01

Undirected Graphs

02 Undirected Graphs

So, in this project, We are going to talk about a lot of topics related to the Undirected Graphs. And below are the topics that we have covered in this project:

03 Definition

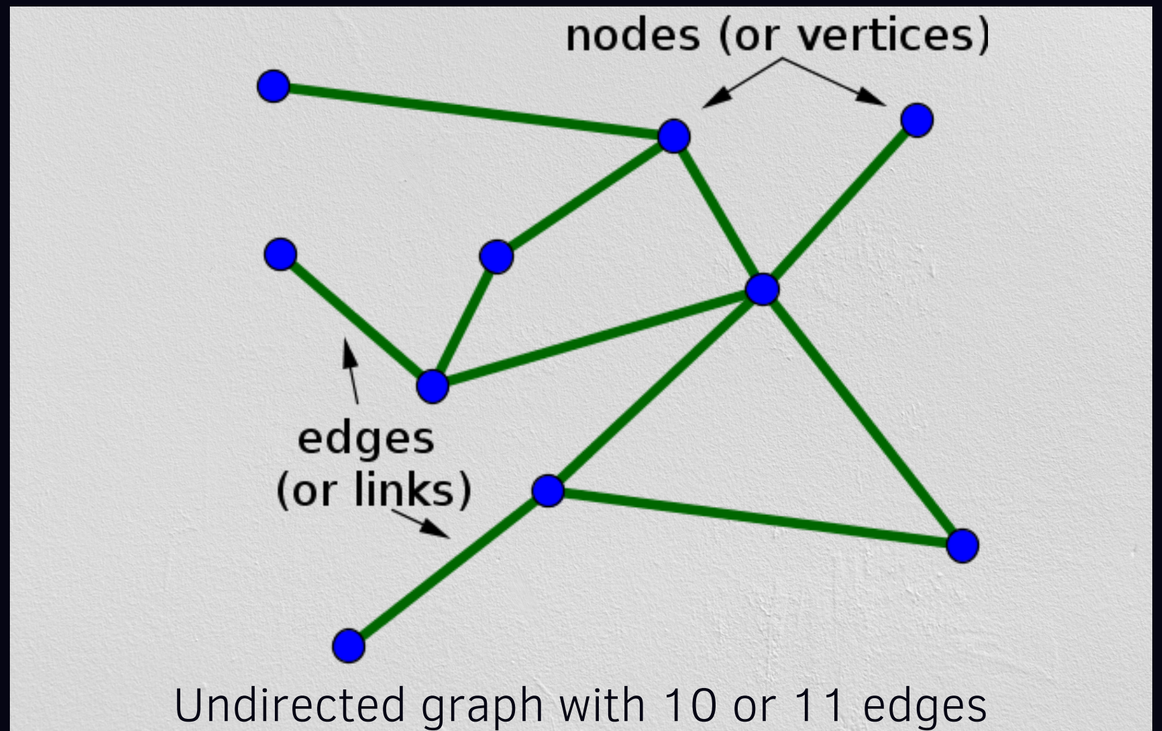
The definition of Undirected Graphs is pretty simple:

Set of vertices connected pairwise by edges.

Graph definition

Any shape that has 2 or more vertices/nodes connected together with a line/edge/path is called an undirected graph.

Below is the example of an undirected graph:



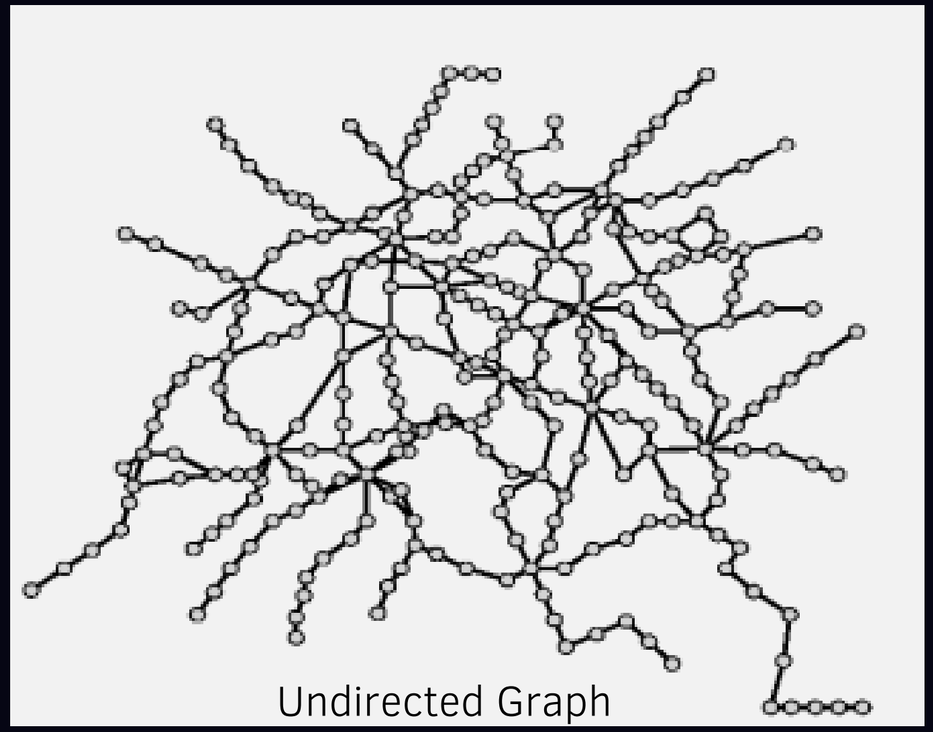
Undirected graph with 10 or 11 edges

04

Vertices are the result of two or more lines intersecting at a point. Edges or Links are the lines that intersect.

These graphs are pretty simple to explain but their application in the real world is immense. You will see that later in this presentation.

Here’s another example of an Undirected Graph:



Undirected Graph

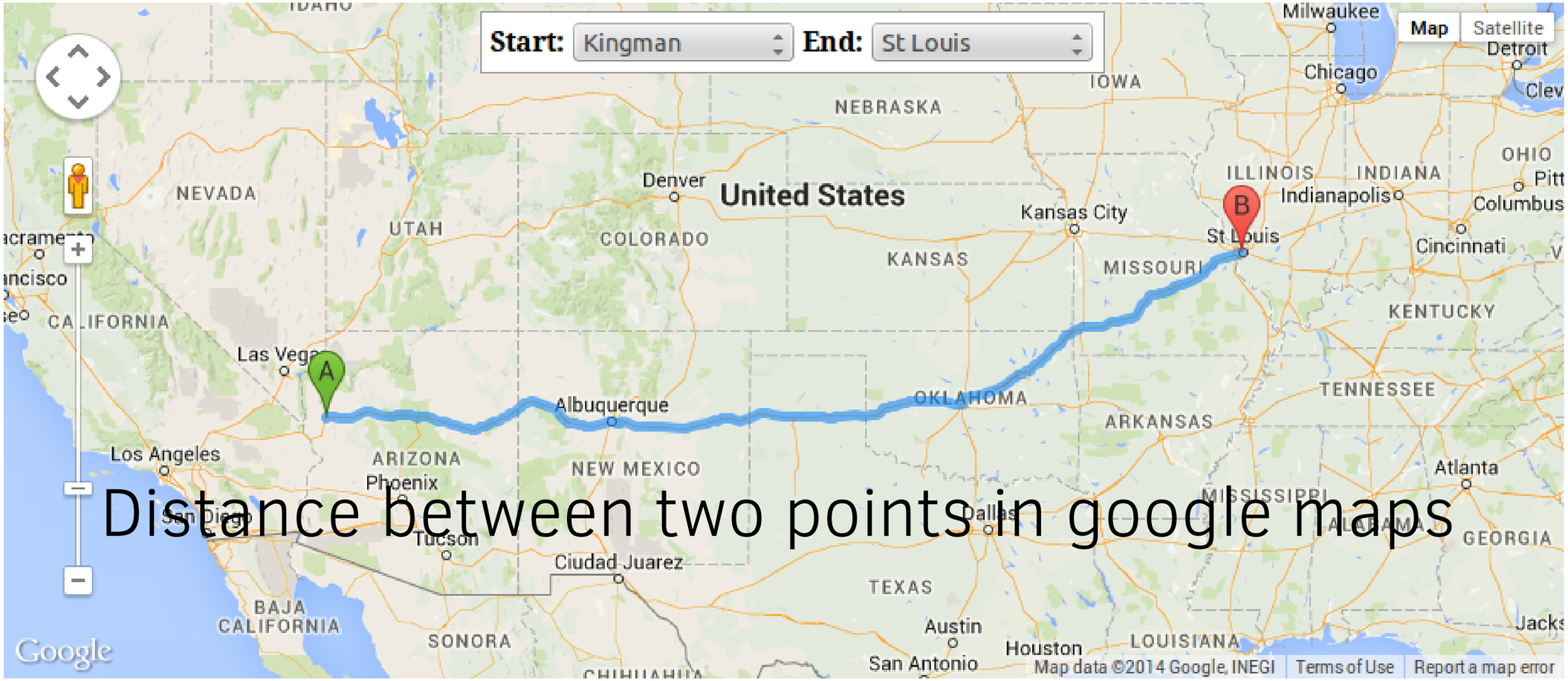
05

Why Study Graphs?
Application of given theory

As we said, there are thousands of practical applications of Undirected Graphs . And being an Information Systems student, it’s our job to study about these applications and try to make it more efficient.
As of today, there are thousands of graph algorithms that are used for different purposes. For example – Google map uses some of the graph algorithms to find the shortest distance between two points on Google Maps.

06

Directions service



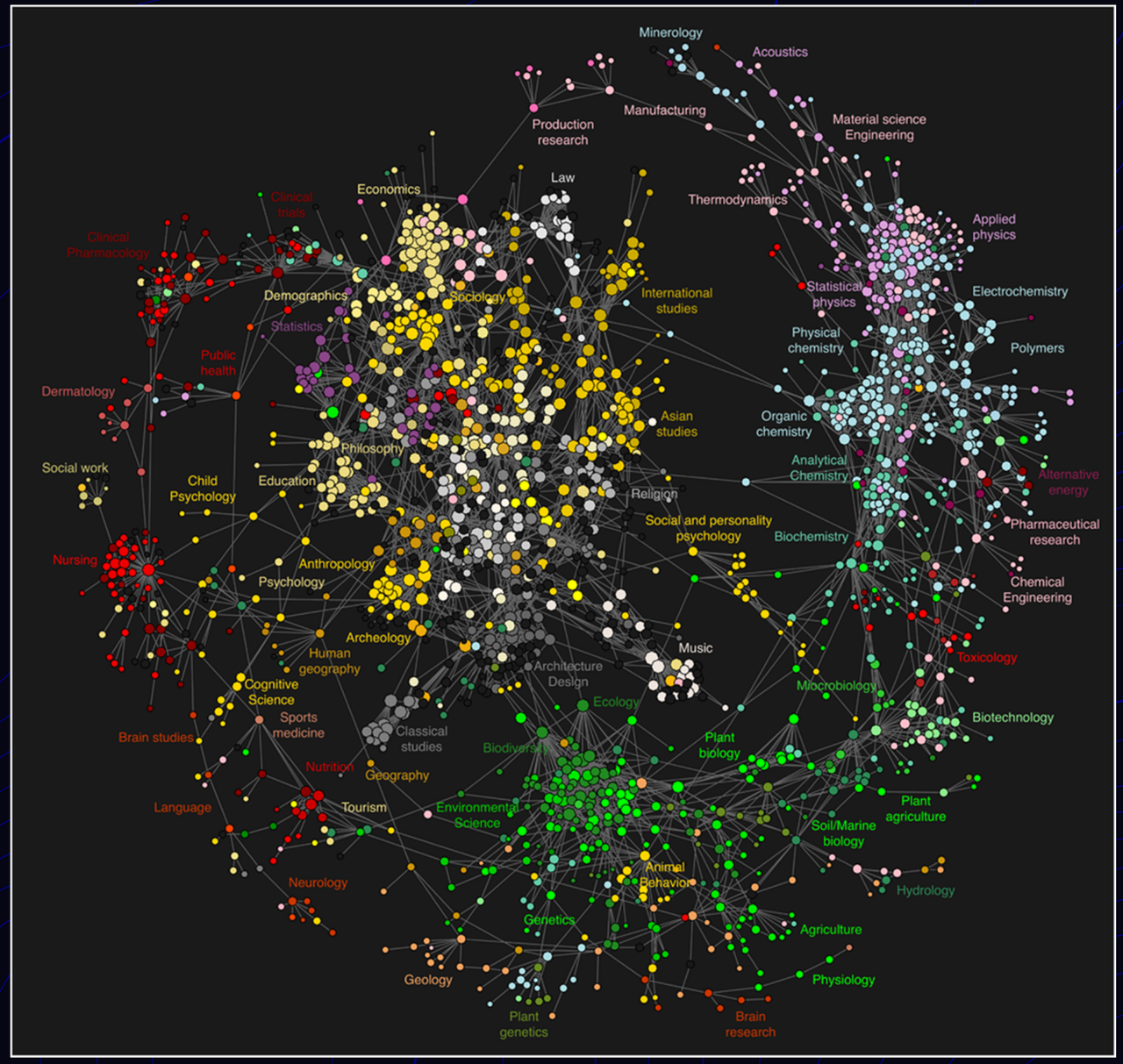
Distance between two points in google maps

07

Many businesses today such as Ola, Uber, Grab etc... relies on the accuracy of these maps. Their entire business model revolves around it. Smallest miscalculation would cause them thousands of bucks.
All of this is made possible just with the help of Graph Algorithms.
Well, let us share with you some more real-world applications of Graphs and the algorithms that made it possible.

08

09



MAP OF SCIENCE CLICKSTREAMS

10

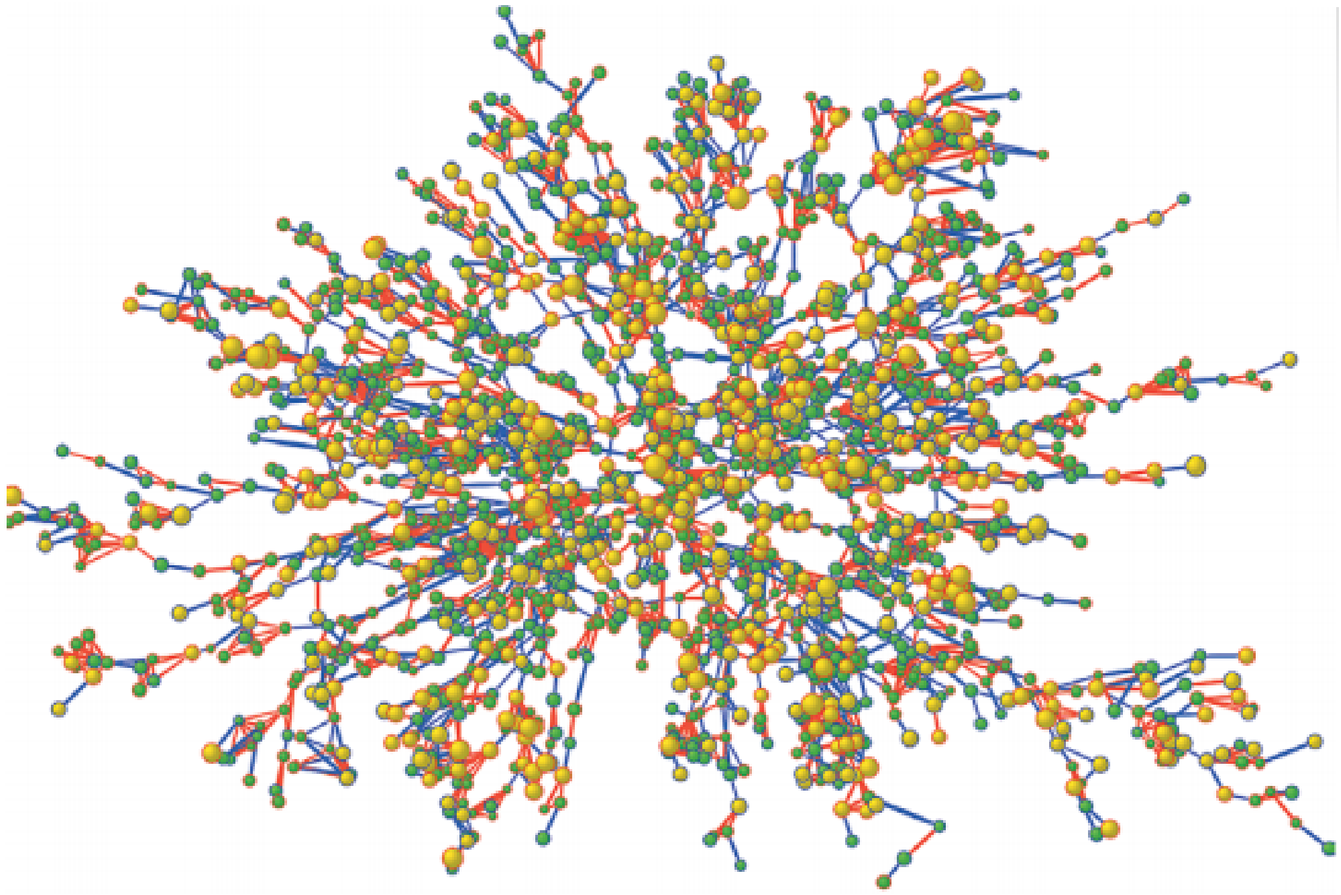
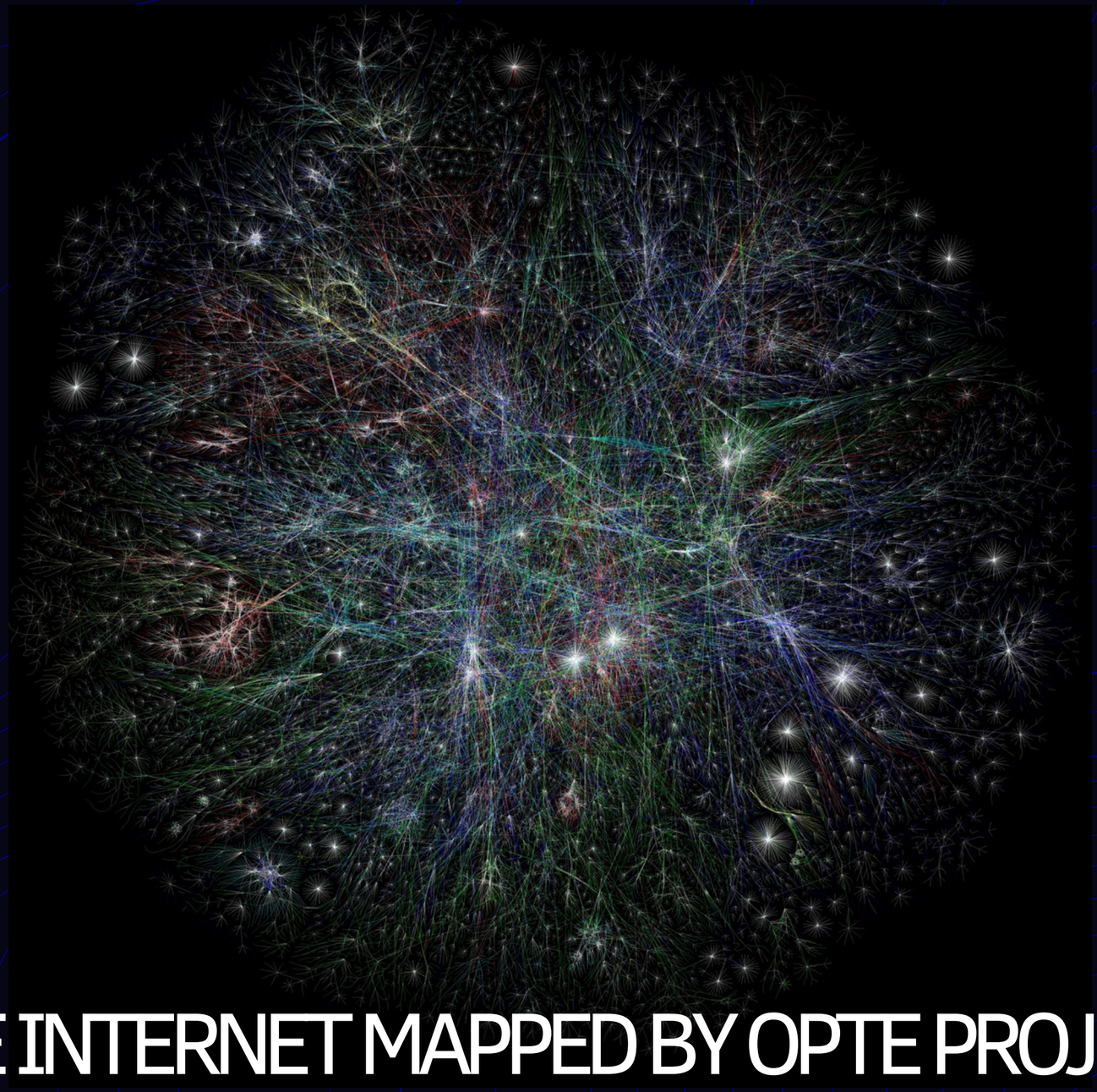


Figure 1. **Largest Connected Subcomponent of the Social Network in the Framingham Heart Study in the Year 2000.**
Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person's body-mass index. The interior color of the circles indicates the person's obesity status: yellow denotes an obese person (body-mass index, ≥30) and green denotes a nonobese person. The colors of the ties between the nodes indicate the relationship between them: purple denotes a friendship or marital tie and orange denotes a familial tie.

FRAMINGHAM HEART STUDY

11



THE INTERNET MAPPED BY OPTE PROJECT

12

These are some of the live applications made possible by studying graphs. Almost all the maps that you and we use is the consequence of studying graphs.

Meaning-out-of-chaos is determined with the help of graph. And if you are going to do anything amazing, you must study the graph algorithms. Mark Zuckerberg is a genius and that’s why he was able to connect everyone around the entire globe. And we are sure he was also a really good student of algorithms.

Graph Terminology

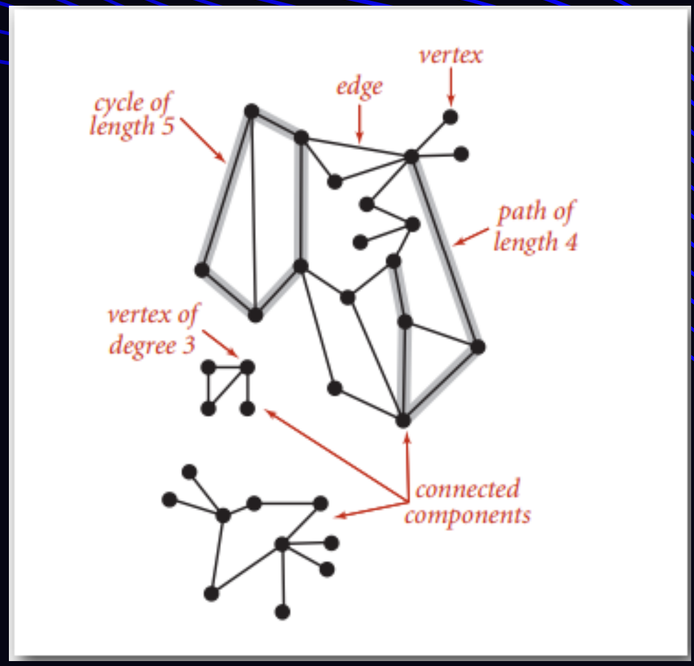
The graph terminology is pretty simple and easy.

Path

Sequence of vertices connected by edge.

Cycle

Path whose first and last vertices are the same.



You can say that the two vertices are connected if there is a path between them.
Say you want to go to point B from some point A.

It is only possible for you to go to that point if there is a path between the two. The path that connects both the point.
Here is the image that depicts the same:

14

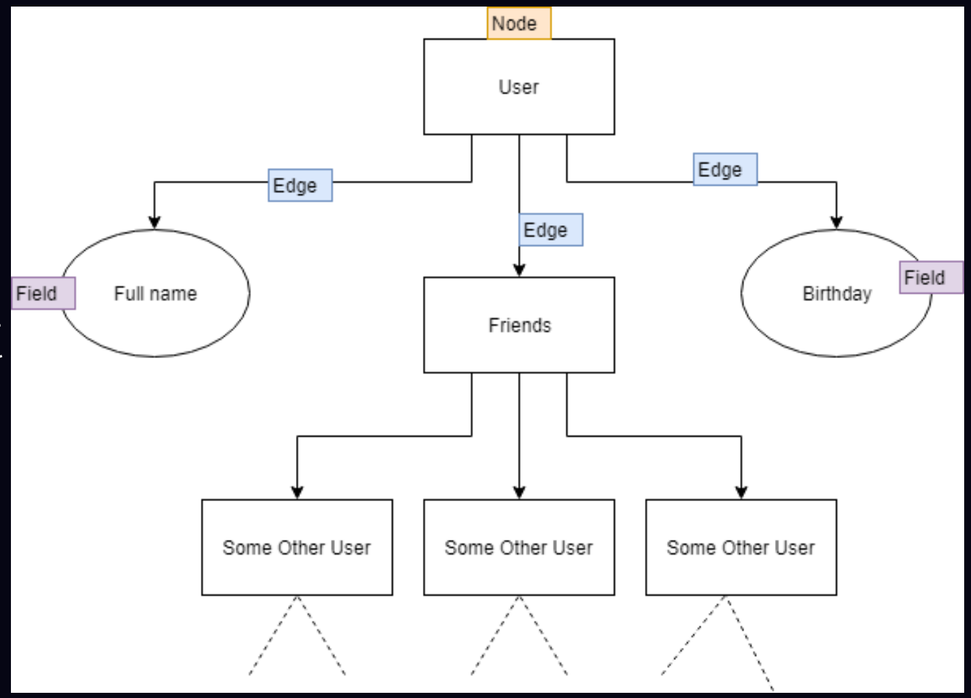
Graph API

We thought of using the Facebook Graph API concept to illustrate it. Because most of you are aware of Facebook and how it works.

And just to be clear, facebook graph API is a real thing. The developers built Facebook this way. Graph API records every action that the user makes on the page.

Facebook’s Graph API is composed of:

- Nodes: Individual objects such as a User, a Photo, a Page or a Comment.
- Edges: they represent connections between different objects such as Photos on a Page or Comments on a Photo.
- Fields: Actual data associated with the object, like User’s birthday or Page’s name.



So in order to access any information, typically you use nodes to get data about a specific object, use edges to get collections of objects on a single object and use fields to get data about a single object or each object in a collection.

16 Implementation Of Graph Using Java.

Graph API

Graph

Graph()// initialize a new Graph

Graph init(File inputFile) //create a graph from input file

List addVertex(Vertex vertex)//add new vertex to the graph

boolean addEdge(Vertex v, Vertex w) //add edge from v to w

int getEdgesCount()// number of edges

List getAdjacentVertices(Vertex vertex) //vertices adjacent to v

Great. Now that we have identified all the required methods, let’s write code to make it work.

Graph

Initialize a new data structure that will hold all the vertices and edges to vertices. Below is the code for the same:

```
@Getter
private final Map<Vertex, List<Vertex>> graph = new HashMap<>();
```

Vertex

Every graph consists of vertices. Create a new class called Vertex

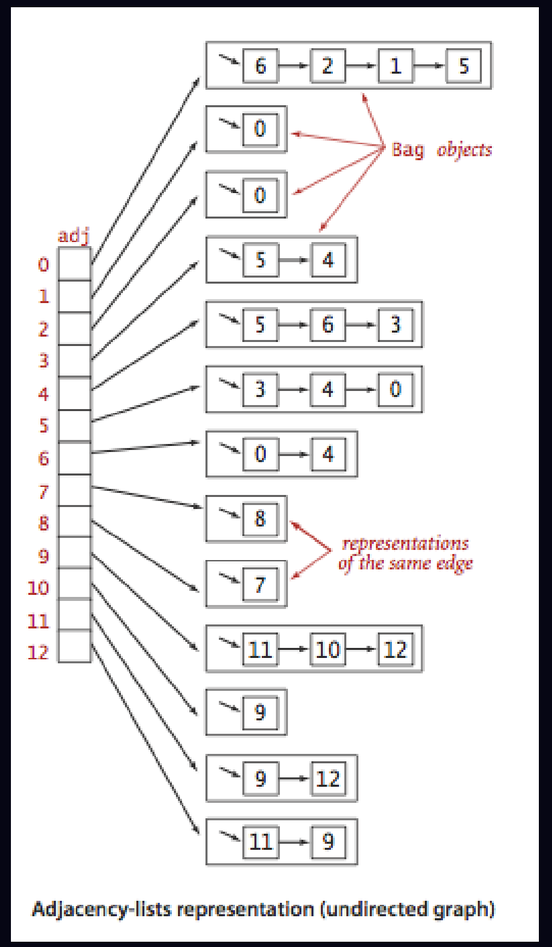
```
@Data

public class Vertex {
    private String label;
    private Vertex(String label) {
    this.label = label;
}
    public static Vertex newLabelInstance(String label) {
    return new Vertex(label);
    }
}
```

The Vertex class will represent the vertex of the graph. It takes one value label. Label is a property that holds the name of the vertex.

Here you are adding a new vertex to the existing graph. The new LinkedList<>()will hold all the vertices that are connected by an edge to this vertex.
In short, every vertex will have its own bag that will contain all the vertices that is connected to this vertex by an edge.
To visualize the above scenario, see the image below:

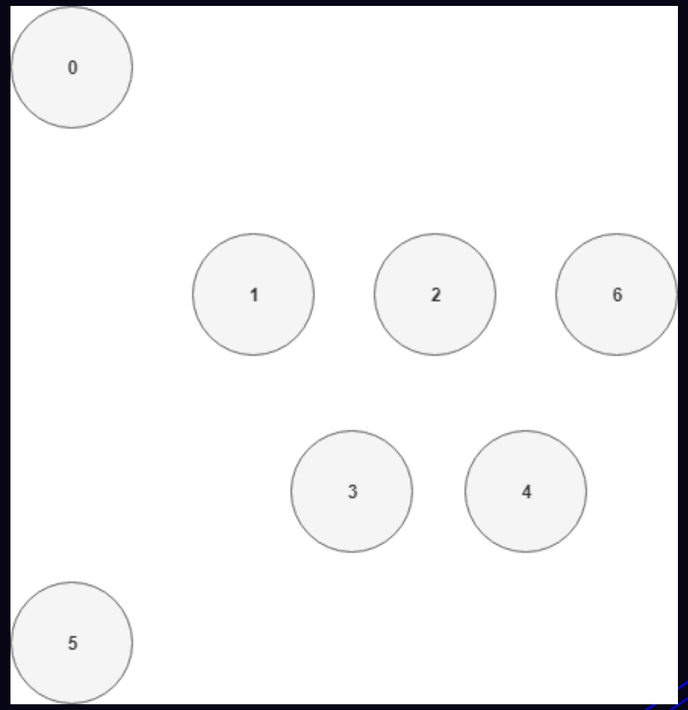
```
//add a new Vertex to the graph
public List<Vertex> addVertex(Vertex vertex) {
    graph.putIfAbsent(vertex, new LinkedList<>());
    return graph.get(vertex);
}
```



Adjacency-lists representation (undirected graph)

This is one way to organize a graph. It is called the Adjacency List. It maintains the Vertex-Indexed array of the list. The information about every vertex of the graph and the edges that connect to it.

After adding the vertices to the graph your Graph will look more like this:



You can add as many vertices you want by calling addVertex(Vertex vertex) method. Main method:

```
public static void main( String[] args ) {
UndirectedGraph undirectedGraph =
UndirectedGraph.newInstance();
undirectedGraph.addVertex(Vertex.newLabelInstance("0"));
undirectedGraph.addVertex(Vertex.newLabelInstance("1"));
undirectedGraph.addVertex(Vertex.newLabelInstance("2"));
undirectedGraph.addVertex(Vertex.newLabelInstance("3"));
undirectedGraph.addVertex(Vertex.newLabelInstance("4"));
undirectedGraph.addVertex(Vertex.newLabelInstance("5"));
undirectedGraph.addVertex(Vertex.newLabelInstance("6"));
}
```

Add Edge

To create an edge that connects two vertices, you need the two vertices. So, let's add the addEdge(Vertex v, Vertex w) method to the graph:

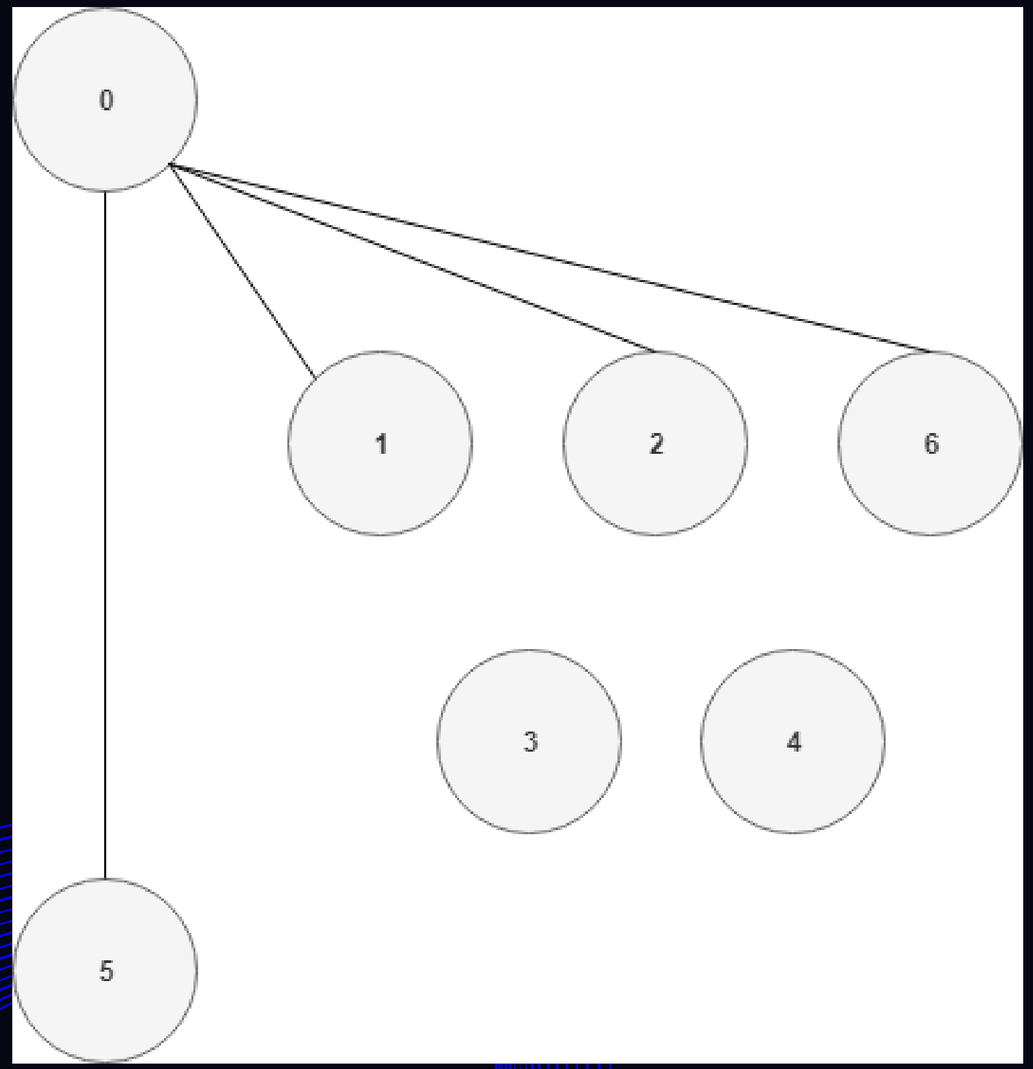
```
public boolean addEdge(Vertex v, Vertex w) {
    if (graph.containsKey(v) && graph.containsKey(w)) {
    edges++;
    return graph.get(v).add(w); } return false;
    }
```

First, you need to first make sure that the graph contains both the vertex before creating an edge between them.
If the graph contains both the vertices than simple increment the edge count and add the edge to vertex V. So, now vertex will have an edge to W (V-W).
After adding the above code to the Graph class, you can use it in the main method to add edges. Main method:

```
public static void main( String[] args ) {
UndirectedGraph undirectedGraph =
UndirectedGraph.newInstance();
undirectedGraph.addVertex(Vertex.newLabelInstance("0"));
undirectedGraph.addVertex(Vertex.newLabelInstance("1"));
undirectedGraph.addVertex(Vertex.newLabelInstance("2"));
undirectedGraph.addVertex(Vertex.newLabelInstance("3"));
undirectedGraph.addVertex(Vertex.newLabelInstance("4"));
undirectedGraph.addVertex(Vertex.newLabelInstance("5"));
undirectedGraph.addVertex(Vertex.newLabelInstance("6"));
undirectedGraph.addEdge(Vertex.newLabelInstance("0"),
Vertex.newLabelInstance("1"));
undirectedGraph.addEdge(Vertex.newLabelInstance("0"),
Vertex.newLabelInstance("2"));
undirectedGraph.addEdge(Vertex.newLabelInstance("0"),
Vertex.newLabelInstance("5"));
undirectedGraph.addEdge(Vertex.newLabelInstance("0"),
Vertex.newLabelInstance("6"));
}
```

21

Now after running above code, your graph contains edges to v1, v2, v5, v6 from v0.
Below is the depiction of the graph after above code:



Adjacent Vertices

Adjacent vertices are the vertices that shares a common edge.

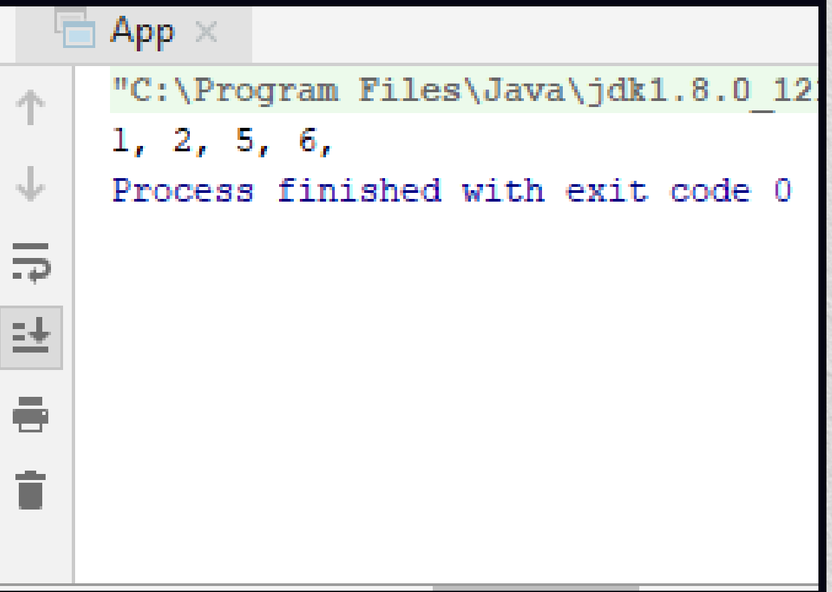
Perhaps, all the adjacent vertices connected to the given vertex. If there is an edge between two vertices they are called adjacent to each other.

Code to print all the adjacent vertices of a given vertex.

```
public List<Vertex>
getAdjacentVertices(Vertex vertex) {
return graph.get(vertex);
}
```

23

The output of the above code will be:




```
public static void main( String[] args ) {
UndirectedGraph undirectedGraph = UndirectedGraph.newInstance();
undirectedGraph.addVertex(Vertex.newLabelInstance("0"));
undirectedGraph.addVertex(Vertex.newLabelInstance("1"));
undirectedGraph.addVertex(Vertex.newLabelInstance("2"));
undirectedGraph.addVertex(Vertex.newLabelInstance("3"));
undirectedGraph.addVertex(Vertex.newLabelInstance("4"));
undirectedGraph.addVertex(Vertex.newLabelInstance("5"));
undirectedGraph.addVertex(Vertex.newLabelInstance("6"));
undirectedGraph.addEdge(Vertex.newLabelInstance("0"),
Vertex.newLabelInstance("1"));
undirectedGraph.addEdge(Vertex.newLabelInstance("0"),
Vertex.newLabelInstance("2"));
undirectedGraph.addEdge(Vertex.newLabelInstance("0"),
Vertex.newLabelInstance("5"));
undirectedGraph.addEdge(Vertex.newLabelInstance("0"),
Vertex.newLabelInstance("6")); for (Vertex v:
undirectedGraph.getAdjacentVertices(Vertex.newLabelInstance("0")))
System.out.print(v.getLabel() + ", ");
}
```

24

Graph processor is going to be an independent class that will perform common operations on all graphs irrespective of its type. So, for this article, you will be adding below functionalities to the graph processor:

- Degree of the graph's vertex
- Maximum degree of the graph
- The average degree of the graph
- Number of self-loops in a graph

To begin with, you need to extract out a Graph Interface. Every graph will implement this interface in order to utilize the common Graph Processor functionalities.

For simplicity, the Graph interface will contain only two methods. Below is the code for the Graph interface.You will see why you have to extract the Graph interface shortly.

So now you will create a Graph processor. You can think of a graph processor as the collection of operation that could be performed on the graph. I have talked about that operation above in bullet points.

```
import java.util.List;
public interface Graph {
    int getEdgesCount();
    Set<Vertex> getVertices();
    List<Vertex> getAdjacentVertices(Vertex vertex
}
```

Graph Processor

Graph processor is going to be an independent class that will perform common operations on all graphs irrespective of its type.

To begin with, you need to extract out a Graph Interface. Every graph will implement this interface in order to utilize the common Graph Processor functionalities.

For simplicity, the Graph interface will contain only two methods. Below is the code for the Graph interface.

```
import java.util.List;
public interface Graph {
    int getEdgesCount();
    Set<Vertex> getVertices();
    List<Vertex> getAdjacentVertices(Vertex
    vertex);
}
```

26

Graph Processor Class (JAVA)

Every method represents a functionality that will be performed on the given graph.

```java
public class GraphProcessor {

    public static int degree(Graph graph, Vertex v){
        return 0;
    }

    public static int maxDegree(Graph graph) {
        return 0;
    }

    public static double averageDegree(Graph graph) {
        return 0.0;
    }

    public static int numberOfSelfLoops(Graph graph)  {
        return 0;
    }
}
```

```java
ipublic class GraphProcessor {
public static int degree(Graph graph, Vertex v){
return graph.getAdjacentVertices(v).size();
 }
public static int maxDegree(Graph graph) {
 int max = 0; for (Vertex vertex : graph.getVertices()) {
if (max < degree(graph, vertex)) max = degree(graph, vertex);
 }
return max;
}
 public static double averageDegree(Graph graph) {
// since for each edge, there are two vertexes associated to it.
// so each edge adds 2 degrees to the graph
// hence multiplied by 2.0 return 2.0 * graph.getEdgesCount() /
graph.getVertices().size();
}
public static int numberOfSelfLoops(Graph graph) {
int count = 0;
for (Vertex vertex : graph.getVertices())
for (Vertex adjacentVertex : graph.getAdjacentVertices(vertex))
if (vertex.equals(adjacentVertex)) count++; return count/2;
 // each edge counted twice
}
 }
```

OUR TEAM:

NURGOZHAYEVA KAMILLA

URMANOVA NAZERKE

MUKHAMBETZHANOVA LEILA

DUISENBAYEV NURLYKHAN

MYRATBEKOV DAMIR

THANKS!