Rich People Problems #27:

- “Too Many Islands” -

And building the fewest bridges between them via a minimum spanning tree computed by Kruskal’s algorithm

Proposal

**Devin Janus**

**CSC316: Data Structures for Computer Scientists**

**dwjanus@ncsu.edu**

**North Carolina State University**

**Department of Computer Science**

**March 21, 2016**

Executive Summary

The team has been employed to create a piece of software that will determine the lowest possible cost to build a series of bridges between a cluster of islands, provided the estimates to build a bridge between each adjacent island –per island. The main objective of this project is to minimize cost, with the only secondary criteria being that every island is reachable.

With these goals in mind, we will be implementing Kruskal’s algorithm to determine the minimum spanning tree (MST) as it relates to the cost of the bridges between the various islands. This algorithm will find the absolute fewest needed bridges to be able to travel to every island, while prioritizing the weight (cost) of the bridges. This will essentially produce one path that connects all the islands using the cheapest possible configuration of bridges.

In analyzing the project requirements and specifications, the team has determined that the most difficult challenges ahead will be writing the code for the logic contained within the Graph data structure and the implementation of Kruskal’s algorithm. In preparation for these obstacles, extra time has been allotted within the task plan to accommodate for potential setbacks.

Table of Contents

[Problem Statement 1](#_Toc440806182)

[Project goals & Benefits 2](#_Toc440806183)

[Algorithm Design 3](#_Toc440806184)

[Proposed algorithm](#_Toc440806185) 3

[Algorithm Design Rationale](#_Toc440806186) 5

[Algorithm Analysis](#_Toc440806187) 6

[Software Design](#_Toc440806188) 7

[Design Pattern(s)](#_Toc440806189) 7

[Black Box Test Plan](#_Toc440806190) 8

[Task Plan](#_Toc440806191) 10

# Problem Statement

For this assignment the team has taken on a private contract from local billionaires, Mike and Maria Money. The couple just recently purchased a large group of islands in the Caribbean and they would like to connect them with a network of bridges. However, wealthy as the couple is, they intend to accomplish this goal while spending the least amount of money possible. Their only requirement is that there is some kind of way to reach every island they own, with the priority being to keep net expenditure at an absolute minimum.

Because so much emphasis is being placed on the cost of the project, the best way to tackle the problem will be through the implementation of Kruskal’s algorithm, which takes a weighted Graph of vertices and edges and creates an acyclical tree structure that contains every vertex of the graph and has the lowest possible total weight. In this case, the vertices of the graph will represent all of the individual islands and the edges of the graph, bridges where the “weight” of each edge will be the bridge’s estimated cost in dollars.

The software will read in a specifically formatted text file, of which each line corresponds to the information of an edge in the graph. After reading a line of text, the program will construct an Edge (vertex1, vertex2, weight, next) from the input that will then be inserted into a weighted, undirected graph, adjacencyList(V,E) containing all possible bridges (E) – in the form of unordered pairs of edges(vertex1,vertex2) – from all islands (V) where V is the set of islands, represented by Vertex (n, E, next), where n is the neighbor vertex u of the object, E is the Edge(v, u, weight, next) and next is the pointer to the next vertex record in the adjacency list. After obtaining the edge and adding it to the adjacenyList of its two vertices, the edge gets inserted into an array-based heap that is partially ordered according to edge weight (the key for each edge will be its weight).

Once the file is done being read and the adjacencyList is built, we will initialize the and heap are complete, we will get the number of islands from the size of our heap which will be the upper bound necessary to run our heap and up-tree methods.The heap and our adjacencyList graph will then be passed into our Kruskal method which will output the MST as an up-tree. The program will then print to system.out the contents of the heap after all insertions, the edges of the mst sorted by vertex and the adjacency list.

# Project goals & Benefits

The goal of this project is to develop a piece of privately usable software for the Money family that will determine which bridges they should build to yield the lowest possible expense, while still being able to reach every island they own. The software in question must be able to construct a weighted graph from a text file of edges and a heap of the edges in the graph. From these structures, the software must then be able to implement Kruskal’s algorithm to determine the MST of the graph.

Still currently under development, the system to be created will be one which is concise, minimal, lightweight, and efficient; capable of calculating the minimal cost from thousands of potential bridges.

The benefits of this project are fairly singular and purely financial for the sponsor, who is currently spending an exorbitant amount of money to buy so many islands. The benefit to the team will be the free tropical vacation we get out of this. As well as a much better fundamental understand of graph theory and its application to data structures in java.

# Algorithm Design

The following section describes the behavior and logic of the methods to be implemented within the MinimumBridgeManager program. Included are insert and deleteMin, which will be used in our heap structure, union and find, which are used in our up-tree, and Kruskal’s algorithm, as it corresponds to our specific implementation. Also included are the algorithms for the supporting heap methods, upHeap and downHeap.

## Proposed algorithms:

**Algorithm**: insert(key k, element e, array H )

**Input:** a key (k) for an element (e) to insert into array (H) with n elements

key( H[ size(H) ] ) <- k

element( H[ size(H) ] ) <- x

size(H) <- size(H) + 1

upHeap( H, size(H) – 1 )

**Algorithm**: deleteMin( array H )

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

**Output:** the minimum element in the array

x <- element( H[0] )

size(H) <- size(H) – 1

swap( 0, size(H) )

downHeap( H, 0 )

return x

**Algorithm**: union( node S, node T )

**Input:** a root node S and a root node T

**Output:** the root node of the combined set

if count(S) is greater than or equal to count (T) then //🡪 if (key(find(S)) >= key(find(T))…

count(S) <- count(S) + count(T)

parent(T) <- S

return S

else

count(T) <- count(s) + count(T)

parent(S) <- T

return T

**Algorithm**: find( node p )

**Input:** a node p

**Output:** the root node of the set containing p

while parent(p) is not null do

p <- parent(p)

return p

**Algorithm**: Kruskals(graph G, heap H )

**Input:** a graph (G) and a heap (H) which contains the edges in G, partially ordered by weight.

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

upTree <- null

components <- |V| //each vertex will start out as a component by itself (= num of V in G)

for each vertex v in G do

makeSet(v)

while components > 1 do

(u,v) <- deleteMin(H) //process edges by increasing weight

U <- find(u)

V <- find(v)

if U does not equal V then

union(U,V)

upTree <- union( upTree, (u,v) )

components -= 1

return upTree

**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 is greater than 0 then

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

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

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

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

**Input:** an array (H) with n elements and position (m) in the array

i <- 0

if 2m + 2 is less than size(H) then

if key( H[ 2m+2 ] )is less than or equals to key( H[ 2m+1 ] ) then

i <- 2m + 2

else

i <- 2m + 1

else if 2m + 1 is less than size(H) then

i <- 2m + 1

if i is greater than 0 and key( H[m] ) is greater than key( H[i] ) then

swap(m,i)

downHeap(H, i)

## Algorithm Design Rationale

In order to implement the above algorithms, an abstract ArrayList structure will be heavily utilized in conjunction with an UpTree, Heap and Graph data structures. As input is read from the file, an edge object will be built and, using the information from the edge, a vertex will also be constructed if it does not yet exist. The edge, (and potentially vertex) is then passed into the Graph where the insert method will place the edge/vertex in the correct place. The edge object is then added to the Heap. Once all edges have been read in, the graph will know how many vertices it has and be able to build the UpTree. The heap and Graph structures are passed into the KruskalsMST method which will build and return the MST in the form of an UpTree.

Because the above algorithms rely heavily on keeping order of the entries by weight and being able to quickly retrieve specific indexes, an ArrayList is used. As opposed to a LinkedList, which would require a O(n) to retrieve a specific index. Also, given that a Heap is a complete binary tree, the implementation of an ArrayList allows for a mathematically accurate way to ensure proper insertion of nodes as well as retrieval/verification of parent nodes. In the context of our Graph, the ArrayList will prove more useful when inserting edges and vertices because we will not have to iterate through (potentially) the entire list before each insertion.

## Algorithm Analysis:

The exact running time T(n) for the **insert** algorithm is:

T(n) = 3 + log(n) and has O(log(n)) seeing as the majority of the algorithm is linear, inserting an element at the end of the array is O(1), the upHeap-ing requires O(log(n)).

The exact running time T(n) for the **deleteMin** algorithm is:

T(n) = log(n) + 4 and has O(log(n)) because all array-based operations in the function are O(1) leaving downHeap, which has O(log(n)) as the maximum running time.

The exact running time T(n) for the **union** algorithm is:

T(n) = 2(log(n)) + 6 and has O(log(n)) because the tree only deals with pointers, assigning the parent values of the proper nodes is O(1) which leaves the find algorithms for the parents of the input nodes to have the longest running time in the algorithm.

The exact running time T(n) for the **find** algorithm is:

T(n) = 2 + (n/2) and has O(log(n)) seeing as the time to find a node within the tree is, at most, the height of the tree itself where *h* is <= log*n*, n >= 2^*h*. Despite an array-based implementation, this particular find algorithm looks for the parent of the input node within the tree, therefore it will not have a O(1) like a standard find within an ArrayList because it must iterate up through the parents of p.

The exact running time T(n) for the **Kruskals** algorithm is:

T(n) = m(3log(n) + 1 + 2log(n) + 1) + n + 2 => m(5logn +2) + n + 2 => 5mlogn + 2m + n + 2

and has O(mlog(n) ) because the while loop must iterate through each vertex V in the graph and for each iteration (m), the max running time of any line is going to be logn for the union operation.

The exact running time T(n) for the **upHeap** algorithm is:

T(n) = 8 + T(n/2) => log(n) + 8 and has O(log(n)) due to the fact that our heap is essentially an array-based implementation of a complete binary tree.

The exact running time T(n) for the **downHeap** algorithm is:

T(n) = 17 + m(log(n)) and has O(m\*log(n) ) because with each iteration we must recheck the nodes in the heap, the longest possible running time would be m \* the time it takes to iterate through a binary tree = logn. Therefore, the maximum running time would be m\*logn.

Because this project revolves heavily around the implementation of array-based binary trees for storage and sorting the max running time for most of the algorithms is O(log(n)). This is due to the fact that we are able to retrieve any indexed item in our various data structures with O(1) thanks to the use of ArrayLists. As opposed to if we used LinkedLists, which would cause our find methods (used to some degree in almost every algorithm) to have a much slower running time of O(n), extrapolated across an entire tree build, the use of LinkedLists would lead to exponential running times.

# P2P1_UML_dwjanus.jpgSoftware Design

## Design Pattern(s)

This software will utilize the Model-View-Controller design pattern to implement the behavior required in the program spec. The MVC design pattern is best suited for this project seeing as it is to be delivered as a piece of software for a private consumer who is only concerned with the final output. The view of which, will be contained within the MinimalBridgeGUI part of the program and strictly handles the user I/O. The GUI will get the file name of bridge information from the user and pass it to the MinimalBridgeManager class constructor which will take the input string to make the file and then process it line by line, building an edge for each line, passing that edge into the adjacencyList and then adding it to it’s heap. This class acts as the controller for the design. It will also be responsible for building the MST by passing the initially constructed Graph and Heap into the KruskalMST method. The model portion of the design will be contained within the data structures of the program. The Heap, Graph, UpTree and ArrayList(s) will hold the logic for the software and will be used by the controller.

# Black Box Test Plan

The MinimalBridgeManager acts as a conduit for the implementation of Kruskal’s algorithm to determine the minimum spanning tree in a graph. This program is designed to construct a series of data structures in order to calculate and output a Tree object (as an UpTree). The customer will be able to access the files they wish to analyze via a stylistic and minimal GUI and upon successful build, view the contents of the heap, the edges contained within the MST and the initial adjacency list. The program utilizes a GUI for the user to be able find their file by either a drop down browse menu or a text field in which they can enter the explicit file path.

This black box test plan intends to test many of the use-cases and possible subflows involved with the file select user interface.

First, the user must start the application. This can be achieved throught the MinimalBridgeGUI as indicated in the design proposal. If using Eclipse IDE, first right-click the MinimalBridgeGUI class in the MinimalBridge source folder. In the menu, scroll down to Run as > Java Application . Or press Alt+Shift+X, J.

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Description | Expected Results | Actual Results |
| InvalidFileType  (Devin Janus) | **Preconditions:**  There **EXISTS** a file “invalidFile.jpg”  The program starts up and the user is prompted for a file.  The user inputs “invalidFile.jpg” in the text field | A FileType exception will be thrown and an error message will be displayed |  |
| InvalidFileName  (Devin Janus) | **Preconditions:**  There **DOES NOT** **EXIST** a file “validFile.txt”  The program starts up and the user is prompted for a file.  The user inputs “validFile.txt” in the text field | A FileNotFound exception will be thrown and an error message will be displayed |  |
| InvalidFileFormat  (Devin Janus) | **Preconditions:**  There **EXISTS** a file “invalidFile.txt” which contains characters and not integers for bridges.  The program starts up and the user is prompted for a file.  The user inputs “invalidFile.txt” in the text field | An ImproperFileFormatException will be thrown and an error message displayed. |  |
| ValidBridges  (Devin Janus) | **Preconditions:**  There **EXISTS** a file “validBridges.txt” which contains properly formatted bridge data.  The program starts up and the user is prompted for a file.  The user inputs “validBridges.txt” in the text field | No error, the file is selected and the program successfully passes it from the GUI to the MinimalBridgeManager. MST is built successfully. |  |
| Valid emptyBridges  (Devin Janus) | **Preconditions:**  There **EXISTS** a file “validEmpty.txt” which contains no text whatsoever.  The program starts up and the user is prompted for a file. | No error, the file is passed through and the MST is built successfully, it is just empty. |  |

# Task Plan

In the task plan below, most objects in the preparation phase will presumably require only a day or so to complete. These tasks include the document inspection, proposal authoring, algorithm and black box test designs and revisions. However, in the soon to come development phase of this project, a lot more space has given to the various tasks outlined. This is to allow for the factoring in of unforeseen drawbacks and to allow for the possibility of having to revise any algorithms or redesign the software model with plenty of time as a buffer. Ideally, the framework code for the entire project would take an evening. The next step would be to finalize and test the data structures from the bottom up so as to provide a little more direction with the code and assert accuracy with parent methods. Once complete, the required methods will be finalized and JUnit tested before tested in Jenkins to verify success. Next will be the major chunk of the code to be used within the KruskalMST method, this will most likely take the longest to write and therefore is the most heavily weighted task as it will require a lot of debugging to ensure the data structures behave appropriately. It is also assumed that various other bugs will appear at this junction. All other tasks have been given roughly a 24hour window to complete. With this schedule there are plenty of buffer days which will allow for any overflow should a task take longer than projected.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Task Description | Owner | Planned  Start Date | Planned  End Date | Status |
| Document inspection/organization | Devin | 3/15/16 | 3/15/16 | Complete |
| Proposal write-up (beginning) | Devin | 3/16/16 | 3/16/16 | Complete |
| Algorithm Design | Devin | 3/17/16 | 3/17/16 | Complete |
| UML Diagram | Devin | 3/18/16 | 3/18/16 | Complete |
| Black box test plan | Devin | 3/19/16 | 3/19/16 | Complete |
| Revise/edit summary and various write-up information for Part 1 submission | Devin | 3/20/16 | 3/20/16 | Complete |
| Skeleton Code – make base data structures | Devin | 3/21/16 | 3/21/16 | Pending |
| Fatten up that skeleton, begin writing bayou-adjacent JUnit tests and make sure data structures work properly | Devin | 3/22/16 | 3/23/16 | Pending |
| Lay down beta GUI to allow for working model | Devin | 3/24/16 | 3/24/16 | Pending |
| KruskalsMST and Graph methods/structures. If they don’t work now, this is the part where they start. | Devin | 3/25/16 | 3/28/16 | Pending |
| Debug, apply finishing touches to code, check Javadoc, PMD, checkstyle and ensure code/test coverage. Polish GUI if necesary | Devin | 3/29/15 | 4/2/15 | Pending |
| Write and edit final report | Devin | 4/3/15 | 4/4/15 | Pending |

**Team Contact Information**

Devin Janus: Captain, Alternate, Recorder, Design Analyst, Auctioneer, Senior Testing Engineer, QA

[dwjanus@ncsu.edu](mailto:dwjanus@ncsu.edu) 919.586.1684