Project 4 Design Document

Bill Yao (b29yao)

1. Overview of classes

In this project, I have implemented three classes, vertex, illegal_exception and graph, in order to form a graph data structure to store a weighted graph. In this project, we store researcher IDs and "influence weight" in the class and play around with the several functionalities.

For vertex class, whenever we have an input coming in, we need to consider the start and end vertex, as well as the edge connecting two vertices. Therefore, I implemented three vectors, "adj_vertex", "adj_weight" and "parent_vertex" to hold its adjacent vertices, edges and parent vertices. Three vectors are updated simultaneously so that we can extract corresponding adjacent, parent vertices correctly. Also we have a boolean value "inserted" that determine if the vertex is in the graph. The variable "id" is used for storing the researcher ID number and "m_key" is used to track the weights between vertices when implementing the maximum spanning tree.

For illegal_exception class, this class does nothing but print out a "illegal argument" error message if the input is invalid.

For graph class, we have a dynamically allocated fixed array of size 23133 that contains a fixed number of vertices, as well as two variables "vertex_num", "edge_num" to store the number of edges and vertices. To implement MST, we have a vector heap that stores vertex and a mst vector that stores the mst node. Also we have the following listed functions:

void insertion(int a, int b, double w): given id 1, 2 and the corresponding weight. We need to check if this pair of vertices exists in the graph. If not, we push id_2 to the adjacency_vertex vector of the id_1, and weight into the adjacency_weight vector. Also, we want to push id_1 into the parent_vertex of id_2. Then we incremental the vertex number and edge number based on the condition if two vertices are already in the graph.

void print(int a): given the id position, we directly print the adjacency list of the vertex out based on the order they were pushed in.

void deletion(int a): given the id position, we first delete the vertex from the array, and empty its corresponding adjacency vector and weight vector. Also, we need to find its parent vertices, and delete this vertex from its' parents' adjacency vector to prevent it is still accessibly after deletion.

void getMST(int a): apply the prime algorithm to find the mst from the given position. We first push back the vertex to the heap and start doing heapify. By pushing the max node into the mst vector, we eventually return the mst vector size.

int getSize(): directly return the number of the vertex in the graph by printing "vertex_num" variable stored in the graph class.

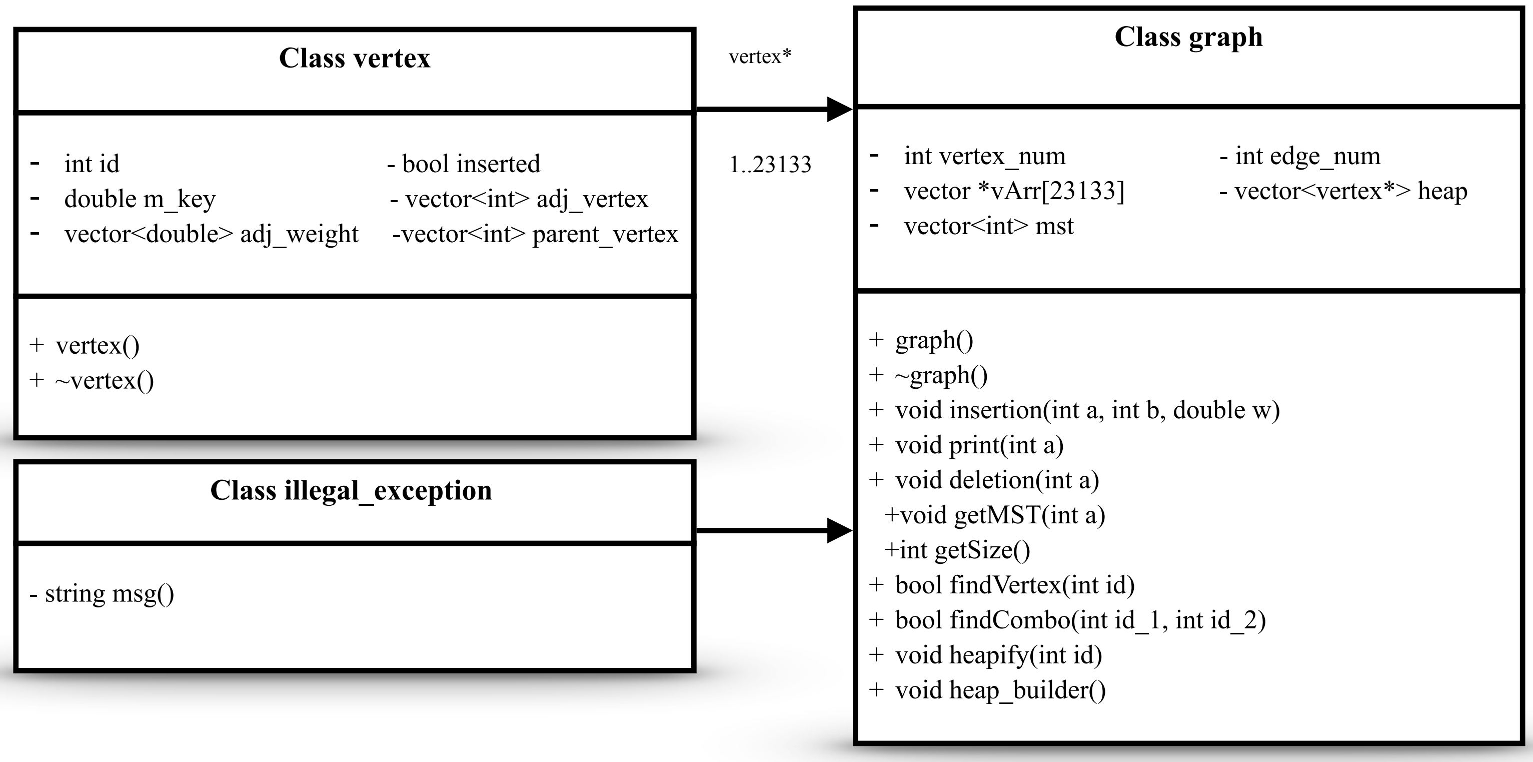
bool findVertex(int id): given the id, find if the vertex exists in the corresponding position in the fixed array. If not, return false. Otherwise, return true.

bool findCombo(int id_1, int id_2): given two ids, find the id_1 position in the array and check if id_2 is in its' adjacency vector. If yes, this edge exists. Otherwise, return false.

void heapify(int id): left_hand is 2*i + 1 and right_hand is left_hand +1. If the left_hand is larger than the parent, then we set the largest to be the left side.

void heap_builder(): within the heap vector, we loop from the middle of heap size down to 0, and we do the heapify on the current index.

2. UML class diagram



3. Design Decisions

For **illegal_exception** class, we only have a public string that prints out an "illegal argument". Therefore we don't need to use any constructor or destructor in the class.

For **vertex** class, we need to initialize the member variables in the constructor, and resize the three vector size to be 0. Then in the destructor, we need to clear three vectors from the vertex. There are no operators that we used to override, nor did we use any pass by reference on any parameters since they are constantly taking in new data as the input is being pushed in and popped out.

For **graph** class, we also need to initialize member variables in the constructor, and for each vertex in the fixed array, we dynamically allocate them. In the destructor, we free every single vertex we use in the array and delete the pointer. Again, we did not use any operators or pass by reference on parameters.

4. Testing Strategy

Test1: we can test insertion and deleting function together. For example, when we enter a line (1 2 0.5) for insertion, we can reenter this line again, see if a “failure” message pop out to ensure same edges wouldn’t be inserted repeatedly. Once we input several edges into the graph, we can delete a vertex and use print function to check if the adjacency vertex is still printed at this location. Also we can print adjacency vertices at other vertices to see if the deleted vertex is removed completely.

Test2: once we start inserting and deleting vertices from the graph, we can call the size function to check if every vertex is inserted or deleted. Also, we wanna make sure that inserted vertices wouldn’t be inserted again in the graph with size function.

Test3: we can load the data set into the graph and try mst function with some vertices. Once we finish that we can delete vertices from the graph and retry the mst function to see if the mst size is still the same.

Test4: we can input some invalid lines into the program such as (-1 1 0.5) or (1 2 1.2) to check if the illegal argument is correctly called.

5. Performance considerations

For **insertion** function, we first need to search the neighbour of the vertex and determine if there is position to insert in the graph, and the worst expected running time is **O(n).** If the vertex is already in the graph, we only need to insert the edge into the vector, therefore the best expected running time is **O(1)**.

For **print** function, we need to print the adjacency list of the target vertex based on the order they were inserted. The expected running time is associated with the number of adjacency vertex from the node. Therefore, the running time would be **O(degree(a))**.

For **deleting** function, we need to linearly delete all the edges associated with the vertex. If the vertex only has one edge with other vertex, the best running time is **O(1)**. Otherwise, the expected running time would be **O(n)**.

For **mst** function, by applying the prim algorithm, we implemented a head data structure in the project. The expected running time is The worst case of running time will be V* T(extract) + O(E). For binary heap, the extract time and modify key time will be O(lgV). Thus the running time will be O(V lgV + E lgV) = **O(E lgV)**

For **size** function, we directly print the vertex_num variable that stores the number of vertices. Therefore, the running time is **O(1)**.