Abstract

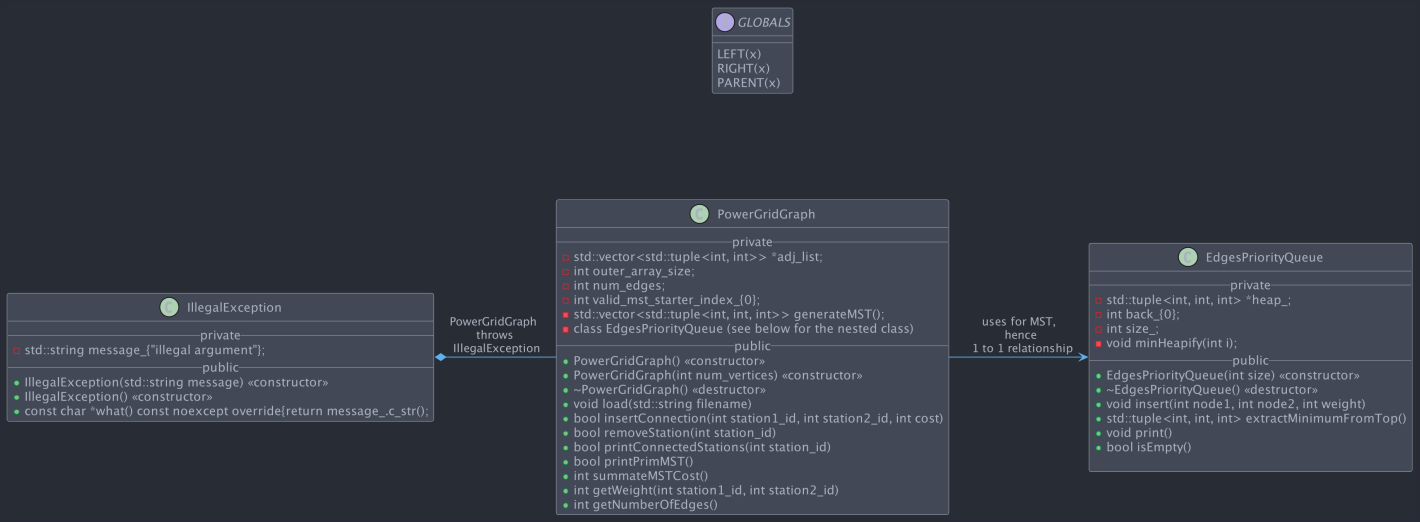
This is the design document of Chaitanya Sharma for Project 4 for ECE 250's Winter 2023 offering.

1 Introduction

My program consists of a major class PowerGridGraph, and one nested class inside that, namely EdgesPriorityQueue. My code has three macro functions(preprocessor directives) which are used to declare the left and right child and the parent of a node in the priority queue minimum heap.

I attribute the latex template which I've used for this design document to myself as I've used it in the past for my other projects in this course.

I've tested all edge test cases possible and also used the test cases provided by the jekelautograder on https://github.com/JZJisawesome/ece250-testcases.



2 Class Structure

2.1 class IllegalException

I attribute this class to myself as I've used it in the past for my other previous projects in this course. I'm using a default constructor and a parameterized constructor for this class. Though I'm not using the parameterized constructor in my code, I've included it for future use. This is a simple exception class which I've inherited from the std::exception class. It has a private member variable message_ which is a string which stores the message to be displayed when the exception is thrown. It has a constructor which takes a string as an argument and assigns it to the message_ variable. It also has a default constructor which assigns the default message to the message_ variable. It has a what() function which returns the message_ variable as a const char*.

2.2 class PowerGridGraph

This class represents the whole power grid graph. It has a private member variable adj_list which is a dynamically allocated array of std::vector of std::tuple of dual ints. The other important private member variable is num_edges which stores the number of edges in the graph, and the outer_array_size which stores the size of the static adj_list array.

There's also a valid_mst_starter_index_ private member variable which is used to keep track of the int index of the adj_list array which is the starting point of the MST generation. This lets me start the MST generation from any arbitrary node with more than 0 edges, and not just from the first node in the adj_list array which just couldn't just be there in some cases. The alternative to this would be to run a for loop at time of generateMST() function call to find the first node with more than 0 edges, but that would be a waste of time as well as ruin the time complexity of my generateMST() function. The value of this private member variable is maintained appropriately by insertConnection() and removeStation() functions.

This class has two constructors which are the default constructor and the value init constructor which takes in a single int as an argument, and dynamically allocates the adj_list array with the size. It also has a destructor which deallocates the adj_list array, to make the program memory safe.

Graph's Private Member Functions And Classes

2.2.1 std::vector<std::tuple<int, int, int» generateMST()

This function is used to generate the minimum spanning tree. It first checks if the number of edges is 0, and if it is, it returns an empty vector of tuples. Then it creates a boolean array of size outer_array_size and initializes all the values to false. I treat the boolean array as a set to check if a vertex has been visited or not in O(1) time, and set all of them to false.

Then this function creates a vector of triple int tuple, which will represent our final product, the minimum spanning tree of the graph. Then it creates a pointer to an object of the EdgesPriorityQueue class, with its size init constructor of the size of the number of edges. Then it creates a random index for the starting vertice of the minimum spanning tree, and sets the value of the boolean array at that index to true. This random index is assigned to the valid_mst_starter_index_ private member variable of the PowerGridGraph class. What this random index is determined by the insertConnection function and the removeStation function. Then it iterates through the adjacency list of the random index, and inserts the values of the tuple in the priority queue. Then it enters a while loop which runs until the priority queue is empty, and in each iteration, it extracts the minimum element from the priority queue, and stores it in a tuple.

The time complexity of Prim's algorithm is O(|E|log(|V|)), where |E| is the number of edges and |V| is the number of vertices in the graph. In this function, the number of edges is given by the variable num_edges, which is used to create the priority queue with an initial capacity of num_edges. The size of the priority queue can increase up to |E| during the algorithm execution, but its maximum size is O(|E|) because each edge can be inserted and extracted at most once. The while loop iterates until the priority queue is empty, which takes O(|E|log(|E|)) time because each insertion and extraction operation takes O(log(|E|)) time. Inside the while loop, there are two for loops that iterate over the adjacent vertices of the current minimum edge. The total number of iterations of these for loops is at most 2|E|, because each edge is visited twice (once for each adjacent vertex) at most. Therefore, the overall time complexity of this function is O(|E|log(|E|) + 2|E|) = O(|E|log(|E|)). However, the number of edges |E| may be proportional to the number of vertices |V| in the worst case (e.g., for a complete graph), so the time complexity can also be expressed as O(|E|log(|V|)).

Also, a good thing about my implementation is that, while researching about Prim's algorithm, I found out that a lot of people implement a hash table to find the home node for a given edge along with the priority queue, but what I did was while inserting into my priority queue and making it, I just stored the home node in the tuple itself, and that saved me a lot of time and space complexity allowing me to find the home node in O(1) time.

"MST" "COST" common function

2.2.2 class EdgesPriorityQueue (private nested class)

This class represents the priority queue which I've used to implement the generate_MST() function. While implementing the Prim's Algorithm as explained in the ECE 250 Lectures, I would need to repetitively access the most minimum edge out of a given set of possibly reachable unvisited edges from the visited edges, and there I've utilized the efficiency of a priority queue. This class has a private member variable heap_ which is a dynamically allocated array of std::tuple of triple ints. It has a private member variable heap_size_ which is just used to keep track of the size of the heap. Also, it has a private member variable back_ which is used to keep track of the actual size of heap as well as the index of the last element in the heap. This class's constructor and destructor deal with safely dynamically allocating and deallocating the heap_ array.

———— EdgesPriorityQueue's Private Member Functions ————

2.2.2.1 void minHeapify(int i)

This function is used to maintain the heap property of the priority queue. It takes in an int index which is the index of the node which is to be checked for the heap property and keeps on heapifying the subtrees. It checks if the node at the given index is greater than its children, and if it is, it swaps the node with the minimum of its children. It then recursively calls itself on the index of the child which was swapped with the parent node. The runtime complexity of this function is O(log(n)) because it is a recursive function. It takes an index i as input and assumes that the sub-trees rooted at the left and right children of the node at index i are already min-heaps. The function then compares the node at index i with its left and right children and swaps the node with the smallest child if necessary. The worst-case time complexity of minHeapify is O(log(n)) where n is the number of elements in the heap. This is because the function's runtime is proportional to the height of the tree, which is logarithmic in the number of nodes.

————EdgesPriorityQueue's Public Member Function ————

2.2.2.2 void insert(int node1, int node2, int weight)

This function is used to insert a new edge into the priority queue. It takes in three ints, which are the two nodes and the weight of the edge. It first creates a std::tuple of ints and assigns the three ints to it. It then checks if the heap is full, and if it is, it returns. It then assigns the std::tuple to the last index of the heap, and then increments the back_ variable. The function then performs a heapify-up operation to maintain the min-heap property. I did not feel the need to implement a separate recursive heapify-up function as the less recursion I do it's better/ The "heapify-up" operation starts from the newly added node and compares its value with its parent's value. If the value of the parent node is greater than the value of the newly added node, the two nodes are swapped. The process continues up the tree until the parent's value is smaller than the newly added node's value or until the root is reached. The worst-case time complexity of the insert function is O(log(n)), where n is the number of elements in the heap. This is because the function performs a heapify-up operation that takes at most O(log(n)) iterations to restore the min-heap property. We assume that the worst case is that the new node is the smallest node in the heap, and thus must be swapped with all of its parents, which would be the same as height of the tree.

2.2.2.3 std::tuple<int, int, int> extractMinimumFromTop()

This function is used to extract the minimum edge from the top of the priority queue. It first checks if the heap is empty, and if it is, it returns a std::tuple of ints with all the values as -1. If the heap has only one element, it returns that element and decrements the back_ variable. Otherwise, it assigns the first element of the heap to a std::tuple of ints, and then assigns the last element of the heap to the first element of the heap, which is the most minimum element according to the weight which is the third element of the tuple. It then decrements the back_ variable and calls the minHeapify function to maintain the min-heap property. The worst-case time complexity of the extractMinimumFromTop function is O(log(n)), where n is the number of elements in the heap, since it performs all actions in O(1) time i.e. constant time, and then calls the minHeapify function which takes O(log(n)) time to restore the min-heap property.

2.2.2.4 void print()

This is a completely optional function that I added to the priority queue class, and does not get called from the driver function but just exists for debugging purposes. This function is used to print the heap. It first prints the string "state: " and then iterates through the heap and prints the values of the tuple in each index. It then prints the string "==========" to separate the heap from the next output.

————————————Graph's Public Member Function————————————

2.2.3 void load(std::string filename)

This function is used to load the graph from a file. It takes in a std::string which is the name of the file. It just straight up opens the file and checks if it is open or not. If it is not, it returns voidly. Then it reads the first line of the file, which is the number of vertices in the graph, and assigns it to the outer_array_size variable. Then it reads the rest of the file line by line, and for each line, it calls the insertConnection function to insert the edge into the graph. There is no need for runtime analysis of this function as prescribed by the specification document. "LOAD"

2.2.4 bool insertConnection(int station1_id, int station2_id, int cost)

This function inserts a connection between two stations in the graph. It takes three parameters, station1_id, station2_id and cost, which are the IDs of the two stations and the cost of the connection between them, respectively. First, it checks if the IDs of the stations are valid (i.e., between 1 and 50000) and the cost is positive. If any of these conditions is not satisfied, it throws an IllegalException.

Then this function first gets the maximum of the two station IDs and checks if it is greater than the current size of the adj_list array. If it is, it creates a new array with the size of the maximum station ID and copies the contents of the old array to the new one. Then it deletes the old array and assigns the new array to the adj_list pointer. This is done to make sure that the array is large enough to hold the new connection.

Next, it checks if the two stations are the same. If they are, it returns false because a station cannot be connected to itself. Then it checks if there is already a connection between the two stations. If there is, it returns false because a connection

Afterwards, it adds the new connection to the adj_list array. Then it updates the valid_mst_starter_index_ by checking if the size of the adjacency list of the two stations is greater than the current value of valid_mst_starter_index_.

Also, after inserting, it checks if the number of edges in the current valid_mst_starter_index_ is greater than the number of edges in the station with the largest adjacency list. If it is, it updates the valid_mst_starter_index_ to the current station. If it seems a bit complicated in english, see what I've done with these ternary operators.

```
valid_mst_starter_index_ = (adj_list[valid_mst_starter_index_].size() > max(adj_list[station1_id - 1].size(), adj_list[station2_id - 1].size()) ?
valid_mst_starter_index_ : (adj_list[station1_id - 1].size() > adj_list[station2_id - 1].size() ? station1_id - 1 : station2_id - 1);
```

Finally, it increments the number of edges and returns true.

The runtime of the insertConnection function is O(1) in the best-case scenario but mostly true, where it only needs to insert the new connection into the adjacency list of the two stations. However, in the worst-case scenario where the size of the adj_list needs to be increased to accommodate the new station, the runtime is O(|V|), where |V| is the number of vertices in the graph. This is because the function creates a new adjacency list with size new_size, and then copies the old adjacency list into the new one, which takes time proportional to |V|. Basically any and all cases where the size of the array does not need to be increased will have a runtime of O(1). But if the size of the array does need to be increased, then the runtime will be O(|V|). Therefore, the overall runtime of the insertConnection function is O(|V|) in the worst-case scenario, "INSERT"

2.2.5 bool removeStation(int station_id)

This function removes a station from the graph. It takes one parameter, station_id, which is the ID of the station to be removed. First, it checks if the ID of the station is valid (i.e., between 1 and 50000). If it is not, it throws an IllegalException. Then it checks if the station is already removed. If it is, it returns false because a station cannot be removed twice. Then, it runs a for loop on the connected edges of the station to be removed. For each connected edge, it runs another for loop on the connected edges of the other station, and removes the edge from the other station's adjacency list. Then, it clears the adjacency list of the station to be removed. Finally, it checks if the station to be removed was the valid MST starter. If it was, it sets the valid MST starter to the first station in the graph that is not removed. Then, it returns true.

The time complexity of this function depends on the number of edges incident to the station being removed, which is at most the degree of the station. Let D be the maximum degree of any vertex in the graph.

The outer for loop iterates over all the edges incident to the station being removed. The number of iterations of this loop is at most D, the degree of the station. Inside the loop, there is another for loop that iterates over the adjacent vertices of the station, and the number of iterations of this loop is at most D as well. Therefore, the overall time complexity of the two nested loops is $O(D^2)$. The operation adj_list[station_id - 1].clear() has a constant time complexity. The last for loop iterates over all the vertices in the graph to find a valid MST starter index. The number of iterations of this loop is at most |V|, the number of vertices in the graph. The inner if statement has constant time complexity, so the overall time complexity of the loop is O(|V|). Therefore, the overall time complexity of this function is $O(D^2 + |V|)$. In the worst case, the graph is a complete graph, and D = |V| - 1, so the time complexity becomes $O(|V|^2)$. However, in practice, the degree of vertices is typically much smaller than |V|, so the actual time complexity is usually much lower than the worst case. "DELETE"

2.2.6 bool printConnectedStations(int station_id)

This function prints the IDs of the stations connected to the station with ID station_id. It takes one parameter, station_id, which is the ID of the station whose connected stations are to be printed. Then, it checks if the ID of the station is valid (i.e., between 1 and 50000). If it is not, it throws an IllegalException. Then, it checks if the station is removed. If it is, it returns false because a removed station cannot be connected to any other station. Then, it checks if the graph is empty. If it is, it returns false because there are no edges in the graph. Then, it runs a for loop on the connected edges of the station whose connected stations are to be printed. For each connected edge, it prints the ID of the other station. Then, it returns true.

The time complexity of this function is O(E), where E is the number of edges adjacent to the station with ID station_id. This is because the function simply iterates over the adjacent edges and prints out their IDs, which takes constant time per edge. If the station has no adjacent edges, the function returns immediately, so there is no additional runtime cost. "PRINT"

2.2.7 bool printPrimMST()

This function prints the edges of the MST generated by Prim's algorithm. It takes no parameters. Then, it checks if the graph is empty. If it is, it returns false because there are no edges in the graph. Then, it calls the generateMST() function to generate the MST. Then, it runs a for loop on the edges of the MST. For each edge, it prints the IDs of the two stations connected by the edge, as well as the cost of the edge. Then, it returns true.

Since the function depends on the generateMST() function, the time complexity of this function is the same as the time complexity of the generateMST() function, which is $O((E \log V) + E)$. This is because the function simply iterates over the edges of the MST and prints out their IDs and costs, which takes constant time per edge. If the MST has no edges, the function returns immediately, so there is no additional runtime cost. But since we do not need to optimize and analyze the runtime of printing cost related to MST output, we can ignore the runtime analysis of this function.

Since the for loop is sequential to generation of MST, hence the addition of E is made, but since $O((E \log V) + E)$ is asymptotically same as $O(E \log V)$, we can ignore the addition of E. "MST"

2.2.8 int summateMSTCost()

This function returns the total cost of all edges in the MST generated by Prim's algorithm. It takes no parameters. Then, it checks if the graph is empty. If it is, it returns 0 because there are no edges in the graph. Then, it calls the generateMST() function to generate the MST. Then, it runs a for loop on the edges of the MST. For each edge, it adds the cost of the edge to the sum. Then, it returns the sum.

The time complexity of this function is $O((E \log V) + E)$, where E is the number of edges in the MST. This is because the function simply iterates over the edges of the MST and adds their costs to the sum, which takes constant time per edge. If the MST has no edges, the function returns immediately, so there is no additional runtime cost.

Since the for loop is sequential to generation of MST, hence the addition of E is made, but since $O((E \log V) + E)$ is asymptotically same as $O(E \log V)$, we can ignore the addition of E. "COST"

2.2.9 int getWeight(int station1_id, int station2_id)

This function returns the weight of the edge between two stations. It takes two parameters, station1_id and station2_id, which are the IDs of the two stations. Then, it checks if the IDs are valid. If they are not, it throws an IllegalException. Then, it checks if the IDs are greater than the size of the outer array. If they are, it returns -1 because there is no edge between the two stations. Then, it runs a for loop on the adjacency list of the first station. For each edge in the adjacency list, it checks if the second station is the second station of the edge. If it is, it returns the weight of the edge. If it is not, it continues to the next edge. Then, it returns -1 because there is no edge between the two stations.

The time complexity of this function is $O(E)$, where E is the number of edges in the adjacency list of the first station. This is because the function simply iterates over the edges of the adjacency list and checks if the second station is the second station of the edge. If the adjacency list has no edges, the function returns immediately, so there is no additional runtime cost. "WEIGHT"