Definition of a Graph

What is a Graph in Computer Science?

A Graph is a type of collection that is non-linear and consists of vertices (sometimes called *nodes*; not to be confused with nodes in a linked list) and edges. An edge is a close relationship or connection between two vertices. In fact, if there is an edge between two vertices, the nodes are said to be adjacent.

Purpose of a Graph

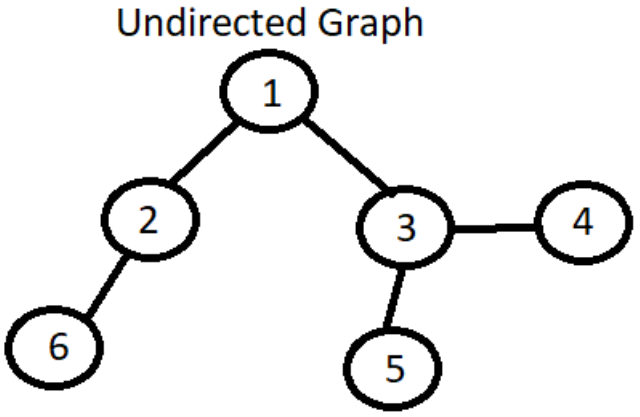
What is the purpose for graphs?

Graphs come from graphing theory in mathematics. We can utilize Graphs to solve a number of real-world problems, however.

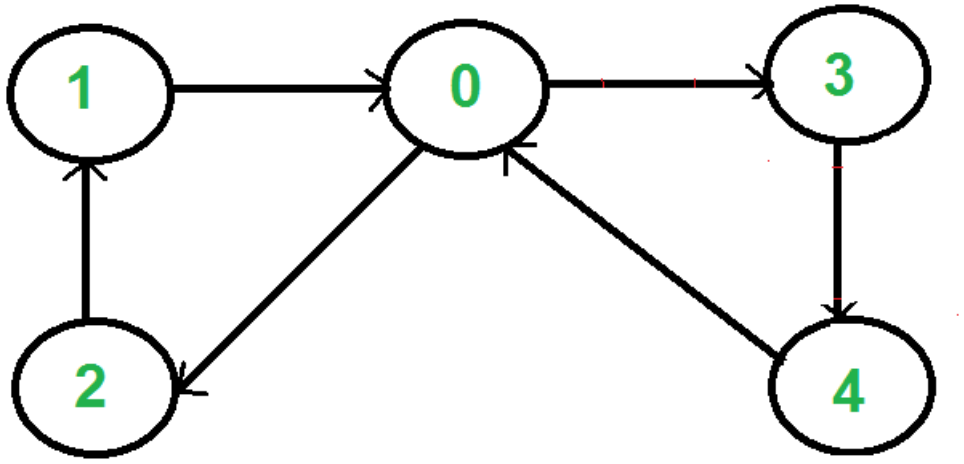
Problems that deal with networks, paths, maps, etc. can utilize the Graph data structure. There are other uses as well, but these are the most common.

Modeling Graphs to better understand them

How do we model a Graph to better understand it?



The above is an example of a graph with the vertices labeled as 1 to 6. The lines between them are the edges that represent a close relationship (known as adjacency or being in close proximity). This particular example is called an “undirected” graph because movement (or traversal) can work in both directions. We can think of an undirected edge as being symmetric since it flows both ways.



This graph, however, is an example of a “directed” graph because the edges point to an asymmetric relationship. We can traverse the nodes in only one direction but not back the way we came.

This particular graph also contains what we call a “cycle.” A cycle is when we have three or more vertices connected in a closed chain. This example has two cycles: 0-2-1 and 0-3-4.

When a graph contains NO cycles, we refer to it as “acyclic” (pronounced "a-sick-lick").

- Our first example is an Undirected Acyclic Graph.
- The second example is a Directed Cyclic Graph.

Another type of graph is a Directed graph with no cycles. This particular type of graph is often called a “DAG” which stands for “Directed Acyclic Graph.” These are useful for topological sorting (the ordering of vertices in a graph). The text provides examples of Topological sorting if you are interested in learning more about it.

For now, we will stick with the Undirected Graph example to understand the Graph API we are going to write.

Data Structure to represent the Graph?

What data structure should we use to store a graph behind-the-scene?

There are two ways we generally store a graph in Computer Science. These are as an Adjacency List or an Adjacency Matrix.

An Adjacency List for the Undirected Graph Example would be:

0 →

1 → 2, 3

2 → 1, 6

3 → 1, 4, 5

4 → 3

5 → 3

6 → 2

An Adjacency Matrix would look like:

(The columns are also numbered from 0 to 6)

// Edges between vertices are denoted by a 1; 0 means no edge between vertices

Row 0 -> 0 0 0 0 0 0 0

Row 1 -> 0 0 1 1 0 0 0

Row 2 -> 0 1 0 0 0 0 1

Row 3 -> 0 1 0 0 1 1 0

Row 4 -> 0 0 0 1 0 0 0

Row 5 -> 0 0 0 1 0 0 0

Row 6 -> 0 0 1 0 0 0 0

Our custom example is going to use an Adjacency Matrix to represent the Graph in memory. My reasoning is that:

A matrix can benefit from the constant time (O(1)) access since we use an array

However, all things in Computer Science are a trade-off. What do we give?

We give up v^2 space (v = number of vertices in Graph) to get constant time access

What are the time and space complexities of an Adjacency List?

Time: O(v) to see if there is an edge between two vertices

Space: v + e space to store vertices and edges

The decision of whether to use a List or Matrix is not generally made in a vacuum. The prevailing wisdom is that:

- Dense Graphs favor a Matrix
- Sparse Graphs favor a List

The terms "dense" and "sparse" are not particularly precise. The truth of the matter is that, unless there is an extreme amount of density or an extreme level of sparseness, the difference between the two is unlikely to be significant in practical settings.

Therefore, it isn't worth the time to worry too much about density or sparseness. It should be obvious if the situation presents as one extreme or the other.

What functionality should our Graph class contain?

Our Graph class should contain the following methods:

```
Graph(int v); // Constructor to create a graph with “v” vertices

int V(); // Return how many vertices in the graph

int E(); // Return how many edges are currently in graph

void addEdge(int v, int w); // Add an edge in graph between these two nodes

int[] adj(int v); // Return array of adjacent verts to “v”

boolean isAdjacent(int v1, int v2); // Return whether they are adjacent in graph

String BFS(int v); // Return string with BFS path starting from “v”

String DFS(int v); // Return string with DFS path starting from “v”

boolean isConnected(int v1, int v2); // Return whether two nodes are somehow connected
```

BFS stands for "Breadth-first Search" and DFS stands for "Depth-First Search." These concepts will be explained on the next page.

Create a Graph class from scratch

How do we write the code from scratch?

```
package Main;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.Queue;

public class Graph{

	// Fields

	private int[][] g; // Use Adjacency Matrix

	private int v; // Number of verts

	private int e; // Number of edges

	// Constructor

	public Graph(int v){

		g = new int[v][v];

		this.v = v;

		e = 0;

	}

	// Methods

	public int V(){

		return v;

	}

	public int E(){

		return e;

	}

	public void addEdge(int v1, int v2){

		g[v1][v2] = 1;

		g[v2][v1] = 1;

		e++;

	}

	public boolean isAdjacent(int v1, int v2){

		return g[v1][v2] == 1;

	}
```

```
public boolean isConnected(int v1, int v2){
	int[] bfs = BFStoArray(v1);
	for(int i = 0; i < bfs.length; i++){
		if(bfs[i] == v2)
			return true;
	}
	return false;
}
public int[] adj(int v){
	ArrayList<Integer> al = new ArrayList<Integer>();
	for(int i = 0; i < this.v; i++){
		if(g[v][i] == 1)
			al.add(i);
	}
	int[] ret = new int[al.size()];
	for(int i = 0; i < ret.length;i++)
		ret[i] = al.get(i);
	return ret;
}
public String adjString(int v){
	int[] a = adj(v);
	String ret = "Verts adjacent to " + v + ": ";
	for(int i = 0; i < a.length; i++)
		ret += a[i] + " ";
	return ret;
}
/* Helper function that can be used later by isConnected as well */
private int[] BFStoArray(int v){
	ArrayList<Integer> al = new ArrayList<Integer>();
	boolean[] visited = new boolean[this.v];
	Queue<Integer> q = new LinkedList<Integer>();
	q.add(v);
	visited[v] = true;
	int visit;
```

```
        while(!q.isEmpty()){
            visit = q.remove();
            al.add(visit);
            for(int i = 0; i < this.v; i++){
                if(isAdjacent(visit, i) && !visited[i]){
                    q.add(i);
                    visited[i] = true;
                }
            }
        }
        int[] ret = new int[al.size()];
        for(int i = 0; i < ret.length; i++)
            ret[i] = al.get(i);
        return ret;
    }
    public String BFS(int v){
        String ret = "BFS for Vert #" + v + ": ";
        int[] bfs = BFStoArray(v);
        for(int i = 0; i < bfs.length; i++)
            ret += bfs[i] + " ";
        return ret;
    }
    /* Initiator (wrapper) method for recursive call */
    public String DFS(int v){
        boolean[] visited = new boolean[this.v];
        String ret = DFS(v, visited, "DFS for Vert #" + v + ": ");
        return ret;
    }
    /* Workhorse (helper) method for recursive call */
        private String DFS(int v, boolean[] visited, String str){
        str += v + " ";
        visited[v] = true;
        for(int i = 0; i < this.v; i++){
            if(isAdjacent(v, i) && !visited[i])
```

```
                    str = DFS(i, visited, str);
                }
                return str;
            }
        }
```

Using our Graph class in practice

How would we use this code in the main method?

Below is an example of how we can use our Graph class code:

```
package Main;

public class GraphTest{

	public static void main(String[] args){

		Graph g = new Graph(7); // Create new graph with 7 vertices

		g.addEdge(1, 2);

		g.addEdge(1, 3);

		g.addEdge(2, 6);

		g.addEdge(3, 4);

		g.addEdge(3, 5);

		p("Number of Vertices: " + g.V());

		p("Number of Edges: " + g.E());

		for(int i = 0; i < g.V(); i++)

			p(g.adjString(i));

		p("1 and 5 are adjacent: " + g.isAdjacent(1, 5));

		p("1 and 3 are adjacent: " + g.isAdjacent(1, 3));

		p(g.BFS(1));

		p(g.DFS(1));

		p("Verts 1 and 5 are connected: " + g.isConnected(1, 5));

		p("Verts 0 and 4 are connected: " + g.isConnected(0, 4));

	}

	public static <E> void p(E item){

		System.out.println(item);

	}

}
```

Other options for Graphs?

There is also the ability to weight the edges that connect the vertices. How we could accomplish this is by allowing a value other than 1 for edges in an adjacency matrix. Doing this allows us to have a way to define ideas such as distance between vertices in an edge. This can open up interesting possibilities when using shortest path algorithms to determine the most efficient traversal between two nodes.

If you are interested in Shortest Path Algorithms, the book contains examples of common implementations including: Dijkstra's, Prim's MST, Topological Sorting for DAGs, Bellman-Ford, and more.

There are many more ideas than this with graphs, but this gives you a foundation to build upon if you are interested in exploring the concept further.