[Project 3]

Final Report

**<Theodore Reger, Devin Janus, Samuel Jessee>**

**CSC316: Data Structures for Computer Scientists**

**<tlreger@ncsu.edu, dwjanus@ncsu.edu, sijessee@ncsu.edu>**

**North Carolina State University**

**Department of Computer Science**

**<April 4, 2016>**

Executive Summary

This is a proposal for a project that creates a graph using text file input, and finds the minimal spanning tree of the graph based on the data received. The final product will output the edges of the graph, the minimal spanning tree of the graph and the adjacent vertices of every vertex in the graph. In the problem statement section I present an explanation of the need for this program, and how it could be used by the sponsor. In the Project Goals and Benefits section I present the developmental forecasts of the project and why the sponsor will find such software useful. In the Algorithm Design I propose algorithms for use in the project, I explain my reasoning for the data structures that I use in the project, and I analyze how the project would behave using my proposed algorithms. In the software design section I present a UML Diagram as a plan for how the proposed software will be implemented and built. In the Black Box Test Plan section I present the future methods that will be used to hand test this project and its graphical user interface. The Task Plan section shows a day to day schedule of how work on the project will progress. It also includes important dates for development, the contact information of the members of the team and who will be completing what sections of the project. This Task Plan is largely subject to change due to the scholarly obligations of my partner and myself. Even who is doing what parts of the project may be subject to change, as we have not yet discussed our plans for attacking the development of this software.

Table of Contents

Add a table of contents here. Word processing software usually has a feature to make generating tables of contents simple and easy. Check the documentation for your specific piece of software.

For example, in Word, you should be able to right-click the table of contents below and select “Update field” -- the contents are pulled from the headers used through the rest of the document.

[Problem Statement 1](#_Toc447474339)

[Project goals & Benefits 2](#_Toc447474340)

[Algorithm Design 3](#_Toc447474341)

[Proposed algorithm 3](#_Toc447474342)

[Algorithm Design Rationale 6](#_Toc447474343)

[Algorithm Analysis 6](#_Toc447474344)

[Algorithm Changes 10](#_Toc447474345)

[Software Design 11](#_Toc447474346)

[Design Pattern(s) 11](#_Toc447474347)

[Design Changes 11](#_Toc447474348)

[Black Box Test Plan 13](#_Toc447474349)

[Black Box Test Plan Changes 13](#_Toc447474350)

[Task Plan 18](#_Toc447474351)

[Task Plan Changes 18](#_Toc447474352)

[Discussion 19](#_Toc447474353)

[Challenges 19](#_Toc447474354)

[Threats to Validity 19](#_Toc447474355)

[Reflection 19](#_Toc447474356)

[Conclusion 20](#_Toc447474357)

[Conclusions 21](#_Toc447474358)

# Problem Statement

The problem for this project is to develop a software that will create a minimum spanning tree amongst all the given vertices. This is represented by the idea of some millionaires wanting a software that determines the cheapest way to build bridges that connect all of their islands. If the millionaires own hundreds of islands, it will take forever for an employee of theirs to determine the best way to build the bridges. The contractor will have to wait for the calculation to be finished, and the employee’s calculations will be subject to human error. Waiting, verifying calculations and recalculating all waste time, time that could be spent making bridges for other islands.

# Project goals & Benefits

I will deliver a software to the sponsor that calculates the best way to build bridges between an indefinite number of islands. This is better than the current system, as it will be much faster for a computer to figure this out than an employee. The sponsor will be able to tell the building contractor the plans for construction immediately after receiving the cost projections, and the results will not be subject to human error. The time spent prepping and preparing for construction will be drastically cut down.

# Algorithm Design

## Proposed algorithm

------------------------------

The heap structure, with the insert and deleteMin operations; use an array-based implementation.

**Algorithm** insert(edge e, array H)

**Input** a key *k* for an element *x* to insert to an array *H* with n elements

H[size(H)] 🡨 e 3

size(H) 🡨 size(H)+1 2

upHeap(H, size(H)-1) 2\* upHeap

T(V) = 3 + 2 + 2 \* 9 \* log( V ( V – 1 ) / 2 )

**Algorithm** upHeap(array *H*, integer *i*)

**Input** the position of the new element in the heap (i), and an array *H* with n elements

if i > 0 then 1

if key(H[(i-1)/2]) > key(H[i]) then 7

swap((i-1)/2,i) 1

upHeap(H, (i-1)/2)

**Worst case scenario: upheap is called height number of times. E represents the number of edges in the heap array, V represents the number of vertices.**

Height = log( E ) = log ( V ( V-1) / 2 ),

T(V) = 9 \* log( V ( V – 1 ) / 2 )

**Algorithm** deleteMin(array *H*)

**Input** an array *H* with n elements

**Output** the minimum element in the array

x 🡨 element(H[0]) 3

size(H) 🡨 size(H) – 1 3

swap(0, size(H)) 1

downHeap(H,0) downHeap

return x 1

T(V) = 8 + 21 \* log( V ( V – 1 ) / 2 )

**Algorithm** downHeap(array *H,* integer *m*)

**Input** an array *H* with *n* elements, position *m in the array*

i 🡨 0 1

if 2m + 2 < size(H) then 4

if key(H[2m+2] key(H[2m+1]) then 7

i 🡨 2m + 2

else 1

i 🡨 2m + 1 1

else if 2m + 1 < size(H) then

i 🡨 2m + 1

if i > 0 and key(H[m]) > key(H[i]) then 6

swap(m,i) 1

downHeap(H,i)

**Worst case scenario: downheap is called height number of times. E represents the number of edges in the heap array, V represents the number of vertices.**

Height = log( E ) = log ( V ( V-1) / 2 ),

T(V) = 21 \* log( V ( V – 1 ) / 2 )

The up-tree structure, with the union and find operations; use an array-based implementation.

**Algorithm** union(vertex S, vertex T)

**Input** a root vertex S and a root vertex T

**Output** the root vertex of the combined set

if count(S) >= count(T) then 3

count(S) 🡨 count(S) + count(T)

parent(T) 🡨 S

return S

else 1

count(T) 🡨 count(S) + count(T) 2

parent(S) 🡨 T 2

return T 1

**V represents the number of vertices**

T (V) = 9

**Algorithm** find(vertex p)

**Input** a vertex p

**Output** the root vertex of the set containing p

while parent(p) null do Height + 1

p 🡨 parent(p) Height

return p 1

**Height represents the height of the tree. V represents the number of vertices in tree**

T(V) = 2 \* Height + 2 = 2 \* log(V) + 2

An adjacency list representation of graphs.

Kruskal's algorithm for computing the MST of a graph.

**Algorithm** KruskalMST(graph G)

**Input** a graph G

**Output** a set of edges representing a MST

edgeSet 🡨 null 1

H 🡨 empty heap 1

While G has edges e do E+1 = V(V-1)/2 + 1

insert(e, H) E \* insert = V(V – 1)/2 \* insert

components 🡨 |V| 2

while G has vertices v do V + 1

makeSet(v) V

while components > 1 do V

edge 🡨deleteMin(H) (V – 1) \* deleteMin

U 🡨 find(Node1(edge)); ( V -1) \* (2 + find)

V 🡨 find(Node2(edge)); (V-1) \* ( 2 + find)

if U != V then V - 1

union(U,V) (V – 1) \* union

edgeSet 🡨 edgeSet ∪ edge V – 1

components 🡨 components – 1 2 \* (V – 1)

return edgeSet 1

**V represents the number of vertices in the graph. E represents the number of edges.**

------------------------------

## Algorithm Design Rationale

This project will be using several different data structures, including: graphs, heaps, matrices and edges. The Graph object will act as the go between for the Manager and the other data structures. It will be able to manipulate data structures that it is composed of. The heap will be used to store all edges of the graph. A matrix will be used to keep track of vertex adjacency. Edges will be used to link vertices inside a graph. Each edge will also carry a weight value which will be used for determining the minimum spanning tree.

## Algorithm Analysis

----------------------------

The heap structure, with the insert and deleteMin operations; use an array-based implementation.

**Algorithm** insert(edge e, array H)

**Input** a key *k* for an element *x* to insert to an array *H* with n elements

H[size(H)] 🡨 e 3

size(H) 🡨 size(H)+1 2

upHeap(H, size(H)-1) 2\* upHeap

T(V) = 3 + 2 + 2 \* 9 \* log( V ( V – 1 ) / 2 )

**Algorithm** upHeap(array *H*, integer *i*)

**Input** the position of the new element in the heap (i), and an array *H* with n elements

if i > 0 then 1

if key(H[(i-1)/2]) > key(H[i]) then 7

swap((i-1)/2,i) 1

upHeap(H, (i-1)/2)

**Worst case scenario: upheap is called height number of times. E represents the number of edges in the heap array, V represents the number of vertices.**

Height = log( E ) = log ( V ( V-1) / 2 ),

T(V) = 9 \* log( V ( V – 1 ) / 2 )

**Algorithm** deleteMin(array *H*)

**Input** an array *H* with n elements

**Output** the minimum element in the array

x 🡨 element(H[0]) 3

size(H) 🡨 size(H) – 1 3

swap(0, size(H)) 1

downHeap(H,0) downHeap

return x 1

T(V) = 8 + 21 \* log( V ( V – 1 ) / 2 )

**Algorithm** downHeap(array *H,* integer *m*)

**Input** an array *H* with *n* elements, position *m in the array*

i 🡨 0 1

if 2m + 2 < size(H) then 4

if key(H[2m+2] key(H[2m+1]) then 7

i 🡨 2m + 2

else 1

i 🡨 2m + 1 1

else if 2m + 1 < size(H) then

i 🡨 2m + 1

if i > 0 and key(H[m]) > key(H[i]) then 6

swap(m,i) 1

downHeap(H,i)

**Worst case scenario: downheap is called height number of times. E represents the number of edges in the heap array, V represents the number of vertices.**

Height = log( E ) = log ( V ( V-1) / 2 ),

T(V) = 21 \* log( V ( V – 1 ) / 2 )

The up-tree structure, with the union and find operations; use an array-based implementation.

**Algorithm** union(vertex S, vertex T)

**Input** a root vertex S and a root vertex T

**Output** the root vertex of the combined set

if count(S) >= count(T) then 3

count(S) 🡨 count(S) + count(T)

parent(T) 🡨 S

return S

else 1

count(T) 🡨 count(S) + count(T) 2

parent(S) 🡨 T 2

return T 1

**V represents the number of vertices**

T (V) = 9

**Algorithm** find(vertex p)

**Input** a vertex p

**Output** the root vertex of the set containing p

while parent(p) null do Height + 1

p 🡨 parent(p) Height

return p 1

**Height represents the height of the tree. V represents the number of vertices in tree**

T(V) = 2 \* Height + 2 = 2 \* log(V) + 2

An adjacency list representation of graphs.

Kruskal's algorithm for computing the MST of a graph.

**Algorithm** KruskalMST(graph G)

**Input** a graph G

**Output** a set of edges representing a MST

edgeSet 🡨 null 1

H 🡨 empty heap 1

While G has edges e do E+1 = V(V-1)/2 + 1

insert(e, H) E \* insert = V(V – 1)/2 \* insert

components 🡨 |V| 2

while G has vertices v do V + 1

makeSet(v) V

while components > 1 do V

edge 🡨deleteMin(H) (V – 1) \* deleteMin

U 🡨 find(Node1(edge)); ( V -1) \* (2 + find)

V 🡨 find(Node2(edge)); (V-1) \* ( 2 + find)

if U != V then V - 1

union(U,V) (V – 1) \* union

edgeSet 🡨 edgeSet ∪ edge V – 1

components 🡨 components – 1 2 \* (V – 1)

return edgeSet 1

**V represents the number of vertices in the graph. E represents the number of edges.**

T(V) = 2 + V(V-1)/2 + 1 + V(V – 1)/2 \* insert + 2 V + 1 + V + V + (V – 1) \* deleteMin + 2 \*( V -1) \* (2 + find) + V – 1 + (V – 1) \* union + 3 \* (V – 1) + 1

= ½\*V^2 – V/2 + (½\*V^2 – V/2) \* insert + 8 \* V + (V – 1) \* deleteMin + 2 \*( V -1) \* (2 + find) + (V – 1) \* union + 1

= ½\*V^2 – V/2 + (½\*V^2 – V/2) \* ( 5 + 18 \* log( ½\*V^2 – V/2 ) ) + V \* (8 + 21 \* log(½\*V^2 – V/2 ) – 8 – 21\* log( ½\*V^2 – V/2) + ( 2\*V – 2)\* (4 + 2 \* log(V) ) + 9\*V – 8

= ½\*V^2 – V/2 + (5/2\*V^2 – 5/2\*V + 9\* log( ½\*V^2 – V/2 )\* v^2 – 9 \* log( ½\*V^2 – V/2 )\* V + 8\*V + 21\*V \* log(½\*V^2 – V/2 ) – 8 - 21 \* log( ½\*V^2 – V/2) + 8\*V + 4\*V\*log(V) – 8 – 4\*log(V) + 9\*V – 8

= ½\*V^2 + 5/2\*V^4 – 5/2\*V^3 + 9\*V^2\*log( ½\*V^2 – V/2 ) + 12 \*V\* log( ½\*V^2 – V/2 ) + 24.5\*V – 24 – 21 \* log( ½\*V^2 – V/2) + 4\*V\*log(V) – 4\*log(V)

T(V) / v^4 = 1/2/(V^2) – 1/2/(v^3) + 5/2 – 5/2/(V) + 9\* log( ½\*V^2 – V/2 ) / (V^2) + 12/(V^3) \*log( ½\*V^2 – V/2 ) + 24.5/(V^3) – 24/(v^4) – 21 / (v^4) \* log( ½\*V^2 – V/2 ) + 4/(V^3)\*log(V) – 4/(V^4)\*log(V)

Lim (V🡪infinity)(T(V)/V^4) = 5/2

T(V) is O( V^4 )

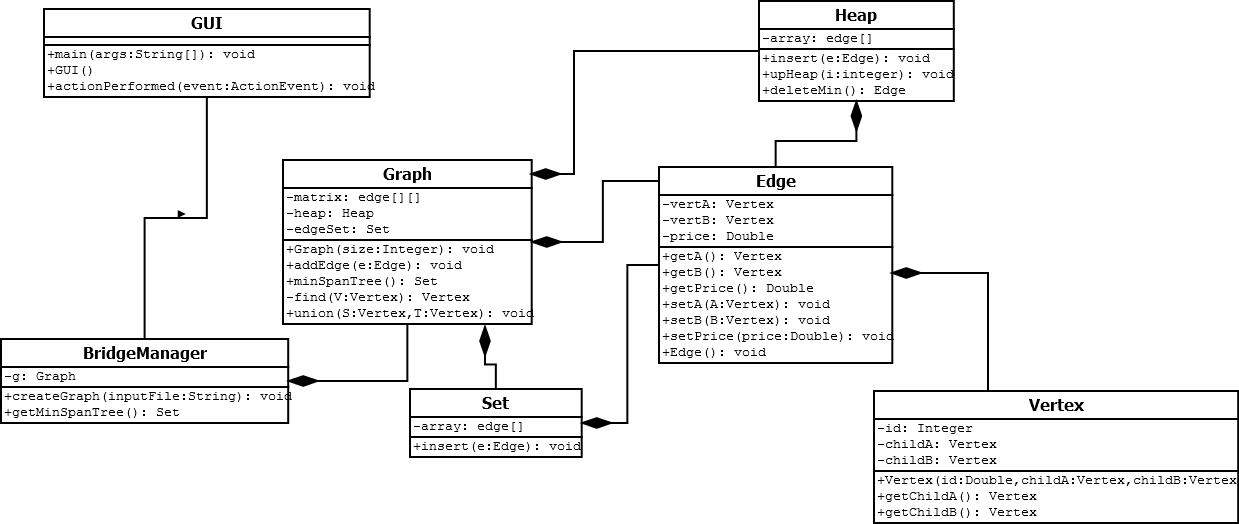
---------------------------

## Algorithm Changes

While implementing Kruskal’s algorithm into the project, we found that creating a component class would be unnecessarily burdensome. Having one component essentially meant that all vertices were in a single uptree. We had already implemented our uptree so that a head node would store the number of nodes in its uptree as its data. We decided to take advantage of this, and made it so that the while loop continued while the size of the uptree was less than the number of vertices in the graph. The resulting algorithm had the same effect. In order for this to work correctly, we had to traverse through all the graph’s bridges and create a list of vertices that did not repeat any. We used this list to determine how many vertices were in the graph. We also had to sort the list of bridges returned by kruskal’s algorithm in order to match the expected string output.

# Software Design

---------------------------



-------------------------

## Design Pattern(s)

I am incorporating the Model-View-Controller design into the software design of my Minimum Bridge Manager project. This design is effective for this project because it organizes the classes into the interface, or view, the manager, or controller, and the classes that perform calculations, or model. This organization makes the programmer write specific types of functions for the correct classes, allowing for code to be easily findable. This pattern, however, does not ensure that the classes inside the model, view or controller will use functions relevant to the purpose of the class. Despite this limitation, using this pattern will help keep classes organized.

I am also incorporating the Composition design pattern into the software design of this project. Composition is using declaring an object as a field of class. As a result, some data of an object becomes accessible to the class the object was declared in, and the object has the same lifespan as the class itself. The purpose of using this design pattern is to allow a single instance of an object to be used in multiple methods of an object. Without this design pattern, an instance of an object could not be used in more than one place. The design is limited in that if the field storing the instance of the object is overwritten, the previous instance is lost. This should not be an issue, however, since no classes in the software design are static.

## Design Changes

In our final design, we incorporated several changes. We did not have a vertex class, as vertices did not need to reference other vertices. Instead, vertices were simply the integer values representing their island. The vertices referencing other vertices function was replaced with an uptree class which kept track of whether a vertex had been connected to another or not when finding the minimum spanning tree. The set class was replaced by the ArrayBasedList class, an array list which accepted any object as a parameter. Instead of using an adjacency matrix to keep track of vertex adjacency, we used an adjacency list. The adjacency list was an ArrayBasedList of bridges, and the bridges acted as a linked list, with the bridge in the list pointing to all other bridges that it is adjacent to. This change was made in order to follow the requirements of the project.

# Black Box Test Plan

## Black Box Test Plan Changes

Very little changed about the black box tests except the output format. The original output was changed for part 2 of the project, so we had to alter our program to match. As a result, we also had to alter the black box tests to match the program. The input format remained the same, and all else remained the same aside from a few interface changes. The interface changes in the black box test plan occurred because the actual GUI was implemented slightly differently from the original plan.

---------------------

|  |  |  |  |
| --- | --- | --- | --- |
| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
| **Two Islands** | **Preconditions:**  **Create a text file matching the following, without quotes:**  **“1 0 7.0” Name the file “two\_isl.txt” without the quotes. Make sure that the file is in the same folder as the Minimum Bridge Manager program.**  **In your eclipse window, right click on the Minimum Bridge Manager project. Select “run as” then “java application.” The program should open to an input text box and a button.**  **Test Description:**  **In the input text box, put “two\_isl.txt” without the quotes. Press the find minimum bridges button.** | **Heap[Bridge[island1=1, island2=0, cost=7.0]]**  **AdjacencyList[**  **Island 0: -> Bridge[island1=1, island2=0, cost=7.0]**  **Island 1: -> Bridge[island1=1, island2=0, cost=7.0]**  **]**  **List[Bridge[island1=1, island2=0, cost=7.0]]** | **The expected output was met.** |
| **Four Islands** | **Create a text file matching the following, without quotes:**  **“2 0 7.0**  **3 2 12.0**  **0 3 14.0**  **1 0 5.0**  **3 1 10.0**  **1 2 6.0” Name the file “four\_isl.txt” without the quotes. Make sure that the file is in the same folder as the Minimum Bridge Manager program.**  **In your eclipse window, right click on the Minimum Bridge Manager project. Select “run as” then “java application.” The program should open to an input text box and a button.**  **Test Description:**  **In the input text box, put “four\_isl.txt” without the quotes. Press the find minimum bridges button.** | **Output should appear in the interface window. It should read:**  **Heap[Bridge[island1=1, island2=0, cost=5.0], Bridge[island1=2, island2=0, cost=7.0], Bridge[island1=1, island2=2, cost=6.0], Bridge[island1=3, island2=2, cost=12.0], Bridge[island1=3, island2=1, cost=10.0], Bridge[island1=0, island2=3, cost=14.0]]**  **AdjacencyList[**  **Island 0: -> Bridge[island1=0, island2=3, cost=14.0] -> Bridge[island1=1, island2=0, cost=5.0] -> Bridge[island1=2, island2=0, cost=7.0]**  **Island 1: -> Bridge[island1=1, island2=0, cost=5.0] -> Bridge[island1=1, island2=2, cost=6.0] -> Bridge[island1=3, island2=1, cost=10.0]**  **Island 2: -> Bridge[island1=1, island2=2, cost=6.0] -> Bridge[island1=2, island2=0, cost=7.0] -> Bridge[island1=3, island2=2, cost=12.0]**  **Island 3: -> Bridge[island1=0, island2=3, cost=14.0] -> Bridge[island1=3, island2=1, cost=10.0] -> Bridge[island1=3, island2=2, cost=12.0]**  **]**  **List[Bridge[island1=1, island2=0, cost=5.0], Bridge[island1=1, island2=2, cost=6.0], Bridge[island1=3, island2=1, cost=10.0]]** | **The expected output was met.** |
| **Common Weight** | **Create a text file matching the following, without quotes:**  **“2 0 7.0**  **3 2 7.0**  **0 3 7.0**  **1 0 7.0**  **3 1 7.0**  **1 2 7.0**  **3 2 12.0” Name the file “common.txt” without the quotes. Make sure that the file is in the same folder as the Minimum Bridge Manager program.**  **In your eclipse window, right click on the Minimum Bridge Manager project. Select “run as” then “java application.” The program should open to an input text box and a button.**  **Test Description:**  **In the input text box, put “common.txt” without the quotes. Press the find minimum bridges button.** | **Output should appear in the interface window. It should read:**  **Heap[Bridge[island1=2, island2=0, cost=7.0], Bridge[island1=3, island2=2, cost=7.0], Bridge[island1=0, island2=3, cost=7.0], Bridge[island1=1, island2=0, cost=7.0], Bridge[island1=3, island2=1, cost=7.0], Bridge[island1=1, island2=2, cost=7.0], Bridge[island1=3, island2=2, cost=12.0]]**  **AdjacencyList[**  **Island 0: -> Bridge[island1=0, island2=3, cost=7.0] -> Bridge[island1=1, island2=0, cost=7.0] -> Bridge[island1=2, island2=0, cost=7.0]**  **Island 1: -> Bridge[island1=1, island2=0, cost=7.0] -> Bridge[island1=1, island2=2, cost=7.0] -> Bridge[island1=3, island2=1, cost=7.0]**  **Island 2: -> Bridge[island1=1, island2=2, cost=7.0] -> Bridge[island1=2, island2=0, cost=7.0] -> Bridge[island1=3, island2=2, cost=7.0] -> Bridge[island1=3, island2=2, cost=12.0]**  **Island 3: -> Bridge[island1=0, island2=3, cost=7.0] -> Bridge[island1=3, island2=1, cost=7.0] -> Bridge[island1=3, island2=2, cost=7.0] -> Bridge[island1=3, island2=2, cost=12.0]**  **]**  **List[Bridge[island1=1, island2=2, cost=7.0], Bridge[island1=2, island2=0, cost=7.0], Bridge[island1=3, island2=2, cost=7.0]]** | **The expected output was met.** |
| **Incomplete Graph** | **Create a text file matching the following, without quotes:**  **“2 0 7.0**  **0 3 14.0**  **1 0 5.0**  **1 2 6.0”**  **Name the file “incomp.txt” without the quotes. Make sure that the file is in the same folder as the Minimum Bridge Manager program.**  **In your eclipse window, right click on the Minimum Bridge Manager project. Select “run as” then “java application.” The program should open to an input text box and a button.**  **Test Description:**  **In the input text box, put “incomp.txt” without the quotes. Press the find minimum bridges button.** | **Output should appear in the interface window. It should read:**  **Heap[Bridge[island1=1, island2=0, cost=5.0], Bridge[island1=1, island2=2, cost=6.0], Bridge[island1=2, island2=0, cost=7.0], Bridge[island1=0, island2=3, cost=14.0]]**  **AdjacencyList[**  **Island 0: -> Bridge[island1=0, island2=3, cost=14.0] -> Bridge[island1=1, island2=0, cost=5.0] -> Bridge[island1=2, island2=0, cost=7.0]**  **Island 1: -> Bridge[island1=1, island2=0, cost=5.0] -> Bridge[island1=1, island2=2, cost=6.0]**  **Island 2: -> Bridge[island1=1, island2=2, cost=6.0] -> Bridge[island1=2, island2=0, cost=7.0]**  **Island 3: -> Bridge[island1=0, island2=3, cost=14.0]**  **]**  **List[Bridge[island1=0, island2=3, cost=14.0], Bridge[island1=1, island2=0, cost=5.0], Bridge[island1=1, island2=2, cost=6.0]]** | **The expected output was met.** |
| **Repeated Weight** | **Create a text file matching the following, without quotes:**  **“2 0 8.0**  **3 2 7.0**  **0 3 8.0**  **1 0 6.0**  **3 1 7.0**  **1 2 6.0**  **3 2 7.0”**  **Name the file “repeat.txt” without the quotes. Make sure that the file is in the same folder as the Minimum Bridge Manager program.**  **In your eclipse window, right click on the Minimum Bridge Manager project. Select “run as” then “java application.” The program should open to an input text box and a button.**  **Test Description:**  **In the input text box, put “repeat.txt” without the quotes. Press the find minimum bridges button.** | **Output should appear in the interface window. It should read:**  **Heap[Bridge[island1=1, island2=0, cost=6.0], Bridge[island1=3, island2=2, cost=7.0], Bridge[island1=1, island2=2, cost=6.0], Bridge[island1=2, island2=0, cost=8.0], Bridge[island1=3, island2=1, cost=7.0], Bridge[island1=0, island2=3, cost=8.0], Bridge[island1=3, island2=2, cost=7.0]]**  **AdjacencyList[**  **Island 0: -> Bridge[island1=0, island2=3, cost=8.0] -> Bridge[island1=1, island2=0, cost=6.0] -> Bridge[island1=2, island2=0, cost=8.0]**  **Island 1: -> Bridge[island1=1, island2=0, cost=6.0] -> Bridge[island1=1, island2=2, cost=6.0] -> Bridge[island1=3, island2=1, cost=7.0]**  **Island 2: -> Bridge[island1=1, island2=2, cost=6.0] -> Bridge[island1=2, island2=0, cost=8.0] -> Bridge[island1=3, island2=2, cost=7.0] -> Bridge[island1=3, island2=2, cost=7.0]**  **Island 3: -> Bridge[island1=0, island2=3, cost=8.0] -> Bridge[island1=3, island2=1, cost=7.0] -> Bridge[island1=3, island2=2, cost=7.0] -> Bridge[island1=3, island2=2, cost=7.0]**  **]**  **List[Bridge[island1=1, island2=0, cost=6.0], Bridge[island1=1, island2=2, cost=6.0], Bridge[island1=3, island2=2, cost=7.0]]** | **The expected output was met.** |

# Task Plan

-------------------------

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Task Description** | **Owner** | **Planned**  **Start Date** | **Planned**  **End Date** | **Status** |
| **Write skeleton, including comments** | **sijessee** | **3/22/2016** | **3/23/2016** |  |
| **Write BridgeManager** | **sijessee** | **3/23/2016** | **3/24/2016** |  |
| **Write Bridge** | **sijessee** | **3/24/2016** | **3/25/2014** |  |
| **Write Heap** | **dwjanus** | **3/25/2016** | **3/26/2016** |  |
| **Write ArrayBasedList** | **dwjanus** | **3/26/2016** | **3/26/2017** |  |
| **Write UpTree, Kruskal’s** | **tlreger** | **3/27/2016** | **3/27/2016** |  |
| **Debug/White Box/Black Box Testing** | **tlreger**  **dwjanus**  **sijessee** | **3/27/2016** | **4/2/2016** |  |
| **Final Project Report** | **tlreger**  **dwjanus**  **sijessee** | **4/3/2016** | **4/4/2016** | **Due: 4/4/2016** |

**----------------------------**

**Team Contact Information**

Teddy Reger, Kruskal’s Developer, tlreger@ncsu.edu, 919-995-4694

Devin Janus, Heap Developer, dwjanus@ncsu.edu, 919-586-1684

Samuel Jessee, Bridge Developer/Javadoc Author, sijessee@ncsu.edu, 919-610-1546

## Task Plan Changes

The task plan had to be adjusted from its original form in order to accommodate three people rather than two. In addition, it had to be adjusted to provide for the creation of classes that had not existed in the original software design. In the end, our development process was quite collaborative in places, so it is difficult to specify exactly when and where every individual helped out in the development of the project. Ultimately, every project requirement was met on schedule.

# Discussion

## Challenges

One challenge we faced with this project was group communication. At first, group members worked on parts of the project without discussing with the group first, and used only Git commit messages to communicate with the group. As a result of the lack of direct communication, the group had to resolve multiple GitHub conflicts, and on several occasions, two group members would start working on the same thing, which wastes time since both members were creating the same code. To address this issue, our group started a chain of emails to communicate clearly who would be responsible for which tasks. Whenever a group member finished a task, or made a major change to the project, they would reply to the email chain and update the group so we could decide what to work on next. Once we began communicating more directly, it became much easier to complete the project efficiently, without Git conflicts or wasted time.

## Threats to Validity

Since many classes were developed by certain members independently of others, some functions were repeated unnecessarily in the code. For example, the adjacency list and minimum spanning tree both require bridges to be sorted into order by value of the islands, yet these sorting function were carried out separately. When dealing with more than five thousand islands, these sorts of inefficiencies can add up and drastically increase the run time. Similarly, little work was done in any of the classes to reduce overall runtime. We, the developers, put more emphasis on making the program work than improving runtime. This could potentially break the program if it were put under given sets of islands significantly larger than 5000, the limit of the project. One example of inefficiency, contained in the Kruskal’s algorithm, was that the number of vertices in the graph was determined by sorting through all the bridges, adding the vertices of the bridges to a list, and making sure that a vertex was not added multiple times. The total number of vertices in the graph could have been determined while the input was being read, but instead it was done with a big O of n^2 algorithm inside of Kruskal’s.

## Reflection

One thing that our group did really well on this project is time management. We began working on the project early enough that we had plenty of time to finish without having to cut any corners. As a whole, all of our group members worked very diligently to complete the project on time, so we did not have to rush, and no one had to pick up any extra slack for another group member.

What we struggled with, as discussed above, was communication. At first our group didn't communicate very well, which led to some confusion. We probably would have done a lot better by contacting each other from the beginning.

As far as the project itself, there wasn't much we could really have improved. Our program performs exactly like it is supposed to, and meets the parameters described in the problem, so there is not any significant improvement that could be made to the algorithms unless they could be written more efficiently to improve runtime.

## Conclusion

Our project is able to take input from a text file containing two vertices to a line and a price for constructing a bridge between those vertices. Using that info, it implements an ArrayBasedList, a Bridge, an Adjacency List, a Heap and an UpTree in order to determine a minimum spanning tree of bridges. In other words, it determines which bridges to build so that all vertices will be connected at the lowest price. This project makes it easier for the user to know which bridges to build so that the costs will be minimal. It does not require any user calculations, just input. This eliminates human error and speeds up the process of building.

# Conclusions

When implementing the project, we discovered that the creation of a set of bridges class was unnecessary. According to Kruskal’s Algorithm, the bridge of the lowest cost must be removed from the heap, with its vertices’ sets unioned, until there is only one set of vertices in the graph. This is unnecessary because the UpTree, which keeps track of the vertices of the bridges that have been picked from the heap, knows all vertices have been added when a vertex’s data is equal to the total (negative) number of vertices in the graph. Our project was limited by its efficiency, as well as its utility. It is not as efficient as it could have been because we rarely went back to edit code that already worked. In utility, the project is limited in that it can only benefit the person who has to figure out what bridges to build. A contractor still has to show up to the construction site and give an approximate (imperfect) estimate of the build cost. Ideally, a project of this sort would be able to figure out exactly how much a bridge from vertex a to vertex b would cost, in addition to what bridges should be built. We could have made our algorithms more runtime efficient by increasing class interdependency and reducing the number of functions that were developed multiple times in separate classes.