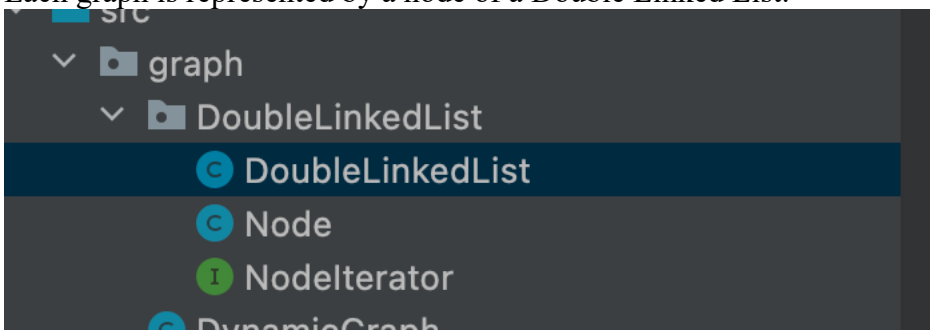
REPORT:
PART 1.
User defined parameter: Label(string)

```
DynamicGraph<String, String> dynamicGraph = new DynamicGraph<>(DynamicGraph.directed);
System.out.println("Adding vertexes A,B,C,D,E");
Vertex<String, String> vertex0 = new Vertex<~>( data: "A", id: 0, weight: 1.0, boosting: 2);
System.out.println("A is added: id: 0, weight: 1, boosting: 2");
Vertex<String, String> vertex1 = new Vertex<~>( data: "B", id: 1, weight: 1.0, boosting: 3);
System.out.println("B is added: id: 1, weight: 1, boosting: 3");
Vertex<String, String> vertex2 = new Vertex<~>( data: "C", id: 2, weight: 1.0, boosting: 0);
System.out.println("C is added: id: 2, weight: 1, boosting: 0");
Vertex<String, String> vertex3 = new Vertex<~>( data: "D", id: 3, weight: 1.0, boosting: 3);
System.out.println("D is added: id: 3, weight: 1, boosting: 3");
Vertex<String, String> vertex4 = new Vertex<~>( data: "E", id: 4, weight: 1.0, boosting: 1);
System.out.println("E is added: id: 4, weight: 1, boosting: 1");
Vertex<String, String> vertex5 = new Vertex<~>( data: "F", id: 5, weight: 3.0, boosting: 2);
System.out.println("F is added: id: 5, weight: 3, boosting: 2");
```

I implemented the Graph interface in this section. My main vertex classes name is DynamicGraph. We can add new vertices into graph by using constructor.
There are id,weight and boosting attirbutes for each vertex.
Each graph is represented by a node of a Double Linked List.



Data source of the graphs are double linked list. When we are iterating trough each vertice, we use NodeIterator.

Adding new vertices:

```
 */
public Vertex<E, T> addVertex(Vertex<E, T> vertex) {
    vertexLastList.add(vertex);
    Node<Vertex<E, T>> node = vertexList.add(vertex);
    vertex.setPosition(node);
    return vertex;
}
```

For us to add new vertices, we use and edge :
We use doubleLinkedlist's add function for adding the new vertice and edge.

```
private DoubleLinkedList<Vertex<E, T>> vertexList;
private DoubleLinkedList<Edge<E, T>> edgeList;
```

```java
 */
public Node<E> add(E data) {
    Node<E> node = new Node<E>(data);

    // If list is empty, add to the head
    if (size == 0) {
        head = node;

        // If list is not empty, add to the tail
    } else {
        tail.next = node;
        node.previous = tail;
    }

    // Adjust the tail and size
    tail = node;
    size++;
    return node;
}
```

Export matrix function creates the adjacent matrix .

```java
dynamicGraph.printGraph(dynamicGraph.exportMatrix());
```

```java
System.out.println("Trying newVertex function: ");
System.out.println("Adding new vertex with label F");
System.out.println(dynamicGraph.newVertex( label: "F",  weight: 2).toString());
```

newVertex generates a new vertex by given parameters and return that vertex.

I use weight and label parameters fort his.

```java
System.out.println("Trying filter graph function with key value label: ");
System.out.println("Return graph"+ dynamicGraph.filterVertices( key: "F", filter: "label").toString());

System.out.println("Trying filter graph function with key value id: ");
System.out.println("Return graph"+ dynamicGraph.filterVertices( key: "2", filter: "id").toString());
```

filterVertices function can be used with two parameters: id and label.
When we give that parameters, function filters the vertices by searching it in:

```java
private DoubleLinkedList<Vertex<E, T>> vertexList;
```

Remove and add functions:

```
System.out.println("Removing the vertex F with id");
dynamicGraph.removeVertex( id: 4);
output = "Vertices:\n";
for (Vertex<String, String> v : dynamicGraph.vertices_array())
    output += String.format("%s ", v.toString());
System.out.println(output);
System.out.println("Adding the vertex F again");
dynamicGraph.addVertex(vertex5); // id= 5
output = "Vertices:\n";
```

We can remove by id or label.

```
for (Vertex<String, String> v : dynamicGraph.vertices_array())
    output += String.format("%s ", v.toString());
System.out.println(output);
System.out.println("Removing the vertex F with label");
dynamicGraph.removeVertex( label: "F");
```

There is two printGraph function which is overwritten, one of them takes the exportMatrix function's return value and prints it other one finds the adjacency list and print the list.

The second one is:

```
/**
 * prints the adjacent matrix
 * @param adj adjacent matrix
 */
public void printGraph(ArrayList<ArrayList<Integer>> adj) {
    for (int i = 0; i < vertexList.size(); i++) {
        System.out.println("\nAdjacency list of vertex" + i);
        System.out.print("head");
        for (int j = 0; j < adj.get(i).size(); j++) {
            System.out.print(" -> " + adj.get(i).get(j));
        }
        System.out.println();
    }
}
```

Adj is a vertice arraylist of arraylist which holds the id of the vertex1 and vertex2.

```
dynamicGraph.printGraph(dynamicGraph.exportMatrix());
System.out.println("Testing the Adjacency List printGraph fucntion");
dynamicGraph.printGraph(dynamicGraph.getAm());
```

We can add new edges.Add function takes two vertices and weight.

```
System.out.println("Adding edges:");
System.out.println("Adding first edge from A to B with weight 10");
dynamicGraph.addEdge(vertex0, vertex1,  label: "1'th edge",  weight: 10);
System.out.println("Adding second edge from A to C with weight 3");
dynamicGraph.addEdge(vertex0, vertex2,  label: "2'th edge",  weight: 3);
System.out.println("Adding third edge from B to D with weight 2");
dynamicGraph.addEdge(vertex1, vertex3,  label: "3'th edge",  weight: 2);
System.out.println("Adding fourth edge from B to C with weight 1");
dynamicGraph.addEdge(vertex1, vertex2,  label: "4'th edge",  weight: 1);
System.out.println("Adding fifth edge from C to B with weight 4");
dynamicGraph.addEdge(vertex2, vertex1,  label: "5'th edge",  weight: 4);
System.out.println("Adding sixth edge from C to D with weight 8");
dynamicGraph.addEdge(vertex2, vertex3,  label: "6'th edge",  weight: 8);
System.out.println("Adding seventh edge from C to E with weight 2");
dynamicGraph.addEdge(vertex2, vertex4,  label: "7'th edge",  weight: 2);
System.out.println("Adding eighth edge from D to E with weight 7");
dynamicGraph.addEdge(vertex3, vertex4,  label: "8'th edge",  weight: 7);
System.out.println("Adding nighth edge from E to D with weight 9");
dynamicGraph.addEdge(vertex4, vertex3,  label: "9'th edge",  weight: 9);
System.out.println("Adding nighth edge from E to B with weight 4");
dynamicGraph.addEdge(vertex4, vertex1,  label: "10'th edge",  weight: 4);
```

BFS and DFS traversels:

```
System.out.println("Applying BFS to graph:");
for (Vertex<String, String> v : dynamicGraph.BFS())
    System.out.print(v + " ");

System.out.println("\n\nApplying DFS to graph:");
for (Vertex<String, String> v : dynamicGraph.DFS())
    System.out.print(v + " ");
```

DFS steps:
1. Configure Graph options
2. Mark all vertices as unvisited and uncolored
3. Mark all edges as undicovered
4. Start DFS
5. +1 disconnected graph, trigger connection detection
// Color all vertices with the same color for each vertex start ((v0-> v1) <- v2) [for DiGraph]

Iterate on all neighbors of the current vertex

Recur on neighbor if not visited
Checks if the undirected/directed graph is cyclic
Mark edge as cross if the undiscovered
Mark vertex as visited if more neighbors needs to be visited

BFS steps:

1. Mark all vertices as unvisited
2. Mark all edges as undiscovered
3. Start BFS
4. Add the starting vertex and mark it as visiting
5. Remove a vertex from the queue and mark it as visited
6. Iterator on all neighbors of the removed vertex and add them to the queue
7. If neighbor is not already visited, put it in the queue
8. If neighbor has already been visited, don't put it in the queue
9. Mark edge as cross if undiscovered

There are some helper functions for both DFS and BFS.

BFS_helper:
This function applies BFS for detecting connected components and is connected in DiGraphs
Purpose behind this function is to consider the DiGraph as UnDiGraph by concatenating the in and out edges

PART 2:

```
System.out.println("Part 2 - Calculating the difference between DFS and BFS algorithm paths:");
dynamicGraph.calculateDFSandBFSNumbers();
dynamicGraph.printGraph(dynamicGraph.getAm());
System.out.println(dynamicGraph.getBfsnumber());
System.out.println(dynamicGraph.getDfsnumber());
```

getBFSnumber and getDFSnumber functions do return the value of the minumum number of iterations in that algorithm as we can see at below picture:

```
        Edge<E, T> edge = incidentEdges.next();
        Vertex<E, T> oppositeVertex = edge.getV2();

        // If neighbor is not already visited, put it in the queue
        if (oppositeVertex.getStatus() == Vertex.UNVISITED) {
            bfsnumber++;
            // Mark edge between the removed vertex and the current neighbor as discovered
            edge.setStatus(Edge.DISCOVERED);
            oppositeVertex.setStatus(Vertex.VISITING);
            q.offer(oppositeVertex);

            // If neighbor has already been visited, don't put it in the queue
        } else {
```

When an unvisited not is found and if the path on that time is the shortest path, we incrementing it by 1. Same goes in DFS as well.

```
dynamicGraph.calculateDFSandBFSNumbers();
```

This function finds the difference between that numbers.
Time complexity is same as DFS and BFS since we are doing the same traversel here, we just only incrementing the counters.

PART 3:
Dijkstra algorithm works as below:
1. Mark all vertices as unvisited and reset Dijkstra options
2. Mark all edges as undiscovered
3. Mark the starting vertex
4. Create the Priority Queue (Using a heap)
5. Start from the starting vertex by putting it in the Priority queue
6. Remove the vertex with minimum Dijkstra value
7. Put all the neighbors of the removed vertex in the Priority queue and adjust their Dijkstra value and parent
8. If the neighbor has not been visited, mark it visiting and adjust its configuration
9. If the neighbor is still in the priority queue, check for minimum path cost, adjust if the cost can be reduced

This algorithm basically creates the shortest path from a vertex to all other vertices.

```
while (incidentEdges.hasNext()) {
    Edge<E, T> edge = incidentEdges.next();
    Vertex<E, T> oppositeVertex = edge.getV2();
    double pathCost = edge.getWeight() + polled.getDijkstra_value();
    System.out.println("Current vertex is:" + oppositeVertex.getLabel());
    System.out.println("Current Path Cost is: " + pathCost);
    System.out.println("Boosting value of " + oppositeVertex.getLabel() + "is " + oppositeVertex.getBoosting()
    System.out.println("Apply the boosting");
    pathCost -= oppositeVertex.getBoosting();
    System.out.println("After boosting, current path cost is: ");
    System.out.println(pathCost);
    // If the neighbor has not been visited, mark it visiting and adjust its configuration
    if (oppositeVertex.getStatus() == Vertex.UNVISITED) {
```

When we apply the algorithm, we are finding the cost of the edge(weight) and deleting the current vertice's boosting value during traversel.

Time Complexity of the Functions:

addEdge(...):
Array of the added edges
Add one edge between two vertices. Add another edge in the opposite direction if the graph is undirected
O(1)
addVertex(vertex):
Vertex added
Add a vertix to the graph
O(1)
BFS():
Array of vertices traversed by BFS
Traverse the graph with Breadth First SearchO(|V| + |E|)
BFS(vertex):

Array of vertices traversed by BFS
Traverse reachable vertices in a graph with Breadth First Search starting from a specific vertex O(|V| + |E|)
DFS():
Array of vertices traversed by DFS
Traverse the graph with Depth First Search O(|V| + |E|)
DFS(vertex):
Array of vertices traversed by DFS
Traverse reachable vertices in a graph with Depth First Search starting from a specific vertexO(|V| + |E|)
dijkstra(v):
Trace the shortest path from v to all other vertices
O(|V|log|V|+ |E|)
edges():
NodeIterator
Gives an iterator on the list of edges
O(1)
vertices():
NodeIterator
Gives an iterator on the list of vertices
O(1)
edges_array():
Array of edgesGives an array of all the graph edges
O(|E|)
vertices_array():
Array of vertices
Gives an array of all the graph vertices
O(|V|)