly or indirectly from that recursive call), while in the second situation a vertex that had been seen before would be encountered after that recursive call returned, during a separate, **later** recursive call. The former situation means a cycle is reachable from the vertex, while the latter does not.

These two situations can be distinguished between as follows. In the traversal algorithms given in class vertices are marked (either using tags, or using a visited set) in one of two ways: either not traversed or visited yet, or already traversed or visited. What we can do is to generalize this in the recursive depth–first search algorithm by marking vertices in three different ways, one meaning it has not been visited, one meaning it has finished being visited, and one meaning it is currently in the process of being visited but it has not finished being visited (a recursive call has been made on it, but the call has not yet returned). The algorithm below marks vertices in three ways: “unprocessed”, “active”, and “done” (instead of just true or false as the traversals in class did), but any three different possibilities could be used (such as red, green, or blue). Section 2 above discussed how vertices could be marked for this algorithm and Dijkstra’s algorithm below. Before the algorithm is called all the vertices would have to be marked as “unprocessed”.

```
boolean isInCycle(vertex X)
    mark X as “active”
    for each neighbor Y of X do
        if Y is marked “active”
            then return true   // a cycle reachable from X was found
        else
            if Y’s status is not “done” and isInCycle(Y) returns true
                return true
    end for
    mark X as “done”
    return false
end
```

This algorithm is what your method should implement. Note that just because one recursive call returns false in the last line does not mean that the algorithm might not return true; a later recursive call may do so.

3.10 `ShortestPath<V> Dijkstra(V source, V dest) throws IllegalArgumentException`

This method should perform Dijkstra’s algorithm on its current object graph, starting from `source`. Dijkstra’s algorithm determines the shortest path from the starting vertex to every other vertex, but the method should return the result for just the path between `source` and the vertex `dest`, as follows. Since the method should return two pieces of information, the shortest path between `source` and `dest` and also its cost, and the mechanism in Java to aggregate multiple things of different types together is to put them inside an object, we created a class `ShortestPath` (also in the `graph` package) for this purpose. It just has two fields, a path (an `Iterable` reference) and a cost (an `int`), and two pairs of set and get methods. The returned `ShortestPath` object should contain in its integer `shortestPathCost` field the cost of the path from `source` to `dest`, and an object with an iteration over the vertices along the shortest path should be stored in its `path` field. The vertices must be stored in the path such that they will be iterated over in order in the returned object, with `source` being the first element and `dest` being the last. If `source` and `dest` are the same vertex then the cost in the returned object should be 0, and `path` in the returned object should be a path consisting of just one element, namely `source` (which will be the same as `dest`). If there is no path in the graph from `source` to `dest` the cost in the returned object should be -1, and `path` in the returned object should be an object that will have an empty iteration.

If there are two or more paths in the graph that have the same (equal) smallest weight, it doesn't matter which one your method returns in the `ShortestPath` object.

If either `source` or `dest` are not in the graph the method should throw an `IllegalArgumentException`.

4 Project requirements and related issues

1. Using an adjacency or incidence matrix to implement graphs will result in **significant** loss of credit on the project, since one main purpose of the project is to learn about and use one of the other representations.

2. One purpose (perhaps more important than getting experience using graphs) of this project is for you to create your own data structures, so other than the following you **may not** use any arrays, or any Java library collection (container) classes. If you need any data structures, such as a list, or a stack, or a set, or a map, to store components of graphs or to implement the algorithms described, you must writ it **yourself**. Of course you may reuse any code that you wrote in the earlier projects, either as is or modified if needed.

   - You may use arrays to implement data structures that we covered this semester. **However, your project must use at least one (user–written) linked data structure.**

     Dijkstra's algorithm was explained in class using arrays, so you are allowed to use arrays in it. However, arrays were used in the algorithm just for ease of explanation, and when you actually implement the algorithm you will see that arrays are actually not the best data structure to use in it.

   - You may use `Strings` to store strings (for example, names of things) if needed. You **may not** use `Strings` (or `StringBuilders` or `StringBuffers`) as data structures (you may not use them to store multiple pieces of data). Other data structures should be created and used.

   - You may use the `Iterable` and `Iterator` classes as needed so that you can create classes that implement `Iterable`. (Of course these interfaces are not themselves container or collection classes.

   Other than the above, you **may not use any Java library container or collection classes, or arrays**.

3. If you use arrays to implement any data structures that might need to store components of graphs or other data structures needed by the algorithms, note that there is **no maximum size** for graphs in this project– a graph may contain any number of vertices or edges– so you will need to make your data structure's capacity be able to increase when the array becomes full. (Using linked data structures rather than arrays, of course, will avoid the need to do this.)

4. You should try to use appropriate data structures, but efficiency doesn't matter, meaning that you do not have to use data structures that would perform operations the fastest.

5. Due to issues involved in instantiating a generic class from another generic class, if you want to use a binary search tree it would be easier to write a non–polymorphic one than to try to use your polymorphic tree.

6. You may add whatever classes and methods to the project that you like, but you must retain the package structure in the initial code distribution, meaning that the classes above **must** be kept in the `graph` package, and any new classes must be added there.

7. You may include whatever fields you need in the `Graph` class, subject to the stipulations above about what graph representations you may not use. You may add any private methods you like to the `Graph` class. You may add any classes to the `graph` package or the`tests` package (but do not add any other packages), and may add any methods to the `StudentTests` class. (Of course keep in mind the requirement in the project grading policy that fields and methods should be given an appropriate access method.)

8. The restrictions above do not apply to your student tests; you may use any Java classes and language features you like in writing them.

9. This is not a reqirement, but rather a recommendation: spend the time to write a `toString()` method in all your classes. Well–designed practical Java classes almost always have an `equals()` method, and therefore a `hashCode()` method, and also a `toString()` method. Creating `toString()` methods will help with debugging because it will allow you to see the contents of your data structures to much more easily figure out what's going on.

10. The `getNeighbors()` method directly returns an `Iterable` object, and an `Iterable` object is also returned in the `ShortestPath` object returned by the method `Dijkstra()`. In creating objects that have iterators you do not have to implement the `remove()` method; it can simply throw an `UnsupportedOperationException` if desired. (Of course you may implement it if you would find it useful).

11. All of your source code will be graded for style, meaning if you reuse code from an earlier project it will be graded again, so it would be a good idea to make any changes suggested earlier by the grading TAs.

12. You **may not** use any sorting algorithm in writing the project, meaning neither any that you code yourself nor any from the Java library such as `Collections.sort()`. (Note that the public tests use a sorting algorithm, but your graph may not.)

13. For all methods that have any type of object or objects as parameters, if any object parameter is `null` the method should throw a `NullPointerException`. Since this is an unchecked exception it is not necessary that any methods declare that it could be thrown. (Note that you may not really have to do anything special to cause this to happen.)

14. Grades for this project will be based on:

public tests	50 points
secret tests	30 points
student tests	10 points
source code and programming style	10 points

15. After you submit you must **log into the submit server** and verify that your submission compiled and worked correctly there!

5 Academic integrity statement

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.