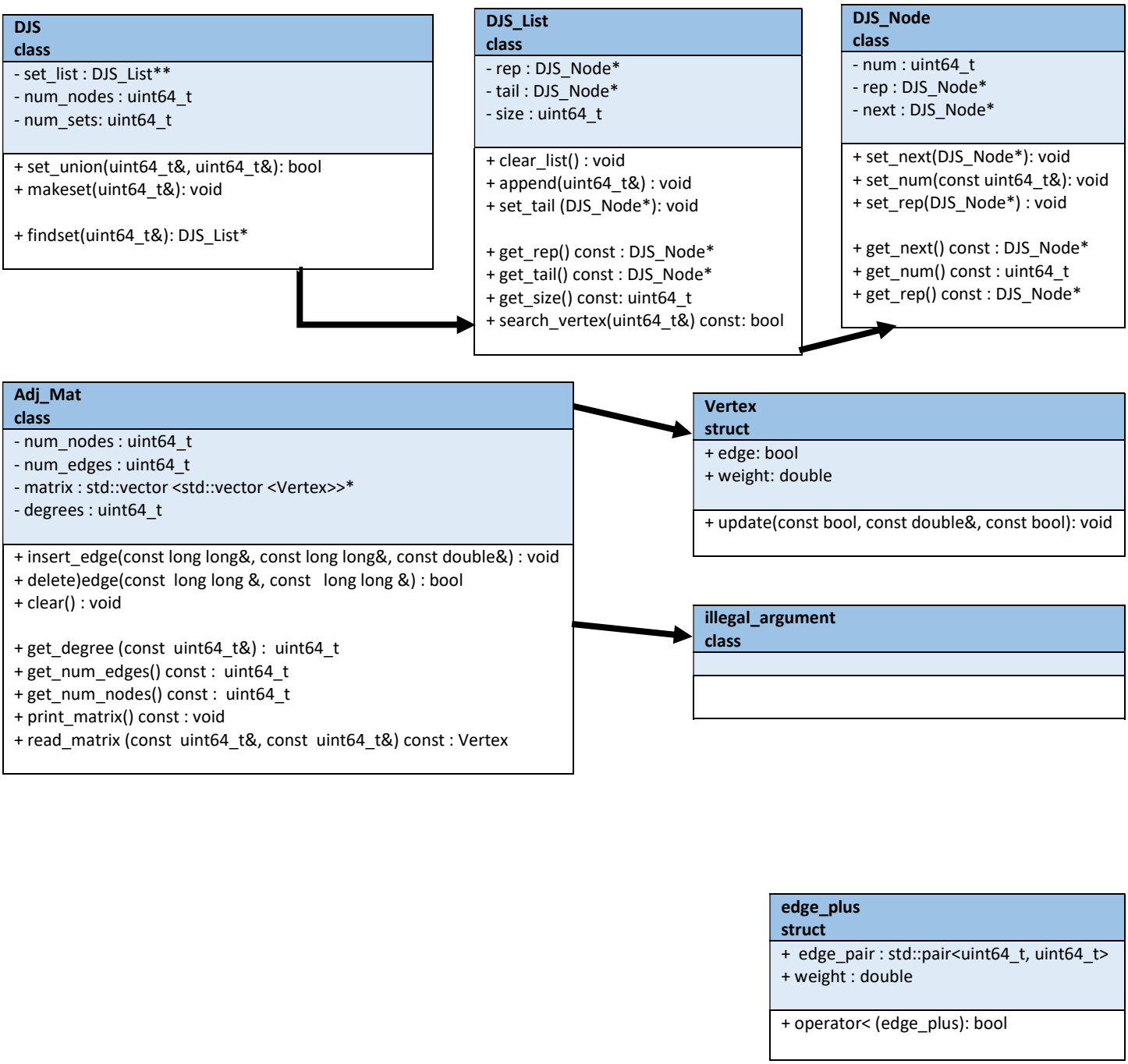
Fall 2020 – ECE 250 Project3 Design Document December 4th 2020 Kyle Dyck

1. Overview of Classes

In this project I designed several classes to implement a graph (as an adjacency matrix) and to determine the minimum spanning tree of a graph using Kruskal's algorithm. Along with the classes used for this algorithm, I also created an empty class illegal_argument for exception throwing / catching. For the graph I designed a class Adj_Mat and a struct Vertex. Adj_Mat contains the matrix (vector of vector of Vertex), the main operations like insert, delete etc. and members such as number of edges, number of nodes, and an array to keep track of the degrees of every node. For calculating the MST, I created a disjoint set (DJS) class with the following: DJS to represent the disjoint set, DJS_List to represent the linked lists in the DJS, and DJS_Node to represent the nodes in the linked lists. Operations include union, makeset, and findset as described in the lecture notes as well as some other methods. Finally, I used a struct edge_plus to hold an edge pair (2 nodes) and the edge weight so that I could use the std::sort algorithm.

2. UML Class Diagram



Fall 2020 – ECE 250 Project3 Design Document December 4th 2020 Kyle Dyck

3. Details and Design Decisions

In general, the type qualifiers go as follows: Most "getters" are const as to not change values, most parameters are const and passed by reference to avoid copying large values.

DJS

- Constructor – One constructor with one argument, that way DJS will always be constructed with a number of nodes which will make it easier to work with later on.
- Destructor – Goes through array of linked lists and deletes every one. Sets pointers to null. Then deletes the array afterwards.

DJS_List

- Constructor – Default constructor that sets everything to 0 or nullptr, then a constructor that takes in a uint64_t and appends the node to the empty list (using the class method append()). This was done to ensure consistency when adding nodes to a list.
- Destructor – Only has to delete rep and set it to nullptr. The destructor of DJS_Node handles the rest.

DJS_Node

- Constructor – Takes an uint64_t that sets "rep" to itself, the "num" to the given number, and next to nullptr. This is to ensure that when creating nodes, each will be distinguishable by their number.
- Destructor – If next is not nullptr, deletes the next node (recursive). This was done so that no holes in the linked list were made. Since this is a singly linked list, losing access to one node will lose the rest of the linked list after it, so making the destructor like this will ensure no leaks.

Adj_Mat

- Constructor – One constructor that takes a long long. Must have number of vertices before creating Adj_Mat so the members can be populated appropriately. This prevents the matrix from being resized but is much simpler; if you need a different size, you have to create a new matrix.
- Destructor – Simply delete matrix and degrees array. The vector calls the destructors of their elements (another vector, and a Vertex which is discussed later) so the matrix is taken care of. Similar thing with the array.

Vertex

- Constructor – Default constructor taking no arguments and setting both member variables to 0. This is to make it easy to use Vertex but also ensuring initialization so random values aren't read in the event that the Vertex is accessed before being given values.
- Destructor – Default destructor, no memory is allocated on the heap due to Vertex so no frees required.

Edge_plus

- Constructor – one constructor taking all arguments to initialize all members and one constructor taking no arguments and setting all members to 0. Again everything is public so no need to require everything at construction but will ensure no random values are accessed if edge_plus gets read before being given values.
- Destructor – Default destructor, no memory in heap due to edge_plus so no frees required.

The following test cases were considered and involved all of these classes / structs:

- Inserting edges with flipped indices
- Inserting edges with weight <= 0
- Multiple inserts called at once
- Multiple deletes
- Inserting / deleting extremes of the matrix
- Successive "n" commands, both valid and invalid
- "n" commands when n = 0 and n < 0
- Degree command with empty graph
- Degree command on nodes 0, < 0, n, >n
- Edge_count command with empty graph and graph of size 1
- D command with empty graph, negative indices, and indices outside range
- Having self-edges in graph and calling MST
- MST with loops in graph
- Replacing edges with different weights

4. Performance Considerations

My implementation of Kruskal's algorithm for determining MST has a running time O(E log E) where V is the number of vertices and E is the number of edges. Sorting the list of edges takes O(E log E) according to the nature of the STL Sort algorithm. Making the disjoint set takes O(V) time since a set must be made for every vertex in the graph. Findset (constant because of the representative nodes) is called E times so will take O(E) time. The cost of union is the cost of combining the smaller linked list into the larger one. This could be linear however as explained in the notes, each time a vertex is moved to a new cluster the size of the cluster containing the vertex at least doubles. So, the total run time for union will be <= O(V log V). The total time will then be O(E log E + V + E + V log V). Considering that the number of edges is greater than the number of vertices, this can be simplified to O(E log E).