## Summary

The goal of this team project is to build a simulator to evaluate the performance of various routing algorithms. This project will implement the following routing algorithms:

1. Random
2. Flooding
3. Shortest path
4. Custom method

A user interface will be implemented for the simulator to display the network topology and let the user edit it. The interface will always provide the metrics of the network and the ability to view and step through a simulation.

## Milestones

The project is divided into 4 iterations, each of which correspond to a milestone with deliverables.

|  |  |  |  |
| --- | --- | --- | --- |
| Milestone | Description | Due Date | Weight |
| 1 | A simple console-based version. Only the random algorithm will be implemented. | Oct 21 | 15% |
| 2 | GUI and unit tests. | Nov 7 | 20% |
| 3 | All routing algorithms are to be implemented and selectable through the GUI. | Nov 21 | 30% |
| 4 | Add the ability to step back in a network (i.e. undo function) and the ability to save/restore a simulation of the network topology. | Dec 5 | 35% |

## Design Choices

The design choices our team made are logged below for each milestone:

**Milestone 4:**

The core new functions added in this milestone are **“Step Back”**, **“Save state”**, and **“Import/Export”**. The **“Step Back”** and **“Save state”** functionality were built together allowing the state of the network to be saved after each step (including metrics, topology, and messages). The **“Step Back”** function was implemented to traverse through the recorded states (State class created to hold the structure of the network state) of the network allowing the user to step forward and backward through the execution of the network. To implement the **“Import/Export”** functionality of the network, the **JAX** standard was used to marshal and unmarshal the network. When exporting a network, the **“Save state”** function is used to contain the topology of the network (along with Frequency and Algorithm, if set), which is then exported to an **XML** document OR imported as a new network. Also when abstracting and refactoring our code we decided to entirely remove the **Edge** class and replace it with an array list of neighbors in each node.

**Assumptions:**

* When exporting, it is assumed that only the topology of the network needs to be exported (simply the nodes and the connections). The metrics and messages are all reset for each new network / imported network.

**Milestone 3:**

As shown in the UML diagrams below, significant changes and additions were made to the overall project design since milestone two. Several design ‘*smells’* were removed from the code base. To accomplish this, we implemented the **MVC** pattern. We did a complete overhaul of our View class and have decoupled our Models from it completely by using the **Observer/Observable** pattern. To connect the two, a **Controller** class was implemented to handle and delegate all actions. The design choices that went in to the layout, look and fell, functionality of the GUI were revamped to look aesthetically pleasing and interactive**.** For example, a user can highlight two nodes and choose to connect or delete an edge or node with a click of a button.

A reset functionality has been implemented which allows a user to reset the simulation but preserve the graph originally created.

In anticipation of the fourth milestone, we have also created a **State** class which is used in the controller to help interact between the models and the view and store the current state of the network.

The average hops metric is available in real time to the user. The metric is calculated by grouping all like messages based on same source and destination and taking an average of their total hop counts. This data is portrayed when clicking the Average Hops button any time during the simulation in a JTable.

A **Constants** class was also created that stores final Strings used throughout. A **METHOD** enums class was also created that stores all possible action methods in the controller. The **ShortestPathAlgorithm** contains the class Path that was created to store the shortest calculated path between a given source and destination. When **ShortestPathAlgorithm** is first initialized it uses the Path class to create shortest paths for ALL source/destination pairs. The **CustomAlgorithm** combines logic from both **RandomAlgorithm** and **FloodingAlgorithm** to have follow the following behavior. The **CustomAlgorithm** sends each message in the **currentMessageQueue** of the network to ONE of its random neighbors, as long as that neighbor is not the message’s previous location.

**Assumptions:**

* No assumptions were made this milestone.

**Milestone 2:**

As shown in the UML diagrams below, some changes and additions were made to the overall project design. The algorithms were refactored into individual class that all share behavior implemented by the *Abstract Class* **Graph**. Any method implemented by the “Milestone 1 **Graph**” that is not done by the algorithms, is now implemented in the **Simulation** to allow the behavior to still exist and behave the way it should. This allowed us to remove a lot of potential ‘*smells’* from the code when multiple algorithms are implemented. Some small adaptations were made to the **Message** class to decouple it from the **Simulation**, allowing it to keep track of the current node it has hopped to, and to keep track of the total number of hops it completes until reaching its destination. The **View** was created to provide users with a simple, and visually pleasing interface to interact with the simulation. The design choices that went in to the layout and functionality of the GUI was loosely based off of the layout of the IDE [**BlueJ**](http://www.bluej.org/)**.** Our simulation offers the option to use a predefined or “*Default*” network that creates the example network shown in the Project Requirements. This “*Default Network”* allows the user a quick and simple way to initialize a full network that will produce ample metrics to calculate efficiency.

**Assumptions:**

* The metrics should be displayed/updated every time the simulation ‘steps’ forward.
* The RANDOM algorithm chooses a random neighbor to forward each message at each hop.
* The frequency is the number of hops a message must complete before a new message is created and injected into the network
* The user cannot edit the graph once the simulation has started
* The routing tables or the path taken by each message is not shown to the user
* The test case classes do not require UML diagrams

**Milestone 1:**

For implementing our four routing algorithms, we decided to use the strategy pattern by creating a **RoutingAlgorithm** interface. At this point the **RoutingAlgorithm** interface has all the code for the how each algorithm should run, we will be changing this in Milestone 2. **ALGORITHM** enum class was created to be able to call a specific algorithm.

A **Message** class was created to store a random source and destination of a message being sent through the network.

A class was also created for **Graph** which includes a set of nodes and list of edges also modeled as classes (**Nodes** and **Edge**)

A **Simulation** class was created to simulate the project generating a **Graph** and displaying progress and metrics to the user using a **Controller UI** class

**Assumptions:**

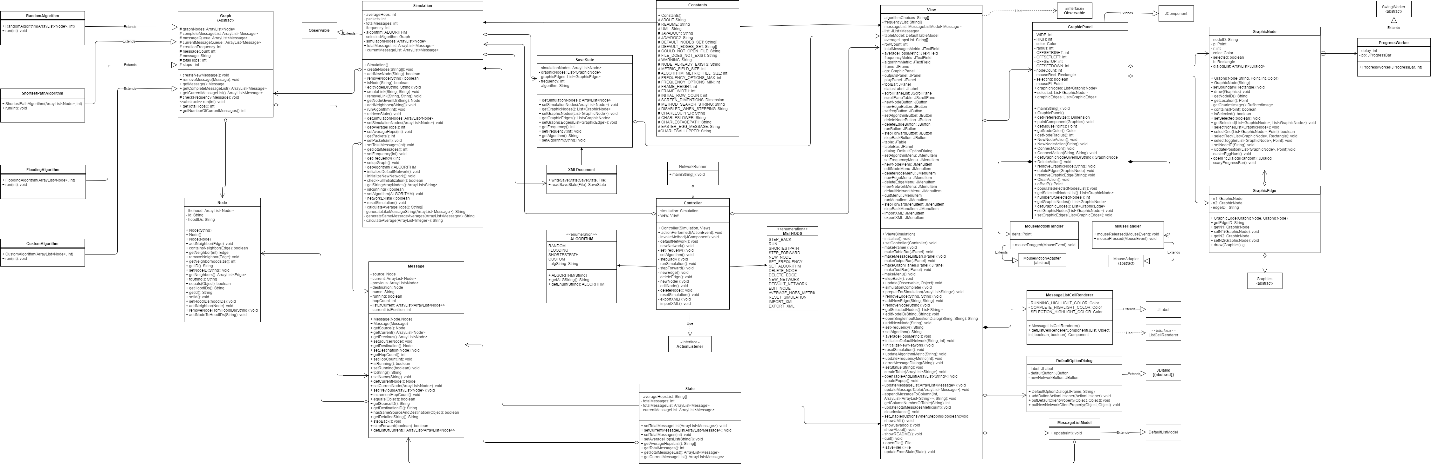
* The user should not be able to step through the simulation (for Milestone 1)
* The metrics are therefore printed at the end of the simulation
* The RANDOM algorithm chooses a random neighbor to forward each message at each hop.
* The frequency is the number of hops each message takes before a new message is injected into the network
* The user cannot edit the graph once it has been created (Milestone 1)
* There are no intermediate metrics displayed between hops- only at the end
* The routing tables or the path taken by each message is not shown to the user
* There is a chance of the simulation running infinitely on a large graph and low frequency (functions as designed)

## UML Diagram

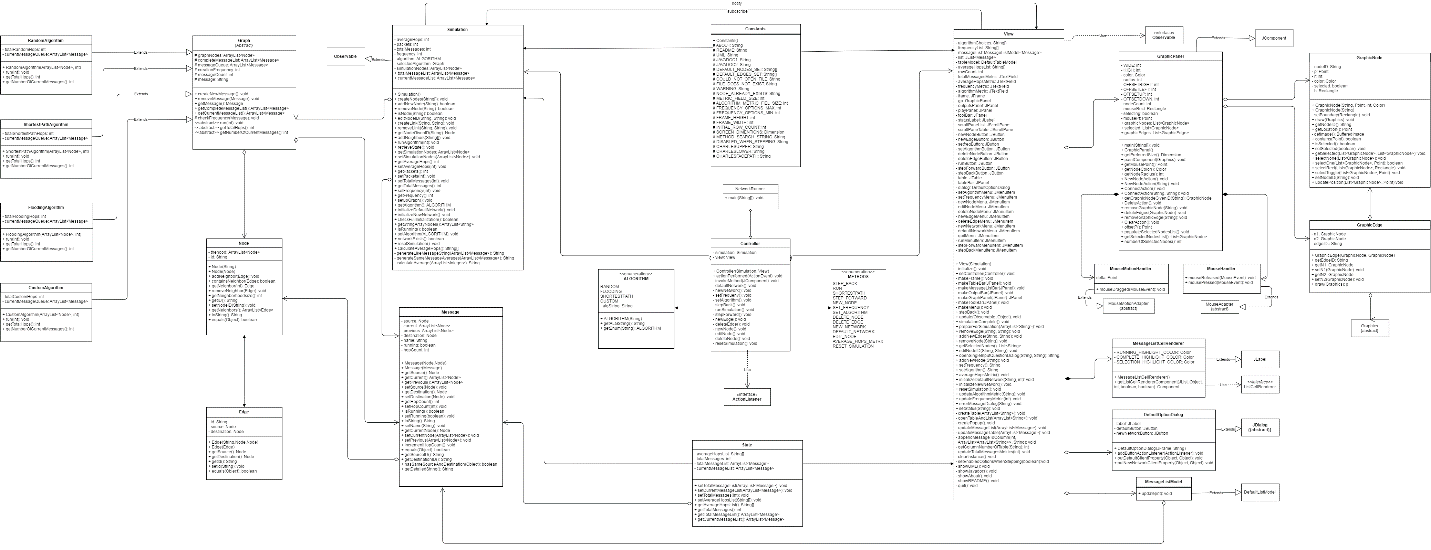
UML diagrams of the network for each iteration are shown below:

Please refer to the PDF of the UML for a clearer version.

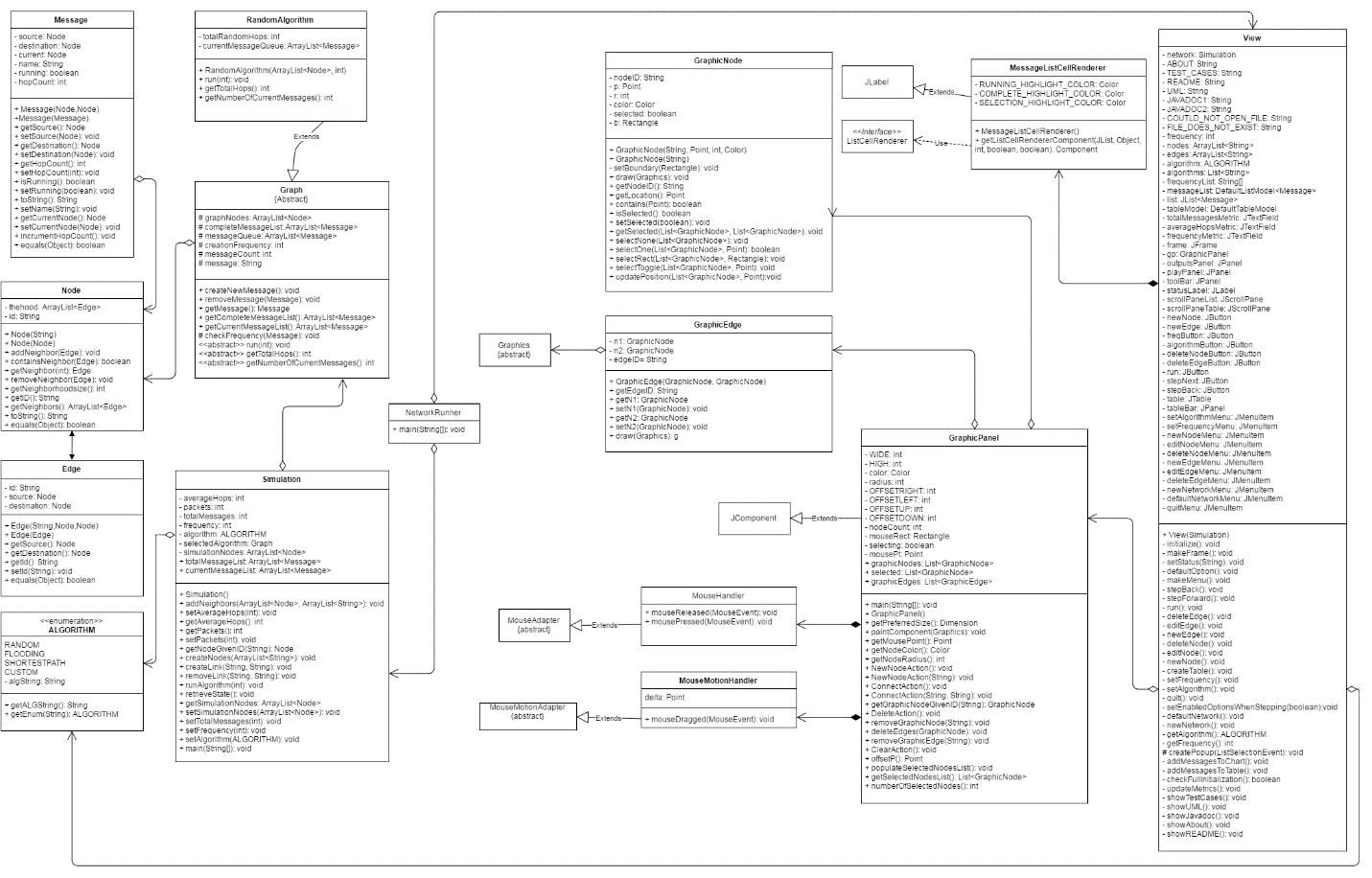
**Milestone 4:**



**Milestone 3:**



**Milestone 2:**



**Milestone 1:**

https://cloud.githubusercontent.com/assets/14824913/19614758/411b531e-97c6-11e6-9c2b-2970fc85ec8c.png (Please Note: Sequence diagrams were not needed for Milestone 1)

## Developer Contributions

Contributions to project for each developer are listed below for each milestone.

**Milestone 4:**

|  |  |
| --- | --- |
| Developer | Contributions |
| Alex Hoecht | Abstracted algorithms, updated documentation |
| Andrew Ward | Built import and export functions for topology |
| Mohamed Dahrouj | Built import and export functions for topology |
| Shasthra Ranasinghe | Built in step back functionality and polished Gui |
| Everyone | Updated diagrams and documentation |

**Milestone 3:**

|  |  |
| --- | --- |
| Developer | Contributions |
| Alex Hoecht | Implemented all algorithms and tweaked backend |
| Andrew Ward | Updated and added Unit Tests |
| Mohamed Dahrouj | Implemented the second metric and created UML |
| Shasthra Ranasinghe | Tailored the GUI and implemented the MVC Pattern |
| Everyone | Created block comments for JavaDocs, refactored/ removed smells from code |

**Milestone 2:**

|  |  |
| --- | --- |
| Developer | Contributions |
| Alex Hoecht | Implemented and re-factored back end models |
| Andrew Ward | Created Unit Tests |
| Mohamed Dahrouj | Implemented Visual Graph |
| Shasthra Ranasinghe | Created GUI |
| Everyone | Created block comments for JavaDoc, Created UML Diagram, and integrated all code |

**Milestone 1:**

|  |  |
| --- | --- |
| Developer | Contributions |
| Alex Hoecht | Implemented the random routing algorithm and layed groundwork for other algorithms |
| Andrew Ward | Implemented the Graph class consisting of Node and Edge objects |
| Mohamed Dahrouj | Implemented a Simulation class that creates a simulation and stores metrics |
| Shasthra Ranasinghe | Implemented a console UI to interact with users and Simulation |
| Everyone | Created block comments for JavaDoc, Created UML Diagram, and integrated all code |

**Note to Marker:**

* At our first meeting we decided on a very basic idea of how the simulation should run, with many assumptions and a lot of the functionality missing. The tasks were split equally and a branch was created for each member where we worked on our section of the code
* Once we all had each of sections ready, we collectively decided at our second meeting that it was better for the first milestone if we manually place all the code we have into the master branch and delete our individual branches. We then branched from master in order to have at least a semi-functional program to work with and each member knows how the design was implemented in code
* We still had trouble merging code. To work around this we decided to manually copy the most up-to-date code into M1\_Alex branch and work on that together either from our own computers or on the same computer
* For the first milestone, the total packets metric was implemented
* The JavaDocs are included for all public members in the **doc** folder

## How to Run

The steps required to run the project are outlined below for each milestone:

**Milestone 4:**

Please refer to the steps below in Milestone 2.

GUI specific instructions have been updated in a .txt file that is accessible through the GUI under the help menu.

**Milestone 3:**

Please refer to the steps below in Milestone 2.

GUI specific instructions have been updated in a .txt file that is accessible through the GUI under the help menu.

**Milestone 2:**

To run the project, please create a Java Application under "Run Configurations" using the main() method provided in the **NetworkRunner** class. On the other hand, please refer to CuLearn for the .jar file including source code. For running the jar file, please open a command prompt, and use the command **java -jar milestone2.jar**

**Steps:**

1. Interact with GUI to properly create a network that will be simulated.
2. Either use ‘Run’ (run the simulation until termination) OR use ‘Next’ (‘Hop’ to the next state in the simulation).
3. Use ‘New Network’ start a new network OR ‘Quit’ to exit the program.

**Milestone 1:**

To run the project, please create a Java Application under "Run Configurations" using the main() method provided in the Simulation class. On the other hand, please refer to CuLearn for the .jar file including source code. For running the jar file, please open the cmd prompt, and type **\*\*java -jar milestone1.jar\*\***.

**Steps:**

1. Enter all the names of the nodes in the graph with spaces separating them

2. Enter the link between nodes by following the instructions on the console. Provide one link at a time and press Enter

3. Enter the frequency (number of hops each message takes before a new message in injected into the network)

4. The simulation might run infinitely if the frequency is low, otherwise the metrics will be printed out and simulation will end.

Last updated: November 21, 2016