Tutorial #13. Graph representation

Theoretical part

Graphs are special data structures that come not from algorithms (as hash tables and trees do – they solve the problem of efficiency for the same task), but from real-world/mathematical problems. Graphs are very good at describing problems, *modelling real-world cases*, where something (or someone) can "travel" between base points in unpredicted way. E.g. tourist is moving from town to town, electron is traveling over the circuit, TCP-frame can travel though internet, your work activity is travelling from one task to another, etc. Having this as a model, we can answer common questions about these journeys:

- *What is a **shortest path** for [tourist/TCP-frame/worker/…] to get from A to B in terms of [time/resistance/hops/…]? (shortest path)*
- *How can […] **visit all the points** in the fastest way? (travelling salesman problem)*
- *What is the **minimal number of** [roads/links/…] can remain to preserve connections between base points? (spanning tree)*
- *How many **different** […] can you make, that **none of the neighbors will have the same**? (coloring)*
- *How can I arrange my visits to be sure that some points will be accessed always after other points? (concerts, scheduling … - topological sort)*

As you can see, these questions can be asked in multiple practical cases, and answers for them are **existing graph-based algorithms**.

To solve using graphs, you should be able to do 4 things:

1) Understand graphs.
2) Formalize you problem domain as a graph (special type of graph: oriented, not oriented, weighted, complete, connected, …).
3) Implement graphs.
4) Know that there are graph-based algorithms, which can solve your problem.

Edge list structure

You graph data structure contains 2 lists: one is a list of vertices, another – list of edges. Why do we need both:

Graph can be not connected, but we still need a tool to visit all nodes and edges in O(n) time (e.g. you update road tax, or recalculate flight prices, …). Otherwise we will not be able to reach graph parts.

This is a suggested graph class:

```java
import java.util.*;

public class Graph<TDataValue, TWeight> {

    private List<Vertex> vertices = new ArrayList<>();
    private List<Edge> edges = new ArrayList<>();

    public class Vertex {

        TDataValue value;
        int listPosition = -1;

        public Vertex(TDataValue value) {

        public TDataValue getValue() {

        public List<Vertex> adjacent() {
    }

    protected class Edge {

    public boolean addVertex(Vertex vertex) {

    public boolean removeVertex(Vertex vertex) {

    public void addEdge(Vertex from, Vertex to, TWeight weight) {
        Edge e = new Edge(from, to, weight);
        edges.add(e);
        e.listPosition = edges.size() - 1;
    }

    public boolean RemoveEdge(Vertex from, Vertex to) {
}
```

Adjacency list structure

We can add neighborhood lists to make `adjacent()` calls faster (scan not all the collection, but only corresponding edges).(*See* `VertexExtended.incidents` *field*).

NB **Edge** class should also be **extended** to store self-positions in both lists of origin and destination.

```java
import java.util.*;

public class Graph<TDataValue, TWeight> {

    private List<Vertex> vertices = new ArrayList<>();
    private List<Edge> edges = new ArrayList<>();

    public class Vertex {

        TDataValue value;
        int listPosition = -1;

        public Vertex(TDataValue value) {[]

        public TDataValue getValue() {[]

        public List<Vertex> adjacent() {[]
    }

    public class VertexExtended extends Vertex {

        protected List<Edge> incidents = new ArrayList<>();

        public VertexExtended(TDataValue value) {[]

        public List<Vertex> adjacent() {[]
    }

    protected class Edge {[]

    public boolean addVertex(Vertex vertex) {[]

    public boolean removeVertex(Vertex vertex) {[]

    public void addEdge(Vertex from, Vertex to, TWeight weight) {
        Edge e = new Edge(from, to, weight);
        edges.add(e);
        e.listPosition = edges.size() - 1;
    }

    public boolean RemoveEdge(Vertex from, Vertex to) {[]
}
```

You can consider special cases of trees to make them more lightweight. E.g. non-weighted graph does not require to have specific edge objects (you can use adjacent collections as you do for trees).

Adjacency matrix structure

Adjacency matrix is the best one in terms of access speed – You can find nodes and edges in O(1) time, but they case redundant memory consumption for spare graph cases. Consider AMS as *edge list structure* + array containing connections for faster access:

```java
private ArrayList<Edge> nodeConnections = new ArrayList<>();

private Edge getEdge(Vertex from, Vertex to) {
	return nodeConnections.get(
			vertices.size() * from.listPosition + to.listPosition
			);
}
```

For lightweight implementation, you can keep only weight inside the cell (not edge). Also, think how you will delete and add nodes to DS containing such a matrix.

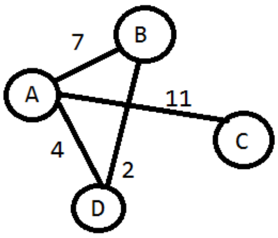
Practical part (for Students)

1) Implement adjacency list structure with methods:
    a. Add vertex
    b. Remove vertex
    c. Add edge
    d. Remove edge
    e. Get adjacent vertices
2) Write a code that can fill your graph from the following file:

```
A B C D E F G H Kolya Vasya
A B 5 D Kolya 1 G Kolya 5 Kolya Vasya 12 F B 7
```

where first line contains node names separated by spaces, and second line contains triplets of ***origin****,* ***destination*** *and* ***weight*** *for egdes.*

E.g.



A B C D

A B 7 A D 4 B D 2 A C 11

3) Write the code that will serialize graph in the same way.