ECE250 Project 4 Design Document Matthew Li (m739li)

For this project, I designed 4 classes: Graph, MinHeap, MinHeapNode, and illegal_exception. These classes are stored in their own separate header file in order to keep the code separated and organized.

Graph

Member Variables

adjacencyList (std::vector<std::tuple<int, int>>*): Stores the adjacency list of the graph.
numberOfVertices (int): Stores the current number of vertices in the graph.
existingVertices(std::vector<int>): Stores the station number of the current existing vertices so that a linear search is not needed to find the vertices in adjacencyList.

Member Functions

Graph(): Assigns adjacencyList to an array of size 50001 to store stations 1 to 50000. Index 0 is not used so the size must be 50001, and subtracting 1 from the station number was going to be too complicated than simply increasing the array size by 1. The constructors also initializes the number of vertices to 0.
~Graph(): Deletes the adjacency list. There is nothing else allocated in this class so nothing else needs to be freed.
int getNumberOfVertices(): Getter for the current number of vertices in the graph.
int getAdjacencyListDest(int firstIndex, int secondIndex): Getter for the destination of an edge in adjacencyList based on firstIndex and secondIndex.
int getAdjacencyListWeight(int firstIndex, int secondIndex): Getter for the weight of an edge in adjacencyList based on firstIndex and secondIndex.
int getAdjacencyListSize(int index): Getter for the size of an index in adjacencyList.
int getExistingVertices(int index): Getter for the value of an index in existingVertices.
void addEdge(int a, int b, int w, bool flag): This member function is called when the INSERT or LOAD command is called. The flag boolean indicates whether to output anything, since the LOAD command doesn't output success/failure per edge but the INSERT command does. First I iterate through adjacencyList at index a and check if any of their destinations are equal to b. If an edge is found, output failure and return to main. This takes worst case O(E) time if all the edges are connected to vertex a. I then check if both of the vertices already exist in the graph; if not, I increment numberOfVertices accordingly and update existingVertices. I then add the edge to adjacencyList with a tuple of the form "weight, destination" to both indexes a and b since this is an undirected graph, and return to main. This takes O(E) time overall because I have to iterate through the adjacencyList index, and all other commands are done in constant time.
void printVertices(int a): This member function is called when the PRINT command is called. First I check if adjacencyList at index a has no edges which takes constant time, in which case I output failure and return to main. Otherwise, I iterate through this index, each time outputting the destination which takes worst case O(E) time if all edges are connected to vertex a, then return to main. This takes O(E) time overall trivially.
void deleteVertex(int a): This member function is called when the DELETE command is called. First I check if adjacencyList at index a has no edges, meaning the vertex does not exist, which takes constant time. If so, I output failure and return to main. Otherwise, I iterate through this index which takes O(E) time, and for each element, I iterate through adjacencyList at the destination index in order to remove the edge from the destination index which also takes O(E) time. To remove the edge, I use the vector's erase function which takes O(E) time. After the edge is found, I check if there are 0 edges at the destination index of adjacencyList, in which case I iterate through existingVertices to remove the vertex which takes O(V) time. After the outermost iteration is done, I iterate through existingVertices again to remove vertex a which takes O(V^2) due to the runtime of vector's erase function and return to main. In the worst case where all loops must be run, this takes O(E) * (O(E^2) + O(V^2)) + O(V^2) = O(E^3).
void primMST(bool flag): This member function is called when the MST or COST command is called. The flag boolean indicates whether MST or COST was called, since the outputs are different depending on the command, but most of the algorithm is the same. First I check if there are no vertices in the graph, in which case I output failure or cost is 0 depending on the flag and return to main. Otherwise, I initialize all vertices to

have a key of INT_MAX which acts as infinity and their parent to -1, indicating that a parent has not been found. This is done in O(V) time because I have to loop through all vertices. I then set the 0th node's key to 0 and I don't need to build the heap because every other key has a value of INT_MAX except for the 0th one. I then iterate through the min heap until the size is 0, each time extracting the minimum vertex, iterating through all the edges of that vertex, and setting the parent of the vertex to either the minimum vertex or the vertex of an edge with less weight and modifying the key. Iterating through the min heap takes O(V) time, extracting the minimum vertex takes O(logV) time which is explained later, iterating through all the edges of that vertex takes O(E) time, and setting the parent and modifying the key takes O(logV) time which is also explained later. Overall, this takes O(V) + O(V) + O(VlogV) + O(ElogV) = O(ElogV). Lastly, I output the MST or the cost depending on the flag and return to main.

MinHeap

Member Variables

size(int): Stores the current number of heap nodes in the heap.
capacity(int): Stores the maximum size of the heap.
stationToIndex(int*): Maps the station number (1 to 50000) to an array index (0 to V).
unknownVertices(MinHeapNode):** Stores all vertices not yet added to the MST.
verticesToDelete(MinHeapNode):** Stores all MinHeapNodes in order to delete them when the destructor is called.

Member Functions

MinHeap(int capacity): Sets the size and capacity equal to the capacity parameter. Initializes an int array of size capacity to stationToIndex, a MinHeapNode* array of size capacity to unkownVertices, and a MinHeapNode* array of size capacity to verticesToDelete.
~MinHeap(): Iterates through verticesToDelete, on each iteration deleting the element at that index. This deletes all allocated MinHeapNodes, and I then delete the verticesToDelete array, the unknownVertices array, and the stationToIndex array. The MinHeap has no other memory allocated to it so nothing else needs to be deleted.
void setVerticesToDelete(int index, MinHeapNode* value): Sets the value of verticesToDelete at the index to value.
void setUnknownVertices(int index, MinHeapNode* value): Sets the value of unknownVertices at the index to value.
int getStationToIndex(int index): Returns the value of stationToIndex at the index.
void setStationToIndex(int index, int value): Sets the value of stationToIndex at the index to value.
int getSize(): Returns the current size of the heap.
void minHeapify(int index): Starts at the index node and goes down the tree, swapping the nodes if at least one of the children is smaller than the index node. Recursively calls itself on the new smallest child if there was one until either a leaf node is reached or neither of the children is smaller than the index node. All operations other than the recursive call are done in constant time, so this has a worst case runtime of O(logV) which is when minHeapify is called on the root node and at least one of the children is smaller than the index node each time, meaning the entire tree is traversed from root to leaf.
MinHeapNode* extractMin(): First, I store the current root node which is what will be returned and put the last node into the root node. Then, I reduce the heap size by 1 and call minHeapify on the root node, before finally returning the previously stored root node. All operations are done in constant time except for minHeapify, which takes O(logV) as explained earlier, so this function has a runtime of O(logV).
void modifyKey(int currentDest, int key): First, I update the key at currentDest to the parameter key, and make the minHeap a heap again. Starting from the currentDest node, I check if the parent's key is greater than the currentDest node. If it is, I swap the two nodes, set the currentDest node to be the parent, and update the parent variable to be the new parent. This has a worse case runtime of O(logV) if modifyKey is called on a leaf node and all parent nodes are greater than the currentDest node, in which case I would traverse the entire tree from leaf to root.

MinHeapNode

Member Variables

vertexNumber(int): Stores the array index that maps the station number to an array index from 0 to V.
key(int): Stores the current weight to get to this node.
stationNumber(int): Stores the actual station number (an integer between 1 and 50000).

Member Functions

MinHeapNode(int vertexNumber, int key, int stationNumber): Sets the vertexNumber to the vertexNumber parameter, the key to the key parameter, and the stationNumber to the stationNumber parameter.
~MinHeapNode(): A default destructor is used since each individual MinHeapNode is deleted during the destructor of MinHeap.
int getKey(): Returns the key of the current node.
int getVertexNumber(): Returns the vertex number of the current node.
void setKey(int key): Sets the key of the current node to the key parameter.
int getStationNumber(): Returns the station number of the current node.

illegal_exception

One member function called **void print():** Outputs “illegal argument” and returns to main(). Due to the simplicity of this class, the default constructor and destructor is used.

UML Diagram