COL 106 Assignment 6

This assignment is based on the graph data structures. In this assignment, I have implemented a graph consisting of triangles. To implement this graph, I have used only Array Lists. The Array Lists store the points, edges, and triangles of the graph.

Here is the explanation of the various classes I have implemented and their properties:

Class ArrayList:

1. I have implemented my own generic array list which I will use for storing Triangle, Edges, and Points. Here I am using Object in place for writing. My array list consists of functions like getting (int index){which returns the element at the position index of the list}, set(int index, Object obj){which sets the value of obj at the index passed}, add(Object obj){which adds the element at the end of the list}. Similarly, I have written functions like add_uniquely(Object obj) which adds the element only when it does not exist already. Add_triangle(), add_point(), add_edge() adds triangles, points and edges in the sorted order respectively. So in this way, I don't need to implement sorting separately. Initially I have created an array of fixed size, but when the elements exceed the limit, then I just double the size of the array.

Class Edge:

1. In the Edge object, I am storing its endpoints source and destination as Point objects. I am also storing an array list of the face neighbors of the edge which is actually a list of triangles connected to that edge. Edge is compared on the basis of its length. Two edges are equal only when their endpoints are the same. I will use all these storage during add triangles.

Class Point:

1. For point objects, I am storing its three coordinates in it. I am also storing its edge neighbors, point neighbors and face neighbors in it which will be used in queries.
2. A point is compared on the basis of its X coordinates first, then Y and then Z.

Class Triangle:

1. In this, I am storing its vertices, edges, edge neighbors and point neighbors.
2. Two triangles are compared on the basis of their count. The triangle which is created first will have low count and will be displayed first in the printing order.

Class Shape:

This is the most important class which handles all the queries.

1. We are maintaining three global lists: **ArrayList<Point> all_points, ArrayList<Triangle> all_triangles, ArrayList<Edge> all_edges**.

2. In the query **ADD_TRIANGLES**, we have to first check whether the coordinates given are collinear or not. If collinear, then we will do nothing. But in the other case, we will create a triangle with the coordinates given. For this, first, we will check whether we have those points already present in the list or not. If already present, then we will just use that point as a vertex of the new triangle instead of creating that point again. Similar check for all the other points. Now we will check the three edges in the edges list and will create a new edge only if it is not found in the list. If found the new will just give the triangle the reference of the existing edge. This way I ensure that I do not create an edge or a point twice in the graph. Now after this, I will update all the information in the points and the edges. I will update the point neighbors and face neighbors in the edges. I will update face neighbors, edge neighbors and point neighbors in the points. This way my graph will get updated without any duplicates.

3. In the query **TYPE_MESH**, if the number of faces plus the number of vertices minus number of edges is 2, then the mesh will be a closed mesh. If any edges have more than two triangles connected in it, then the mesh will be an improper mesh. In other cases, the mesh will be a semi proper mesh.

4. In the query **BOUNDARY_EDGES**, we will return all the edges which have only one triangle connected to it. We will do this by checking the number of triangles connected to each edge and store those which satisfy the boundary edge condition. The edges will be sorted on the basis of their edge length.

5. In the query **NEIGHBORS_OF_TRIANGLE**, first, we will search the triangle in the all_triangles list. If found, then only we will proceed further otherwise return null. If proceeded further, then we will take the edges of the triangle one by one and then inserting all the face neighbors of the edges but the given triangle will be not inserted.

6. In the query **EDGE_NEIGHBOR_TRIANGLE**, we will return the edges of the triangle simply.

7. In the **VERTEX_NEIGHBOR_TRIANGLE**, return all the vertices of the triangle.

8. In the query **EXTENDED_NEIGHBOR_TRIANGLE**, we will first check for the triangle. If found, then only we will proceed. We will take its vertices one by one and take all the face neighbors of the points but not the given triangle. Finally, return the array consisting of the elements of the list used to insert the extended neighbors.

9. In the query **IS_CONNECTED**, we will check whether the given triangles are connected or not. For this, we will first do Depth First Search on the first triangle if found and then see whether the second triangle got visited or not. If it is visited, then both are connected otherwise not.

10. **INCIDENT_TRIANGLES** query is the same as the FACE_NEIGHBORS_OF_POINT query described below.

11. In the query **NEIGHBORS_OF_POINT**, we will return the list of the points which are the neighbors of the given points. But before doing this, we will check whether this point exists or not. I not exist, then we will return null otherwise proceed further.

12. The query **EDGE_NEIGHBORS_OF_POINT** is the same as query no. 11, but here we will return the edge neighbors in place of point neighbors.

13. in the query **FACE_NEIGHBORS_OF_POINT**, we will return the list of all the triangles which we have stored in the given point if it is found in the list of all points. Otherwise, return null.

14. in the query **TRIANGLE_NEIGHBOR_OF_EDGE**, we will first search the given edge in the list of all edges. If not found, return null. Otherwise, return the face neighbors of the given edge.

15. In the query **MAXIMUM_DIAMETER**, we will find the diameter of the graph. For this, we take any triangle and will do the Breadth-First Search over it. This will cover all the triangles of the component in which the given triangle lies. In the BFS, we first visit a node and then inserts it in an array list. Then we insert a null to separate one level by the other. Then we start dequeuing the elements one by one and adding the adjacent triangles of one triangle dequeued. When we again reach null, that means we have completed the visit of one level. This way when we complete BFS then we will know the depth of the graph starting from the given triangle. We will then store it in a temporary variable. Now we will do the same process with every triangle and find the value of the number received. The maximum of these values will be the diameter. But this is the case when we have only one component. If we have more than one component, then we will find the diameter for each component and the diameter of the graph will correspond to that component which has the maximum number of triangles.

16. In the query **CENTROID**, we will first find all the connected components by using DFS explained above and then for each connected components, we will extract all the points of that component and find their centroid as specified. Then return the array containing all the centroids.

17. In the query **CENTROID_OF_COMPONENT**, we are given a point. First we will find that point in the points list, if found then we proceed further otherwise return null. When we found the point, then we will find the component which contains that point by DFS. Then taking all the points of that component, we will find the centroid.

18. In the query **CLOSEST_COMPONENTS**, we will start by DFS. By this we will get the list of all the components. After this, we will apply the Brute force method to find the distance between the two closest components. If the number of components is only 1, then return null. Otherwise, taking two components at a time and take each possible pair of points one from one component and other from others, and then compute the distance between them. Store both points in an array for which the distance is minimum.