ECE 250 - Project 4 – Graph with mst – Design Document

Easson Weisshaar, UW UsedID: etcweiss

December 3rd

Overview of Classes

good

Class:

node - functions as a vertex and allows the storage of researcher numbers

private member variables:

ResearcherNo: a int that stores the researchers number, **includedAsDestination**: a vector that stores pointers to nodes that include edges that store the node as a destination, **discovered**: a bool that stores whether a node has been "discovered" yet, used for the mst function

public member functions:

getRn: returns an int of the *ResearcherNo* stored, **addToDests**: takes in a node pointer as an argument and adds to the *includedAsDestination* vector, **deleteFromDests**: takes in a int argument that correlates to a node in the *includedAsDestination* and removes it, **searchDests**: searches for a given *ResearcherNo* and returns the index it's located at. **changeDiscoveryStatus**: makes the value of *discovered* it's complimented form, **getDestListSize**: returns the size of *includedAsDestinationi*, **getDest**: given an int argument, it returns the correlating element in the *includedAsDestination* list, **getDiscoveryStatus**: returns the value of *discovered*

Class:

edge - allows the storage of edges which includes a origin, destination and weight

private member variables:

origin: a node pointer that stores the origin of the edge, **destination**: a node pointer that stores the destination of the edge, **weight**: a double that stores the weight of the edge

public member functions:

get_origin: returns the *origin* of the edge, **get_destination**: returns the *destination* of the edge, **get_weight**: returns the *weight* of the edge,

Class:

graph - stores the graph throught the vertices and edges

private member variables:

adjacencyList: an array of vectors that stores edge pointers with size 23134 that acts as an adjacency list, size is 23134 to prevent the need to index one less to access the correct element, **vertexCount**: an int that stores the amount of vertices in the graph

public member functions:

printSize: prints the *vertexCount* with the proper surrounding words as specified, **insert**: takes in an int for both ends of the edge and a weight and creates an edge out of it and then stores it in *adjacencyList* at the appropriate index, **returnSize**: just returns the value of *vertexCount*, **deleteNode**: deletes all the edges in its index in *adjacencyList*, then accesses the nodes *includedAsDestination* vector and accesses those edges throught those edges origins spot in the index, **mst**: using Prim's algorithm, we start at the given node and add all the outgoing edges to a max heap, then in the temporary *graph* called mstGraph, we store the heaviest edge. We then repeat the process with the origin of that heaviest edge and continue by checking if an edge's destination has already been discovered or if the destination is already included in the *mstGraph*, **printAdjs**: prints the *ResearcherNo* of all the adjacent nodes by accessing the *adjacencyList* at the index of the passed argument of the function

Class:

heap - maximum heap that assists the mst function in achieving specified runtime

private member variables:

edgeHeap: a vector of edge pointers that stores the contents of the heap, **sizeOfHeap**: stores the amount of elements in *edgeHeap*, different than the size as the element at index 0 is the nullptr

public member functions:

parent/left/right - edgeIndex: three individual functions that return the parent, left child, and right child of the passed index respectively, **returnMax/extractMax**: two separate functions that function the exact same way except *extractMax* also removes the first element and queues up the next biggest element whereas *returnMax* only returns the first element and doesn't do any mutations, **addNewEdge**: adds a new edge tothe end of *edgeHeap* and then inserts it at the correct index by calling *moveUp,* **move – Up/Down**: two separate recursive functions that will move and element up or down respectively. They compare the parent or children indexes based on which functions and then also compare the weights and will move to maintain the heap rules, **mySwap**: a function that swaps two elements in the *edgeHeap*, **returnSize**: returns the value of *sizeOfHeap,* **returnHeapIndex**: returns the element stored at the index passed as argument

ECE 250 - Project 4 – Graph with mst – Design Document

Easson Weisshaar, UW UsedID: etcweiss

December 3rd

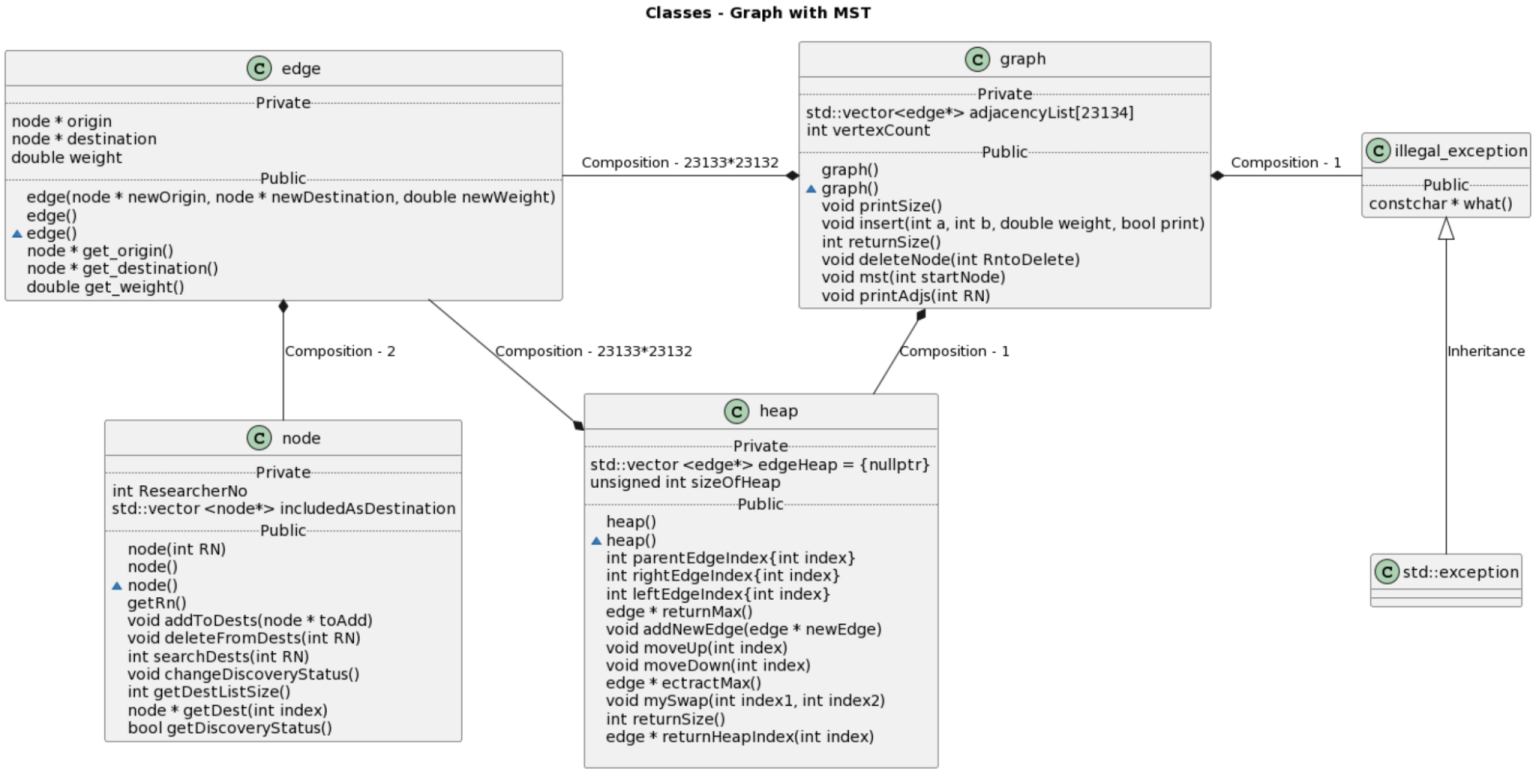
Class:

illegal_exception - exception class that allows certain arguments to have the correct output

public member functions:

what: returns the string “illegal argument”

UML Diagram



Constructors/Destructors

Node:

- (constructor) takes in an int RN and sets *ResearcherNo* to RN and sets *discovered* to false
- (default constructor) sets *ResearcherNo* to 0
- (Destructor) calls clear on the *includedAsDestination* vector

good

Edge:

- (constructor) takes in node pointers for origin and destination and a double for weight and sets the correlating private member variables to those parameters
- (default constructor) sets both node * pointers to nullptr
- (destructor) deletes the origin and destination variables and sets address to nullptr

Graph:

- (constructor) sets vertexCount to 0
- (destructor) clears every vector that has a size above 0 in *adjacencyList*

Heap:

- (constructor) sets *sizeOfHeap* to 0
- (destructor) calls clear on the *edgeHeap* vector

Test cases

ECE 250 - Project 4 – Graph with mst – Design Document

Easson Weisshaar, UW UsedID: etcweiss

December 3rd

load:

- calling load displays “success” only once and testing certain commands gives the impression that everything was added properly

insert (i):

- using invalid arguments results in an output of “illegal argument”
- can’t re-add or redefine an edge that already exists
- add multiple outgoing edges from one node without issue
- able to add a pair where there is an edge going to opposite way already, i.e. 1→2 and 2→1 both are valid and work
- able to reinsert recently deleted nodes

print (p):

- prints a newline if there are no outgoing edges
- prints all adjacent nodes in the order they were added with proper spacing
- doesn’t print any recently deleted nodes
- only prints those directly connected
- doesn’t print for a nonexistent node
- in the case of illegal arguments it outputs “illegal argument”

delete (d):

- deletes the correct node
- anytime the deleted node was included as a destination, those edges got deleted as well
- deletes all of deleted nodes outgoing edges
- doesn’t delete a nonexistent node
- in the case of illegal arguments it outputs “illegal argu ment”

mst:

- outputs “failure” if passed argument doesn’t exist
- if node is a single isolated node, outputs 1
- outputs correct mst with passed argument
- mst changes properly with insertions and deletions -5 more edge cases required

size:

- size changes by two when an insertion involving two new nodes happens
- changes by 1 when only one new node is introduced
- no change if both nodes already existed
- deleting a node only decrements size by 1
- prints 0 when all nodes are deleted

exit:

- program ends on exit

Performance considerations

To achieve a constant runtime for the “size” command, I had the variable *vertexCount* in my graph class track the number of nodes added to the graph and retrieve that value in the command. To achieve a time complexity of O(degree(a)) for printing (p) adjacent nodes in order of addition, I would pushback into the vector stored at the passed index, so it would always be ordered correctly and then just loop through that vector and print every element. To achieve a time complexity of O(|E|log|V|) you must implement a max heap and an adjacency list. The reason why the specified runtime is achieved is that for every time the max is extracted, the cost is O(logV) and since there are it will get called V time, due to the number of nodes, the time complexity is O(VlogV), since we iterate through each edge to add to the heap, we heapify for every edge added, therefore we get the time complexity O(ElogV + VlogV) = O((E+V)logV), since |E| >= |V|, it simplifies to O(ElogV) time complexity. In the case of my

good

implementation for the insertion function, the worst-case time complexity is O(degree(a)) where a is the origin. The reason why is that if there already existing outgoing edges for node an (origin) the program iterates through and determines whether or not the destination of the scanned edge matches the destination currently trying to get added. For my deletion function, the time complexity analysis is as follows. When deleting, It goes through the *includedAsDestination* vector and then accesses each of those nodes' adjacency lists each time and iterates through until the edge containing the node to be deleted is detected. So for that step, the time complexity is O(VlogE) as the worst case is that all nodes are in the *includedAsDestination* list and all edges contain the node to be deleted, the second part is simply O(degree(a)) (where a is the node getting deleted) because it iterates through out the index in the adjacency list where a's edges are stored and deletes them all. Therefore time complexity is O(VlogE + degree(a)).