Nathan Glover

15164755

27 October 2016

Data Structures & Algorithms

Assignment | Semester 2, 2016

Problem

Description

As detailed in the assignment brief, we have been tasked with the problem of computing network reliability at scale using the two-terminal directed network approach. My implementation has taken into consideration as many precautions as possible to ensure computation is quick and efficient whilst still maintaining an easy to understand methodology.

On top of the justifications and descriptions in the following document, you may also refer to my generated javadocs located in the `docs/` folder of the submission structure.

The project UML can also be viewed in the root folder under the name `assignment-uml`.

The project folder was under version control and the private repository can be provided if authenticity of my work flow needs to be inspected.

Please don't hesitate to contact me at nathan@glovers.id.au or my student email nathan.glover@student.curtin.edu.au if you have any queries.

Usage

In order to load and sort a non-sorted file, you can use the following layout for your command line arguments.

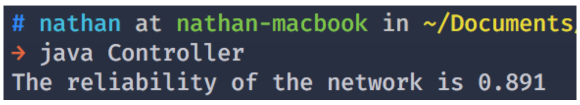
$ java Controller grid_2x5u.nt

Having an empty argument with default the program to use the "sample.nt" network file.

$ java Controller

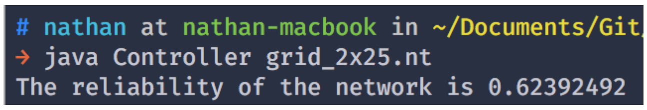
Sample Output

Default argument (no argument):



```
# nathan at nathan-macbook in ~/Documents
→ java Controller
The reliability of the network is 0.891
```

Non sorted file input argument:



```
# nathan at nathan-macbook in ~/Documents/Git
→ java Controller grid_2x25.nt
The reliability of the network is 0.62392492
```

Class Descriptions

BreadthFirstSearch

Filename: BreadthFirstSearch.java

Test Harness: NetworkUnitTest.java

Author: Nathan Glover

Description: Class used to traverse the Network data structure and sort the contained vertices and edges based on their depth through adjacent nodes in a completed network.

Design:

The BreadthFirstSearch has four key private class fields which are all instances of my DSAQueue extending the DSALinkedList data structure. The first two (seen below) hold references to the preallocated and populated vertices and edges queue that live in the Network class instance.

```
private DSAQueue<Vertex> vertices;
private DSAQueue<Edge> edges;
```

The next two fields are similar structures to the vertices and edge queue, however these are specifically used as a place for Devices to be placed as they are sorted.

```
private DSAQueue<Vertex> outVertices;
private DSAQueue<Edge> outEdges;
```

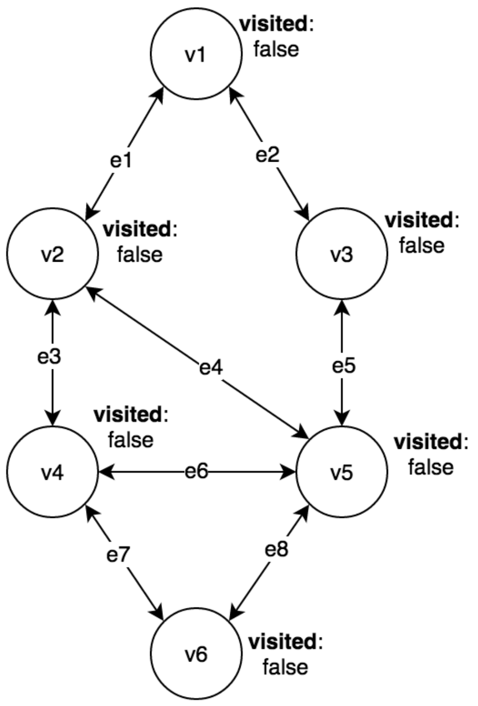
The reason behind using a queue for my data structures was in part due to the way I've designed my Network class. The use of a queue for both the 'vertices' and 'edges' might not be obvious right now, however; 'outVertices' and 'outEdges' queues are being used to house a sorted list of vertices and edges. This means that devices much be ordered in such a way that the vertex SOURCE is expected at the beginning of a list and the TARGET at the end. The same goes for edges that are utilised along the path from the first to the last vertex.

The BreadthFirstSearch constructor takes two arguments:

```
DSAQueue<Vertex> inVertices
DSAQueue<Edge> inEdges
```

These arguments are designed to be the references to existing DSAQueue's containing unordered vertices and edges. Once initialised we will be calling a few other methods that will take our unordered queue of devices and reorder them correctly based on Breadth First traversal. Once we have references to the existing edge and vertex queues we are ready to begin our sorting process. I initially call the resetDiscoveries method to iterate through all the devices and set their 'discovered' private class field to false. The discovered class field will be used during sorting as a way to keep track of whether or not we have visited a device as we traverse through out network structure.

This bring me to my final core method, breadthTraverse. This method is the heart of my sorting method and is based heavily on the ideas presented in the references section of this class description.



To the left you can see an example of how a connected network might look before we begin the breadthTraverse method. Note that I have chosen not to include visited labels for the edges to reduce complexity in the diagram, the visited/discovered states are still includes and used in the algorithm.

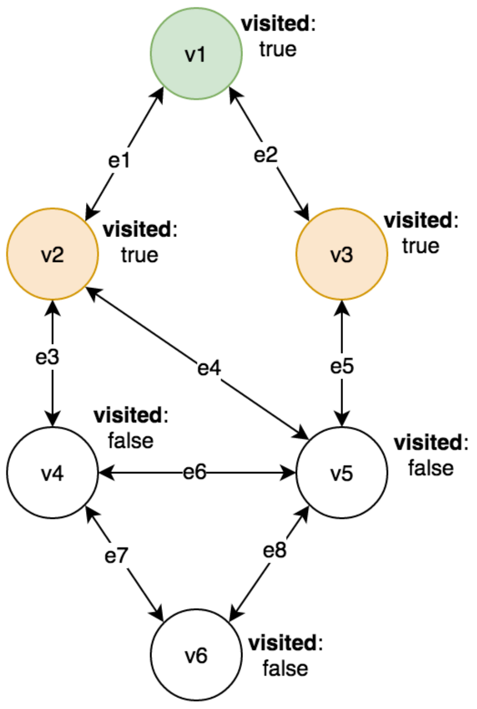
We start by finding our source vertex and setting it to the discovered state, then enqueuing it into our outVertices queue.

Next we call the adjacentV method with our source vertex as our Vertex parameters and also pass it a reference to the outEdges queue.

The adjacentV method iterates through all the edges in the network and looks for edges with FROM connection to the inVertex parameter, using the unique name as its comparator. If a matching vertex is found on the other end of an edge then we proceed to check if the edge has been discovered before. If it hasn't been then we set the edge to discovered and enqueue it into the DSAQueue<Edge> edgeArray that we reference from the input parameter. Finally we enqueue the vertices attached to the adjacent edge we just found and move on.

If the edge found has a TO connection matching the inVertex parameter we perform the same operation as above, however only if the connecting is not directed.

Finally we return the vertexQueue once we've checked all edges in the network. This vertexQueue is a data structure containing all the adjacent vertices in reference to the aforementioned inVertex.



Returning back to our breadthTraverse method we iterate through the queue of vertices we just found and if any of them haven't been discovered, we set them to discovered and enqueue then into another Queue structure called visiting. We also queue the same vertices into our outVertices queue which gives us the beginning of our sorted device arrays.

The updated diagram on the left shows that vertices in green have had their adjacent vertices checked and no longer exist in the visiting queue anymore. Vertices in orange are nodes that have been discovered, but still require their adjacencies to be checked.

This similar process continues until the base case for our while loop is finally achieved; which is when our visiting queue empties, which will only feasibly occur once the target vertex is processed and enqueued into our outVertices queue.

Justification:

The main justification I feel as though I need to make is in regards to my DSAQueue extending the DSALinkedList data structure. The primary benefit to using a queue managed by a linked list is that I don't need to know how many of the devices I will be working with in order to initialise it.

Because the number of devices being used is likely to vary with unknown bounds, I felt it would be a much better idea as it means that I would not have to keep track of a device count in order to initialise an ArrayQueue to the correct length.

The other reason for using this data structure is because I doesn't need to do anything very complicated other than enqueue and dequeue ordered nodes. The LinkedList queue provides just enough functionality whilst still maintaining an elegant level of simplicity.

References:

https://www.youtube.com/watch?v=qBfzDxihJz4

https://www.cs.bu.edu/teaching/c/tree/breadth-first/

https://visualgo.net/dfsbfs

https://en.wikipedia.org/wiki/Breadth-first_search

https://tutorialedge.net/breadth-first-search-with-java

http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/11-Graph/bfs.html

https://www.youtube.com/watch?v=0u78hx-66Xk

Controller

Filename: Controller.java

Test Harness: ControllerUnitTest.java

Author: Nathan Glover

Description: The main entry point to my program, calls OBDDH with appropriate input argument handing

Design:

Works by allowing the user to either specify a network file themselves, else it will just load the 'sample.nt' file from the current program directory.

An example of usage would be:

```
$ java Controller grid_2x5u.nt
```

Device

Filename: Device.java

Test Harness: DeviceUnitTest.java

Author: Nathan Glover

Description: Device class is an abstract class used by both Vertex and Edge classes. It contains as many of the standard device functions as possible to simplify the implementations for the two classes that inherit from it.

Design:

As mentioned above, Device is an abstract class containing all the similar class fields and methods between Vertex and Edges.

The private class fields I chose to include as a part of Device are the following:

```
private String type;
private String name;
private FloatingPoint reliability;
private int cost;
private boolean discovered = false;
```

The methods within this class are mostly getters and setters, and the input validation is fairly simple; especially for methods that I intend to override.

Justification:

Device class was chosen to be abstract to avoid the diamond problem that is typical with multiple inheritance. I kept the input validation fairly simple as mentioned above to give myself some openendedness later on if specifications were to change.

Another justification I feel I have to make is in regards to my cost int value. In the assignment specifications I noted that the Vertex cost is associated with cost in dollars (US) per unit, whereas the cost for edges is in cents (US) per meter. neither of these values require decimal precision in the sample network files, so i've opted not to account for it and stick with Integers.

References:

- Note that I used my own practical 01 code for the Device class included in this assignment.

DSABinarySearchTree

Filename: DSABinarySearchTree.java

Test Harness: DSABinarySearchTreeUnitTest.java

Author: Nathan Glover

Description: Generic implementation of a binary search tree. The iterator within the class is built onto of a DSAStack

Design:

The DSABinarySearchTree class was designed following the guidelines outlined in the lecture slides provided by our unit co-ordinator.

When possible a recursive method was used to traverse the tree maintain simplicity and consistency across method design. The binary search tree is able to offer my Qn class a much faster search method when performing the add() function.

References:

- Note that I used my own practical 09 code for the DSABinarySearchTree included in this assignment.
- http://introcs.cs.princeton.edu/java/44st/BST.java.html

DSALinkedList

Filename: DSALinkedList.java

Test Harness: DSALinkedListUnitTest.java

Author: Nathan Glover

Description: Generic implementation of a double-ended LinkedList. Contains methods for adding and removing entries from the head and tail of a chained link of nodes.

Design:

The design followed for my LinkedList was based primarily on specifications outlined in DSA COMP1002 practical 8. The methods included within the basic design were as follows:

```
public void insertFirst(E value)
public void insertLast(E value)
public E peekFirst()
public E peekLast()
public E removeFirst()
public E removeLast()
```

The methods above give me all the functionality I need to extend my previously designed DSAStack and DSAQueue classes. If you think about how a Stack and Queue work, we either have a LIFO (Stack) or FIFO (Queue) data structure. This means that in order to extend either of these classes with a LinkedList we need at minimum a way to insertFirst, removeFirst (Stack) and insertLast, removeFirst (Queue). Obviously given these requirements it's quite clear that our LinkedList fits this criteria perfectly.

```
public Iterator<E> iterator()
```

Another very helpful function included as part of my LinkedList is an Iterator. This class is an override from the Iterator<E> class and returns reference to another private subclass I wrote called DSALinkedListIterator<E>. The DSALinkedListIterator class contains method overrides for the following functionality:

```
public boolean hasNext()
public E next()
public void remove()
```

Justification:

The only thing I feel as though I need to justify for this class would be the use of Generics. The generics allow me to extend my DSAQueue and DSAStack much more effectively as I can store practically any kind of data in the data structure.

References:

- Note that I used my own practical 08 code for the DSALinkedList included in this assignment.

http://www.oxfordmathcenter.com/drupal7/node/637

DSAQueue

Filename: DSAQueue.java

Test Harness: DSAQueueUnitTest.java

Author: Nathan Glover

Description: Queue data structure that implements Generics and is built on top of the DSALinkedList class.

Design:

The fundamental design factor for the DSAQueue implementation was to give myself a data structure that follows the FIFO (First-in-first-out) system. This means that entries enqueued first, will always be dequeue first (the opposite to a typical stack).

```
public class DSAQueue<E> implements Iterable<E>, Serializable
```

The queue itself implements Iterable; as I would be needing a quick way to run through all my entries from start to finish. It also makes use of the Serializable interface allowing me to save copies of my Queues using serialisable save/load functions.

```
private DSALinkedList<E> list;
```

The private field in the DSAQueue is simply a new DSALinkedList using the same generic type as the one used to create the new DSAQueue.

The following functions are implemented within the DSAQueue class:

```
public void enqueue(E inItem)
public E dequeue()
public E peek()
public boolean isEmpty()
public Iterator<E> iterator()
```

Note that I have no isFull method implemented here due to the way LinkedLists differ from typical array data structures. I tossed around the idea of setting a private class field defining the maximum allowed value, but quickly decided not to in order to keep complexity to a minimum.

Justification:

Similarly to DSALinkedList, the only thing I really need to justify here is the use of Generics. The generics allow me to extend my DSAQueue more effectively and also connect nicely with my generic DSALinkedList. Having the Queue implemented on top of a LinkedList also allowed me to not have to worry about list sizes, as I could theoretically keep adding new entries to my queue without having to worry about hitting a preset threshold (like a normal array).

References:

- Note that I used my own practical 04 code for the DSAQueue included in this assignment.

DSAStack

Filename: DSAStack.java

Test Harness: DSAStackUnitTest.java

Author: Nathan Glover

Description: Stack data structure that implements Generics and is built on top of the DSALinkedList class.

Design:

The DSAStack is constructed with the same methods as the aforementioned DSAQueue. Because its use is limited to being a class extension for my DSABinarySearchTree I won't spend too much time explaining its use.

References:

- Note that I used my own practical 04 code for the DSAStack included in this assignment.

Edge

Filename: Edge.java

Test Harness: DeviceUnitTest.java

Author: Nathan Glover

Description: Edge class inherits from the abstract Device class. Its function is to override and extend the basic functionality that comes with the base Device class.

Design:

This class, similar to the Vertex class contains a lot of specifics that help define the functionality expected from an Edge.

The first set of class fields are used to define the different types of edges our network should support:

```
public static final String DEVICETYPE_CAT4 = "CAT4";
public static final String DEVICETYPE_CAT5 =  "CAT5";
public static final String DEVICETYPE_WIRELESS = "Wireless";
public static final String DEVICETYPE_SATELLITE = "Satellite";

private static final String[] edgeTypes = new String[]{
                        DEVICETYPE_CAT4,
                        DEVICETYPE_CAT5,
                        DEVICETYPE_WIRELESS,
                        DEVICETYPE_SATELLITE };
```

As per specifications, the four static device type can be seen in their raw form above. Below that you can see I've also created an array of these types that help later on when deciding if an input type is valid.

```
private Vertex FROM;
private Vertex TO;
private String s_FROM;
private String s_TO;
```

The next set of class fields (seen above) deal with storing references for the Vertex's on either end of the edge when connected in the network. There are public getters and setters for each of these fields to allow my NetworkIO and Network class to manipulate the contents of these fields when loading and connecting the network devices.

private boolean directed;

The final class field is the boolean variable defining whether or not the edge is directed. A directed edge means that we can handle transmission line connections that are one way.

The Edge class called the device super class when initialising the basic edge object. The setType method calls the checkType method within the class and if the inType String is valid it allows the object to have its type field set.

The setCost method differs a little bit from that in the Device and Vertex classes because of the requirement that some Wireless connections can have a cost of zero. There's input validation based on if the input cost is equal to zero, but the type is not Wireless.

[Note: I have interpreted the assignment sheet in such a way that wireless connections can cost money or be equal to zero, however they do not explicitly have to be equal to zero]

Justification:

The main justification I would like to talk about is the use of an edgeTypes String array to valid input types for my Edges. My thought process around this decision came down to extendability and to reduce code complexity. I knew at some point I would need to compare my input with the four different type variables and I didn't want to have to repeatedly reference the long variable names in multiple methods.

There's also a checkType helper function that could be made public later on down the track to give outside developers access to a method that could pre-validate their input types.

References:

- Note that I used my own practical 01 code for the Edge class included in this assignment.

FileIO

Filename: FileIO.java

Author: Nathan Glover

Description: A abstract class to handle basic FileIO functionality

Design:

Has been designed with limited functionality intentionally, just deals with user input from the console prompting for a filename.

Justification:

The reason I separated FileIO into a different class to NetworkIO was a decision I made early on when completing practical 02. My intension was to eventually have a couple different classes that might need to open and close different files. Obviously now I just have the one (NetworkIO.java), and I could have consolidated my code into one file, however I felt the benefits of keeping them modular was a much better option.

FileIO is abstract to avoid the multiple inheritance diamond problem; not that while in this case we wouldn't have the problem mentioned above, I designed the class specifically incase I have other IO classes needing to inherit from this base class.

References:

- Note that I used my own practical 02 code for the FileIO included in this assignment.

FloatingPoint

Filename: FloatingPoint.java

Test Harness: FloatingPointUnitTest.java

Author: Nathan Glover

Description: FloatingPoint is specified as an ADT responsible for safe handling of floating point numbers.

Design:

The FloatingPoint class is responsible for safe handling of floating point numbers. The term 'safe handling' is difficult to understand without some background.

The issue comes down to a fundamental problem with precision in floating point maths. Typical arithmetics with integers are exact (usually unless the answer falls outside of the integers memory). Floating point however numbers that have a problem dealing with rounding when you have recurring results (3.333333~). Floating point numbers cannot accurately be expressed in binary form in memory and you will usually have odd trail off precision lost during the arithmetic process. Look at the example below:

$$0.1 + 0.1 + 0.1 = 0.3$$

The maths above makes sense to you, and as humans it is correct. However computers don't store the values we just provided in a way that makes equality checks work well. In fact if you subtracted 0.3 from the previous operation you'd end up with a results of something like 5.551115123125783x10^-17.

The floating point class will handle this precision problem ensuring that we always keep our reliability precision set correctly to a given decimal place.

After investigating a couple different options I decided on the use of BigDecimal class to assist with housing precision for my in and out double values.

Justification:

The main justification is around the use of the BigDecimal class is its availability in the core java.math.* library. I figured that if a tool was already available that could solve this problem for me, why re-invent the wheel.

Big Decimal also supports an easy method of adjusting the scale and precision of a given number. The setScale function gives me access to a number of different rounding modes, however it requires the knowledge of how many zeroes are present or required in precision.

I handled this problem with my getZeros function, which takes the current static double value of 'precision' and multiplies it by a factor of 10 until the number becomes greater than one. Then I return the count which is equal to the number of zeros places in the precision.

The downside to using BigDecimal is in the way it deals with its precision in an arbitrary manner. The process of increasing and decreasing the length of an existing figure has a significant enough overhead that a lot of thought had to be put into why I opted for this library.

In the end I decided that the convenience made up for the overhead, and because the precision value is static and won't be altered regularly It's fair for me to assume that we won't be faced with the problem of shifting our precision very regularly.

References:

https://raw.githubusercontent.com/tarelli/jscience/master/src/org/jscience/mathematics/number/FloatingPoint.java

https://raw.githubusercontent.com/latkseb/latkseb-IEEE754CalcTesting/master/src/FloatingPoint.java

http://stackoverflow.com/questions/588004/is-floating-point-math-broken

http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

http://floating-point-gui.de/formats/exact/

HNode

Filename: HNode.java

Author: Department Provided

Description: As detailed in the assignment documentation, HNode is a node in a hybrid decision diagram.

Design:

The following class fields and methods were provided by the department within this class:

```
private FloatingPoint reliability;
private HState state;

public HNode( Vertex source )
public HNode( HNode n )
public boolean isEqual( HNode node )
public void mergeInto( HNode node )
public HNode makePos( Vertex v, Edge elist[] )
public void makeNeg( Vertex v, Edge elist[] )
public boolean isSuccess()
public boolean isFailed()
public FloatingPoint getRel()
public int getCode()
public void display()
```

HState

Filename: HState.java

Author: Department Provided

Description: HState is a department provided class with two internal private classes it makes use of called Block and BlockEntry.

Design:

The following class fields and methods were provided by the department within this class. Note that I have not included the two internal classes within this description.

```
private Block head;
private Block tail;
private static boolean delFrom = false;
private static boolean delTo = false;
private static boolean addTo = true;
boolean success;
boolean failure;

public HState(Vertex source)
public static void setDelFrom()
public static void resetDelFrom()
public static void setDelTo()
public static void resetDelTo()
public static void setAddTo()
public static void resetAddTo()
public HState(HState s)
public boolean isEqual(HState s)
public void contract(Edge e)
void check(Vertex v)
public void delete(Edge e)
public void display()
```

Network

Filename: Network.java

Test Harness: NetworkUnitTest.java

Author: Nathan Glover

Description: Network is primary class that holds all the nodes and connection statuses for the Devices loaded in from file. It houses most of the core methods for manipulating the network state.

Design:

The actual class fields for a Network are quite simple and I only implement two DSAQueue's for my vertices and edges respectively.

```
private DSAQueue<Vertex> vertices;
private DSAQueue<Edge> edges;
```

When a Network is initialised it expects a network filename to be input as a String parameter. As per specifications, if the filename contains ".srt" then the file is assumed to already be sorted and doesn't need to be sorted by my Networks sorting method; else if the filename contains ".nt" then it should be concluded that the file being read in is already sorted in the order outlined in Appendix 1 of the assignment documentation. Once the network is either loaded in or sorted, I save the new network structure back out to the ".srt" format.

The sorting of devices is handled by a call to the BreadthFirstSearch class and passing it a reference to the Network vertex and edge DSAQueues.

I also offloaded my FileIO functionally to the NetworkIO class so I've got references to these save/load methods baked into my Network class. Note that the loadNetwork/saveNetwork methods is the default method for parsing in and out the contents of the network.

```
private void loadNetwork(String inFilename)
private void saveNetwork(String inFilename)
public void saveXML(String inFilename)
public void loadXML(String inFilename)
public void saveSerializable(String inFilename)
public static Network loadSerializable(String inFilename)
```

Justification:

The use of my DSAQueue class in the Network class is due to the importance of the order of nodes when loaded in from a sorted file. The queue data structure fits all the requirements for the base Network class as there aren't any complex methods that require anything other than what the typical Queue can deliver.

The Queue implemented in this class is actually one built onto of a generic LinkedList so we get the added benefit of not having to worry about keeping track of how many device we have before initialising or resizing the data structure. It scales and performs very well when the device number is low.

References:

- Note that I used my own practical 01 code for the Network class included in this assignment.

NetworkIO

Filename: NetworkIO.java

Author: Nathan Glover

Description: The class in charge of handling file loading and saving of network contents. It was designed to Inherit from the abstract FileIO class.

Design:

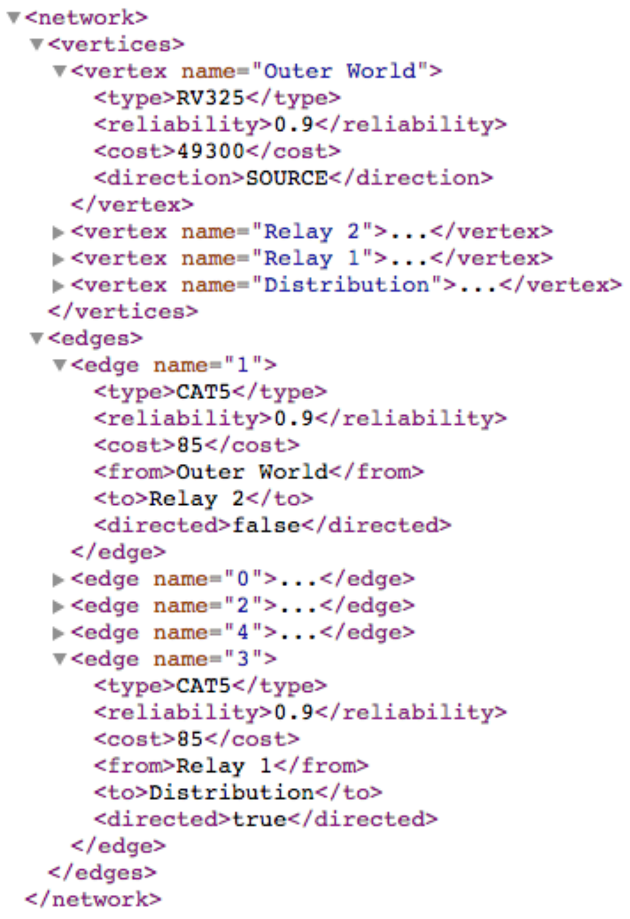
For this class I chose very early on that I would make use of regular expressions to find and extract all the relevant information for the imported network file. Taking a look at the class fields below you can find the patterns I designed for each specific parameter on a typical line specified in the assignment documentation.

```
private static Pattern beginPattern = Pattern.compile("(<NET>)");
private static Pattern endPattern = Pattern.compile("(</NET>)");
private static Pattern vertexPattern = Pattern.compile("(VERTEX)");
private static Pattern edgePattern = Pattern.compile("(EDGE)");
private static Pattern reliabilityPattern = Pattern.compile("REL=(\\d.?\\d+)");
private static Pattern typePattern = Pattern.compile("TYPE=\"(.+?)\"");
private static Pattern costPattern = Pattern.compile("COST=(\\d+)");
private static Pattern namePattern = Pattern.compile("NAME=\"(.+?)\"");
private static Pattern fromPattern = Pattern.compile("FROM=\"(.+?)\"");
private static Pattern toPattern = Pattern.compile("TO=\"(.+?)\"");
private static Pattern sourcePattern = Pattern.compile("(SOURCE)");
private static Pattern targetPattern = Pattern.compile("(TARGET)");
private static Pattern directedPattern = Pattern.compile("(-u)");
```

The loadFile and saveFile methods are completely compatible with each others input/output and I was very careful to ensure that the information I saved matched the structure of the sample.nt and other network files supplied by the department as to ensure other students designs were compatible with my files.

On top of the regular file IO, I also put together an XML save/load method set that utilises the DocumentBuilder classes (please see imports at the top of NetworkIO.java file).

These methods build an easy to understand XML file with all the necessary information contained inside it to rebuild the network if needed. You can see an example of the XML output below:



```
▼<network>
  ▼<vertices>
    ▼<vertex name="Outer World">
       <type>RV325</type>
       <reliability>0.9</reliability>
       <cost>49300</cost>
       <direction>SOURCE</direction>
     </vertex>
    ▶<vertex name="Relay 2">...</vertex>
    ▶<vertex name="Relay 1">...</vertex>
    ▶<vertex name="Distribution">...</vertex>
   </vertices>
  ▼<edges>
    ▼<edge name="1">
       <type>CAT5</type>
       <reliability>0.9</reliability>
       <cost>85</cost>
       <from>Outer World</from>
       <to>Relay 2</to>
       <directed>false</directed>
     </edge>
    ▶<edge name="0">...</edge>
    ▶<edge name="2">...</edge>
    ▶<edge name="4">...</edge>
    ▼<edge name="3">
       <type>CAT5</type>
       <reliability>0.9</reliability>
       <cost>85</cost>
       <from>Relay 1</from>
       <to>Distribution</to>
       <directed>true</directed>
     </edge>
   </edges>
 </network>
```

The final set of methods included allow for the network to be completely serialised out to a specified ".bin" file. The advantage of this is if you have information you need to store about the current network, but you don't want it to be human readable. It also is very effective and storing the entire network state, so when the network is reloaded it won't typically need to have connectNetworkDevices execute again in the Network class.

Justification:

The primary justification I feel as though I need to make is in my choice to use Regular expressions (regex) over StringTokenizer. I have two reasons the first of which is robustness.

Regular expressions are extremely specific when designed effectively and it allowed me to map out exactly what I was expecting for each parameter in the file input. Even with edge cases I was able to simple ignore any patterns that didn't follow the strict assignment specifications and it meant I didn't even have issues with extra spacing or unexpected characters.

The second and probably more reasonable reason I chose regular expressions over StringTokenizer was because StringTokenizer is a legacy class and is discouraged from being implemented in new code (see reference below).

StringTokenizer is a legacy class that is retained for compatibility reasons although its use is discouraged in new code. It is recommended that anyone seeking this functionality use the split method of String or the java.util.regex package instead. (http://docs.oracle.com/javase/6/docs/api/java/util/StringTokenizer.html)

References:

- Note that I used my own practical 02 code for the NetworkIO included in this assignment.

http://docs.oracle.com/javase/6/docs/api/java/util/StringTokenizer.html

OBDDH

Filename: OBDDH.java

Author: Nathan Glover & Department Provided

Description: The main diagram class used to initialise the network and generate all the nodes used to compute the reliability.

Design:

This class was partially provided to use by the Department. The following class fields and methods were pre designed for us:

```
private FloatingPoint reliability;
private Qc current;
private Qn next;
private Network net;
private static int DEFAULT_Qn = 101;
Vertex v;
Vertex nextV;

public OBDDH( String filename )
private void compute()
public void displayRel()
```

On top of the methods outlined above, we were tasked to complete two methods.

```
private void addToQn( HNode n )
private void loadNetwork( String filename )
```

The loadNetwork method was a simple initialisation of the Network class using the OBDDH input filename. The other method, addToQn needed to check if the node was successful, and if so to add the input nodes reliability to the current tally. Else if the node is terminal and successful I add the node to the Qn data structure.

Qc

Filename: Qc.java

Test Harness: QcUnitTest.java

Author: Nathan Glover

Description: Data structure in charge of storing HNodes at the current level of the OBDDH. This is the level that would currently being processed during runtime.

Design:

At this point in time I've implemented the Qc data structure with a DSAQueue. You can see the two private class fields currently in use below:

```
private DSAQueue<HNode> nodeList;
private int nodeCount;
```

When nodes are enqueued and dequeued from the data structure I update a count variable that can be used to track how large each level gets during the compute process.

Justification:

The reason I decided on a DSAQueue backed onto a LinkedList as my data structure came down to the unknown nodes that will be inserted into any given level. I do acknowledge that the functionalities associated with a typical queue aren't really necessary because it was specified that node order isn't an important factor. However an insert, remove and isEmpty check are required for this class and the DSAQueue methods are already well suited for these tasks.

The LinkedList supporting the queue has a O(1) complexity when inserting which is the primary function being used by this class.

References:

https://github.com/grid32/JUNG

Qn

Filename: Qn.java

Test Harness: QnUnitTest.java

Author: Nathan Glover

Description: Data structure in charge of storing HNodes at the next level of the OBDDH. This is the level that would below the level currently being processed during runtime. Nodes are added one at a time and only if an equal node isn't already present.

Design:

Qn went through a couple redesigns during the course of this project. Initially I used a DSAQueue backed onto a LinkedList in the end for similar reasons to the Qc class. however after a couple revisions I eventually ended up picking a Binary search tree (BST).

Something slightly different with this class is the maxSize field that can be set via the Qn alternate constructor.

Justification:

My reasoning behind using a DSABinarySearchTree came down to a few variables.

It was very tempting to simple pick a DSAQueue and keep it simple like the Qc class; however the overhead associated with the add() method was a factor I had to think about. The 'add' method requires a search to be performed which is O(N) complexity. The assignment sheet mentions that this method **has** to be as efficient as possible so this isn't the most optimal solution.

I toyed with a few ideas revolving around the use of my DSAHashTable as well, but however in order to completely empty the data structure for the 'swap' method would have needed a list of keys to be stored to quickly remove all the keys in an optimal fashion.

The variable I used a my key was the HNode 'id' field, as it was unique to each node.

References:

https://github.com/grid32/JUNG

Vertex

Filename: Vertex.java

Test Harness: DeviceUnitTest.java

Author: Nathan Glover

Description: Vertex class inherits from the abstract Device class. Its function is to override and extend the basic functionality that comes with the base Device class.

Design:

This class, similar to the Edge class contains a lot of specifics that help define the functionality expected from an Vertex.

The first set of class fields are used to define the different types of vertices our network should support:

```
public static final String DEVICETYPE_RV325 = "RV325";
public static final String DEVICETYPE_2291K9 = "2291K9";
public static final String DEVICETYPE_1841 = "1841";

private static final String[] vertexTypes = new String[]{
                        DEVICETYPE_RV325,
                        DEVICETYPE_2291K9,
                        DEVICETYPE_1841 };
```

The final class field is the String variable defining whether or not the vertex is a SOURCE or TARGET. The field would only typically be used with two devices per network graph as normally we'd only expect one Source and one Target.

```
private String commDirection;
```

The Vertex class called the device super class when initialising the basic vertex object. The setType method calls the checkType method within the class and if the inType String is valid it allows the object to have its type field set.

The setCommDirection method is the other more specific method in this class. It just simply ensures that the input String is either SOURCE or TARGET when called.

Justification:

The main justification here is again in relation to the vertexTypes String array. It's again also used to validate input types for my vertices. The thought process around this decision came down to extendability and to reduce code complexity. I knew at some point I would need to compare my input with the three different type variables and I wanted to have a way to repeatedly reference the long variable names in multiple methods.

There's also a checkType helper function that could be made public later on down the track to give outside developers access to a method that could pre-validate their input types.

References:

- Note that I used my own practical 01 code for the Vertex class included in this assignment.

Work Log

Below is a list of the commits I made during this project. If the marker would like access to the repository history to verify my works authenticity please send me an email with the request.

Date	Description	Commit
27/09/2016	init	f44ce579ec24c03247354293e425308d8873236c
27/09/2016	FloatingPoint added	8dd101f056deb521f1ae10a08d003ab82a9a511b
27/09/2016	Added assignment documentation	9c337af3f4a7d8ea9aebc154a3823be9604f7928
28/09/2016	Finished some of floating point	19cfe1e5f8db8556d701d36fb55245eebfa86058
02/10/2016	Added devices and updated floatingpoint	97cbd68f6465bd9eeb3c64af29ec60fd53c5d1b0
02/10/2016	Updated unit tests	d41b3553b952fc11d51a0087f32eb27f6cd0d988
02/10/2016	Updated unit tests	60f5135984e32922339fde0e2a19ee82588b3480
03/10/2016	Updated string format	f0ad52f386361579af47989994fbe6fc999199b8
03/10/2016	Updated character device type methods	036edfab56b3925c6276077f1c21f22c89d40ac9
03/10/2016	Updated Device types for use with Strings	013abf4d2502cb9d5fb982c029b6efa59a107b45
03/10/2016	Added all assignment required files	27fd5b55e28af14643769d1dc9a92b7eb8de3a02
03/10/2016	Added all assignment required files	31c57792949e541e05404ef2769ac86017453924
09/10/2016	Prior to changes of FloatingPoint	559b5d68f323bc3eda2f6dcb620c38021bc71171
09/10/2016	Fixed FP	e4fbf5d671aa484b062a05e57364e1beb39cc5f4
09/10/2016	Pre network redesign	d116ce837a9db0d8768403779432bb8cd1cdc447
09/10/2016	refactored in generic queues	0abc050a902b2a61d6a96d13ff83cd45b7ed5902
09/10/2016	Added contains methods and proper unit tests	76a4b86c6e400bd7251ab6d90dae0d989cd4f052
09/10/2016	small changes	f19be42370cf4bd97bd02313c9e6760c993a0f02
09/10/2016	Fixed up some comment block formatting	dc8e48e849b824ce840352fae6a1310b0a78dad8
10/10/2016	Added file writer for sorted file	a730edff82f62c8080e0ff0661ba33956771f344
10/10/2016	Added directed variables	303712cf48367ef85e80ed221c27be2f25003d0d
12/10/2016	added notes	e632341871904f1875eb8a6b4a88a63c81109e2b
14/10/2016	Updated codebase with new files from lecturers	8a2f8443a2f23a2a2d0139dee35c89fd07be082e
15/10/2016	Added adjacent methods	7eb39d224484e1d6e20715233b540742812b1816
15/10/2016	Completed Qc and Qn kind of	84161ee971c6ba52f744abd2db57a84904747dd1
16/10/2016	I think sorting is working on vertices	35bffdf021eea2dfc957f540b508d4e9ff47f0a7

Date	Description	Commit
16/10/2016	I think sorting is working on vertices	9fd6a95c157580527eb4f3962245ee0bb6e4847c
19/10/2016	Added some notes	03e52de184fe69ffc5d3d6b24ab31fc7b978c019
19/10/2016	Added some notes	5ee3d5dbc3f20079268da6b9c202f8c086c14049
19/10/2016	Added updated code3	bd9714b37f9632089f7548cc3149bdc2758d21d5
19/10/2016	Vertex sorting works	ffd6518ec1f4fcbf5c8c052ebc0df6e50301926a
20/10/2016	BFS works. commit before i break it	c639a37c522d61793550b996420154c90e0271ea
20/10/2016	Updated sample files and includes logic for direction	46132b4bc42a72da4b7d54936f1b0abdbc0a5806
20/10/2016	Cleaned up file structure	dc1300533544e6c15a34b85ade1b4545aa6d1992
21/10/2016	Added headers	4a22be2d738d798542d8c0b2b9390dfd46a371b2
21/10/2016	Began updating javadocs	333ad2c9d867822683c4d147e3dad80a392e4e72
21/10/2016	Updated report	661f8661e50ec1524fe2cb3928b7210c971d41b2
21/10/2016	Added UML	1bbeeaa8312b9fa8c5dced8f3cc649016e557c87
22/10/2016	Updated report and worked on javadoc entries	4081dbbee4a092c8f8440813f1555f62c3648d16
23/10/2016	Finished the majority of javadocs for my classes	b8bf0cb5d14da9f9729033b0bcc220ad4777c7fb
23/10/2016	Added supplied v1.4 code and confirmed functionality	c3388cd01d9965a1863468bd8415f4ecb3d01445
23/10/2016	Updated UML	3005f49cb8b34ab60deb8388c258401e0e9f0ff7
23/10/2016	Updated report and ControllerUnitTest	32757a02682586af7ff76f595f9546a098f0f2f0
24/10/2016	Updated report Qn and Qc discussion	87779d443501614aa4dcb7ee657d8b2d8570b0f9
25/10/2016	Commit prior to BST integration	63232b84acc1e967e85064a7db1898146a453791
25/10/2016	Updated Qn to use BST, also updated report	47d70826d3e49598b012cda30b7d71ddb973841c
26/10/2016	Patched in v1.5 code and confirmed it all works	69cce82dc1e0a3f840e5bc1fe56f8b95ad760dcb
27/10/2016	Finished javadocs for BST	49e83a968f70cf117e4661f8cfb12f9a44c4108e
27/10/2016	File submission structure built	f7a044da104e0e50c14f5360973223f46d6234be