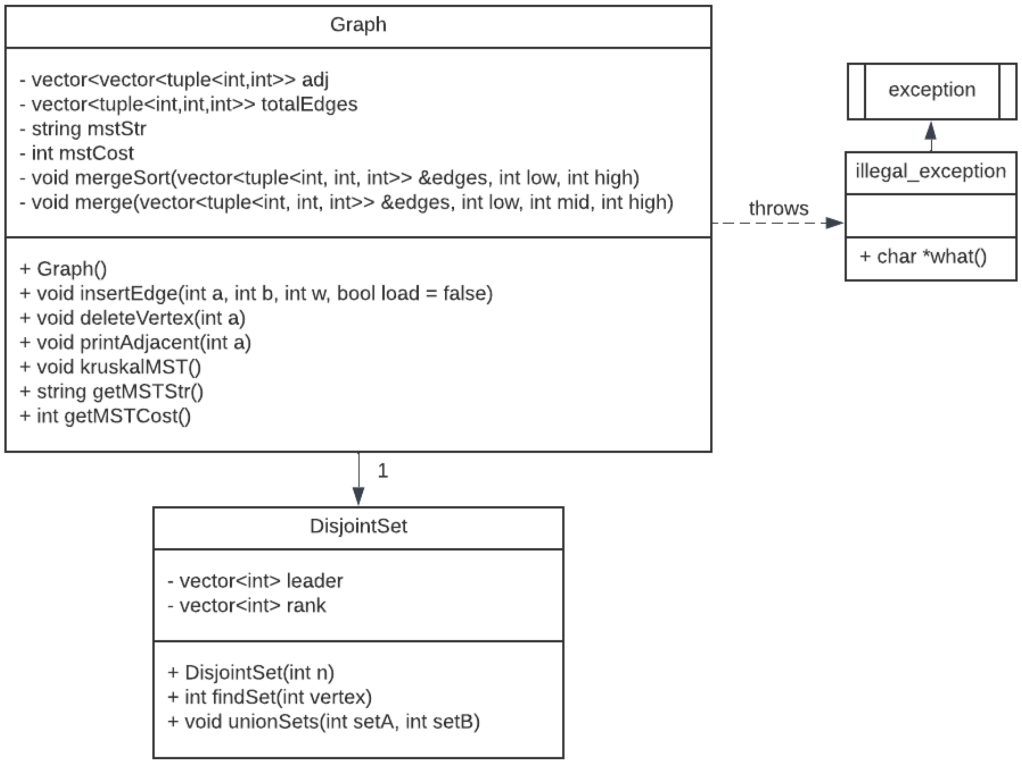
ECE250 Project 4 Design Document

Brandon Vo
April 3, 2023


Graph
- vector<vector<tuple<int,int>> adj
- vector<tuple<int,int,int>> totalEdges
- string mstStr
- int mstCost
- void mergeSort(vector<tuple<int, int, int>> &edges, int low, int high)
- void merge(vector<tuple<int, int, int>> &edges, int low, int mid, int high)
+ Graph()
+ void insertEdge(int a, int b, int w, bool load = false)
+ void deleteVertex(int a)
+ void printAdjacent(int a)
+ void kruskalMST()
+ string getMSTStr()
+ int getMSTCost()
throws
exception
illegal_exception
+ char *what()
1
DisjointSet
- vector<int> leader
- vector<int> rank
+ DisjointSet(int n)
+ int findSet(int vertex)
+ void unionSets(int setA, int setB)

Graph Class

In my graph implementation, I decided to use adjacency lists. This was made as a private member variable using a vector of vectors with a tuple that holds adjacent vertices, called *adj*. I chose to include an adjacency list instead of an adjacency matrix so that I could use less space: (O(|V|+|E|) instead of (O|V|^2). I wanted to have quick edge insertion and quick vertex removal to meet the project requirements. I also decided that I wanted to use Kruskal's algorithm to calculate the MST, so an adjacency list would be the most ideal for this situation. Another vector was created called *totalEdges* to store all the edges in the graph in the format <Va,Vb,Wab> which will be used in the MST algorithm. Two more private member variables are made called *mstCost* and *mstStr* to store the cost and MST output after the MST is computed. I store these values in case I want to print the MST or COST after the minimum spanning tree has already been computed once, and the graph hasn't been modified with an insertion/deletion since our last computation.

My graph class also has two private functions for sorting the edges using merge sort to be used in Kruskal's algorithm. *mergeSort()* is a void function that takes in a tuple vector with vertex a, vertex b, and the edge weight. It also has a low and high integer parameters used as indexes for dividing the graph into two halves. I also made a helper function for merge sort called *merge()* which takes in the same parameters to merge the two halves of a section in the vector.

For the public member variable and functions, I have a Graph constructor, *insertEdge(), deleteVertex(), printAdjacent(), kruskalMST(), getMSTStr(),* and *getMSTCost()* to compute the required commands. In the constructor, the adjacency list size is initialized to a size of 50001 to accommodate for vertices 1 to 50000, and the *mstCost* and *mstStr* variables are initialized to 0 and an empty string respectively. *insertEdge()* takes in an integer for vertex a, vertex b, and the weight between the edge. It also takes in a boolean *load* flag to determine if we are using this function while loading a file. Since we only want to print success once while loading files, we check if this flag is not set to ensure printing is done properly. *deleteVertex()* and *printAdjacent()* take one integer parameter for the vertex to be either deleted or have adjacent vertices printed. Finally, the *kruskalMST()* function does not require any parameters. It computes the MST of the current graph and defines the values to *mstStr* and *mstCost.*

DisjointSet Class

Since I have decided to implement Kruskal's algorithm, I created a DisjointSet class to keep track of connected components in the graph and prevent cycles from being created while calculating the MST. The set has two private member variables, which are vectors of integers for the *leader* (representative) and *rank* of each vertex. The leader is used to identify the set to which it belongs, and the rank defines the height of the tree.

For the functions, I made three public member functions in this class. The constructor, *DisjointSet*(), defines the size of the leader and rank vectors to the size of the adjacency list. The two other functions in this class which are used in Kruskal's algorithm are *findSet()*, which takes in a vertex integer to find the leader of the set that the given element belongs to, and *unionSets()*, which merges two sets into a single set. In *findSet(), we* return a pointer to the representative of the set to which the given vertex belongs to. I do this by recursively following the parent pointers from the vertex which forms a path, then compressing the path by making each vertex point directly to the *leader* representative. In *unionSets()*, I merge two disjoint sets by first checking the rank of the two sets. Since I always want to merge the smaller set into the larger set to keep the overall height small, I compare the ranks of each set to execute this. To merge the sets, I update the leader of the smaller ranked set to point to the leader of the larger ranked set, which makes both sets have the same leader. If the sets are equal, then I arbitrarily decided to merge *setB* into *setA*. I also increase the rank if both sets have the same rank.

illegal_exception Class
The illegal_exception class is used when an invalid input is trying to be used with a command. That is, if the INSERT, PRINT, or DELETE commands input a vertex that is larger than 50000 or less than or equal to 0, it will throw this class which prints out "illegal argument". Also for the INSERT command, we check if the weight is a valid input by ensuring that it is greater than 0. This is used in a try/catch statement to attempt to run those commands, then throwing this class which inherits the exception library and calls the exception::what() function to print the error.

Detailed Command and Runtime Analysis

INSERT: The INSERT command adds a new edge with an integer input *a*, *b*, and w (weight) to the graph using the *insertEdge()* function from the Graph class. As stated earlier, I chose to use an adjacency list to represent the graph, which would allow for quick insertion time. The insert function call is wrapped in a try/catch to ensure the input is valid. The function starts off by checking if inputs *a* and *b* are valid using the given restraints from the project description. If the input is invalid, it will throw the illegal_exception class. To determine if the edge already exists in the graph, I traverse through the edges of vertex *a* and compare if they match with vertex *b*. If it does, it means that a duplicate edge is trying to be inserted, in which "failure" gets printed and the function gets returned. Now, the adjacency list at vertex *a* gets vertex *b,* and the weight is pushed back to the vector, and the list at vertex *b* gets vertex *a* + the weight pushed back. The two vertices and weight also get pushed back to the *totalEdges* vector. *mstCost* and *mst*String are also set to 0 to update that the graph has been modified, which we later check in the MST and COST commands. Lastly, "success" is printed out to show that the insertion was successful.

For the runtime of the INSERT command, it takes O(degree(a)) on average, or at worst. This is because we need to iterate through all edges of vertex a to ensure that we are not inserting a duplicate edge. **If we assume that the graph is sparse, the run time is much closer to O(1).** This is because a sparse graph means that there are about as many edges as vertices, meaning we should not need to iterate through many/any edges of vertex a. This means we can insert an edge in constant time since we use an adjacency list.

PRINT: This command prints all vertices adjacent to input vertex a. In this command, we check if the input is valid using the same method as above and throwing the illegal_exception class if it is not valid. Next, we check if the vertex is not in the graph by using the *.size()* and *.empty()* vector functions on the adjacency list. If we find that the vertex is not in the graph, "failure" is printed out and the function gets returned. Otherwise, we iterate through all edges of vertex a and print the adjacent vertices as we retrieve them. **The runtime of the PRINT command is O(degree(a))** since we need to simply iterate through all edges of vertex a.

DELETE: The DELETE command removes the input vertex *a*, and any edges containing *a* from the adjacency list. It also removes the edge from *totalEdges*. When this command is used, *deleteVertex()* is called, which takes an integer parameter for the vertex *a* to be deleted. It first checks if the input is valid the same way as the INSERT and PRINT commands, and throws the illegal_exception class if not valid. Then, it checks if the vertex is not in the graph by using the *empty()* and *size()* functions on the adjacency list. If it is found to be empty, "failure" will be printed out. If the vertex does exist, we loop through all edges of vertex a to get the edges that connect to *a.* We then have an inner loop that goes through the adjacency list of vertex b and compares if the edge matches with the vertex we are trying to remove. If this is found, it gets erased and breaks out of this loop. We break out of the loop because we found the edge in the adjacency list of vertex b that connects to a. These loops will remove all edges that are connected to vertex a. After this, we also need to go through all edges in the graph and erase it if we find a match. Since *totalEdges* is in the form of <Va, Vb, Wab>, we check if either the first or second index matches the vertex we want to delete. If it does, we simply erase it. Otherwise we continue iterating through *totalEdges*. After that, we clear the vertex by looking it up in the adjacency list, and using the *clear()* function on it. Lastly, we reset the *mstCost* and *mstStr* to their original values, then print out "success" to mark that we need to recompute the MST if required, and show that we have successfully deleted the vertex.

The overall runtime of the DELETE command is O(V+E). To check if the input is valid and the vertex is in the graph, it takes constant time. The next step, which is deleting all edges from vertex a takes O(V+E) time. From looking at the code, there is an outer loop that iterates through all vertices. The inner loop iterates through all edges of the current vertex. Initially, this looks like the time complexity is O(V*E), however with the assumption that every vertex does not have E adjacent edges, this is not a tight bound. If we had a graph with only one edge, we would need to loop through all V vertices, then pick one edge from the adjacency list. This requires V + 2 operations. If we have two edges, this will require V + 4 operations, as we only iterate through the edges of the vertex. This means it takes O(V+2E) = O(V+E) time to delete the edges that connect to vertex a. After that gets removed, we also iterate through *totalEdges* and erase the edge if it contains vertex a. This loop takes O(E) time. Finally, the vertex gets cleared, *mstCost* and *mstStr* are reset, and "success" is printed out. These final instructions take O(1) time. As we can see, the runtime is O(V+2E+E) = O(V+E). **Thus, the worst case time complexity of *deleteVertex()* is O(V+E).**

MST/COST: The MST and COST commands either print out the entire minimum spanning tree in the format of Va Wb Wab(…) or the cost of the tree. Both MST and COST commands require you to calculate the minimum spanning tree at least once for every version of the current graph. These two commands use Kruskal's Algorithm with the *kruskalMST()* function to calculate the minimum spanning tree. This function requires no parameters and is a void function.

To compute the MST, we first check if the graph is empty by checking the adjacency list using the *empty()* function. If it is, we return the function. When using the MST command, we check if *mstStr* is empty or not. Since the adjacency list was found to be empty, "failure" is printed out in the main function. If the COST command was used, "cost is 0" would be printed out. This check takes constant time (O(1)).

The next step for calculating the MST using Kruskal's algorithm is to sort the edges in *totalEdges* by weight in a non-descending order. Using the *mergeSort()* function, I call the function with the *totalEdges* vector, and the low and high indexes of this vector to be sorted. The *mergeSort()* function splits the vector of tuples into two halves, calls itself recursively to sort each half, then merges the two sorted halves back together into a single sorted vector. It merges these two halves back together by calling a helper function called *merge()*. Using merge sort, sorting the total edges vector takes O(E log E) time for all cases. Although quick sort has the same average/worst case time complexity as merge sort O(E log E), I decided to not use quick sort since it has a worst case time complexity of O(E^2). Merge sort gives a consistent time complexity which I found to be ideal for sorting all variety of edge graphs.

After sorting the vector of total edges, we create a disjoint set with the size of our adjacency list using the custom DisjointSet class. This initializes the size of two vectors, *leader* and *rank*, to the adjacency list size. Each index of the *leader* vector from 0 to the list size is assigned to itself, and the *rank* of each index is set to 0. This takes constant time.

The next step in Kruskal's algorithm is to iterate through the sorted edges and add to the MST if they do not create a cycle. We do this by first using the *findSet()* on vertex *a* and *b*, which is a function in the DisjointSet that takes an integer argument for the vertex. This function checks if the vertex is already the representative of the set. If it is not, it recursively calls itself with the leader of the vertex as the argument until the leader of the vertex has been found. The purpose of this is to determine which connected component a vertex belongs to which is required for us identify if there is a cycle in the graph. To analyse the time complexity, we notice that the function performs a recursive call on each parent pointer until it reaches the root of the set. Our tree is kept small in height and balanced when we later union the sets by rank. In a balanced tree, if the tree has n elements, the maximum height of the tree is log E. Therefore, the time complexity of *findSet()* is O(log E).

After finding the representative of the set for vertex a and b, we check if set A and set B are in different connected components. If they are, that means there is no cycle and we can merge the two sets together with the *unionSets()* function. This function takes the two set representatives that need to be merged into one. To optimize the time complexity of this data structure, we always merge the smaller set into the larger set to keep the height of the tree as small as possible. If we were to insert the larger set into the smaller set, the resulting tree could become unbalanced and have a larger height than needed. This would make the *findSet()* function from earlier slower which is not ideal. Thus, if the rank of set A is larger than set B, then set B is merged into set A by assigning the leader of set B to A. If the rank of set B is larger, the same steps occur but vice versa. If both sets have the same rank, it does not matter which set gets merged into what, but we must increase the rank to ensure the height of the resulting tree is logarithmic in the size of the set. I chose to merge set B into set A. The reason why we don't always increment the rank when merging sets is because one set could be a much lower rank than the other. Increasing the rank would making the program slower, since we would have to traverse through a longer path. By incrementing the rank only when they have equal height, we keep a balance within the trees. The time complexity of calling this function is O(1) since it performs operations independent of the size of the input. The key to this function is keeping the tree balanced, which is important for us to keep an O(log n) time complexity for future *findSet()* calls.

Once both sets have been unioned, we add the current weight to *mstCost* and append "Va Vb Wab" to *mstStr*. This process is repeated for all edges in the total edges vector we created. After the MST is finished computing, we will either output MST in string format by using the *getMSTStr()* function, or output the cost of the MST with *getCost()*. This also takes constant time.

The runtime of this algorithm is O(|E|log(|E|)). Since E <= O(V^2), log(E) = log(V^2) = 2log(V) = Olog(V). This means the average/worst time complexity of the MST and COST commands is O(|E|log(|V|)). At best, we can get a time complexity of O(1) *IF* we have already calculated the MST once and the graph has not been modified with an insertion or deletion prior to the last MST computation. The dominating runtime of this algorithm is sorting the total edges vector with merge sort. **Thus, the final average/worst time complexity of COST and MST is O(|E|log|V|)**.