# COM2031 Advanced Algorithms, Autumn Semester 2019

## Graph Data Structures and Operations Revision

Graphs are read from various file formats, or databases to be represented in memory using various Graph data structures. The first section of this document discusses various data structures, then some common applications to graph modelling. Two Java classes accompanies this document for educational purposes. `Graph` class is a template class that represent a graph of (V) generic type vertices, in which V is a generic type that can be any class type declared. You will see examples of this V being a `String` class and being a `City` user defined class in the `DisplayGraphs` class. The `DisplayGraphs` class contain the `main` method that reads various graphs examples using different file formats and different data structures. The `graphTests` method in the `DisplayGraphs` class perform some graph operations on the passed graph object as defined from the different graph example methods and print to the console the output. There is also a `MinHeap` class that implements Heap data structures used in the Dijkstra algorithm implementation in the `Graph` class using adjacency lists in the `dijkstra_al` method.

A current limitation on the `DisplayGraphs` class is that the generic type V will need to have X and Y coordinates to generate the frame visualisation. Further abstractions can be provided in the future that does not limit that the data type such as what is discussed in the graph visualisation section.

**Graph Visualisation:**
Some graphs of the examples presented can be displayed in a frame because they are defined using the `City` user defined class that has X and Y coordinate such as in the US cities example. The UK Cities example cities were assigned X and Y coordinate incrementally using the nextPoint method presented in the first lab of this module. Further aesthetic choices of the grid points to assign to vertices can be applied using graph visualisation packages such as graphViz Java wrapper:

https://github.com/nidi3/graphviz-java

GraphViz is an open source library that contain various graph drawing tools such as the following:

- The *dot* tool draws various directed graphs in "hierarchical" or layered drawings.
- *neato* draws undirected graphs using a ‘‘spring’’ model
- *twopi* draws graphs using a radial layout
- *circo* draws graphs using a circular layout
- *fdp* and *sfdp* draws undirected graphs using a ‘‘spring’’ model, but *sfdp* uses a multi-scale approach to produce layouts of large graphs in a reasonably short time.
- *patchwork* draws the graph as a squarified treemap.
- *osage* draws the graph using its cluster structure.

https://www.graphviz.org

## Large Graphs

The test examples are generally small graphs except the movie’s performers contain 3466 verteces of 100 movies and 3366 performer forming a bipartite graph. You will also find csv files for 1260 movies, and 4188 movies for your own testing. It might be interesting to take this further in your COM2039 module for parallel and distributed experimentation, or your graduation projects.

## Graph Data Structures

Given a graph G = (V, E), as illustrated in the figure, we will explore the data structures to represent with pros and cons.

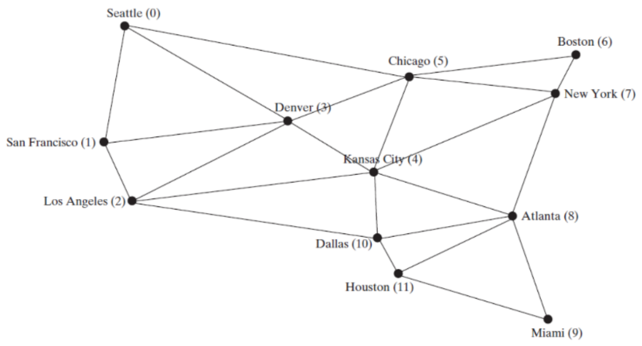


*Figure 1: A graph can be used to model the flights between the cities*

The data structure to represent graphs are either linked lists, arrays or matrix structures. In concrete applications the best structure is often a combination of these all. List structures are often preferred for sparse graphs as they have smaller memory requirements. Matrix structures on the other hand provide faster access for some applications but can consume huge amounts of memory.

List/Arrays structures include the **incidence list**, which is an array of pairs of vertices, and the **adjacency list**, which separately lists the neighbors of each vertex: Much like the incidence list, each vertex has a list of which vertices it is adjacent to.

Matrix structures include the **incidence matrix**, a matrix of 0's and 1's whose rows represent vertices and whose columns represent edges, and the **adjacency matrix**, in which both the rows and columns are indexed by vertices. In both cases a 1 indicates two adjacent objects and a 0 indicates two non-adjacent objects. For weighted graphs, 1 for the existence of an edge can be replaced by the weight of an edge. For the directed graph adjacency matrix, a convention need to be defined as whether to consider the row as the source node and the column as the destination node or otherwise. For the directed graph incidence matrix, the convention can be that $B_{i,j} = -1$ if the edge $e_j$ leaves vertex $v_i$, 1 if it enters vertex $v_i$ and 0 otherwise (many authors use the opposite sign convention).

Other representations include the **Laplacian matrix** as a modified form of the adjacency matrix that incorporates information about the degrees of the vertices. Also, the **distance matrix**, like the adjacency matrix, has both its rows and columns indexed by vertices, but rather than containing specific edge information in each cell, it contains the length of a shortest path between two vertices.

Below are some example implementations for the graph in Figure 1.

### 1.1 Vertices as List of Objects, each containing its neighbours Lists

From COM1029 you studied **adjacency list** by creating a data structures of vertices that contain edges list embedded in it. In your node/vertex data structure, you can add any attribute of interest. If it is a city, you can add population, geographic coordinates, country … etc. If it is a person, you can add date of birth, city of birth … etc. Edges as well can be attributed with more than just the direction and the cost. It the edge represents a road, it can be attributed with date road established, services along the road … etc.

```
public class Node {
     public String name ; // Node name
     public List <Edge > adj; // Adjacent vertices
```

```
	public Node ( String nm){
		name = nm;
		adj = new LinkedList <Edge >();
	}
}

public class Edge implements Comparable <Edge > {
	public Node source ; // First vertex in Edge - if directed,
	otherwise, just first and second node
	public Node dest ; // Second vertex in Edge
	public double cost ; // Edge cost, if weighted, otherwise you
	can ignore this field
	public Edge ( Node s, Node d, double c){
		source = s;
		dest = d;
		cost = c;
	}
}
```

To represent the graph in Figure 1 provided I overload the constructor to take name only for now:

```
Node city0 = new Node ("Seattle");
Node city1 = new Node ("San Francisco");
Edge e1 = new Edge (city0, city1, 0); // since this is not a
weighted nor directed graph, the order of the cities in the
parameters does not matter, and zero for the cost for now.

Node city3 = new Node ("Denver ");

Node city5 = new Node ("Chicago");

Edge e2 = new Edge (city0, city3, 0);
Edge e3 = new Edge (city0, city5, 0);
Edge[] city0Neighbours = {e1, e2, e3};

city0.adj = city0Neighbours;


...
Node city11 = new Node ("Houston", 2099451, "Annise Parker");
Node [] vertices = {city0, city1, ... , city11};
```

Notice that the same edge will be represented twice, as a neighbour to source, and as a neighbour to destination, and how long it takes to define a graph.

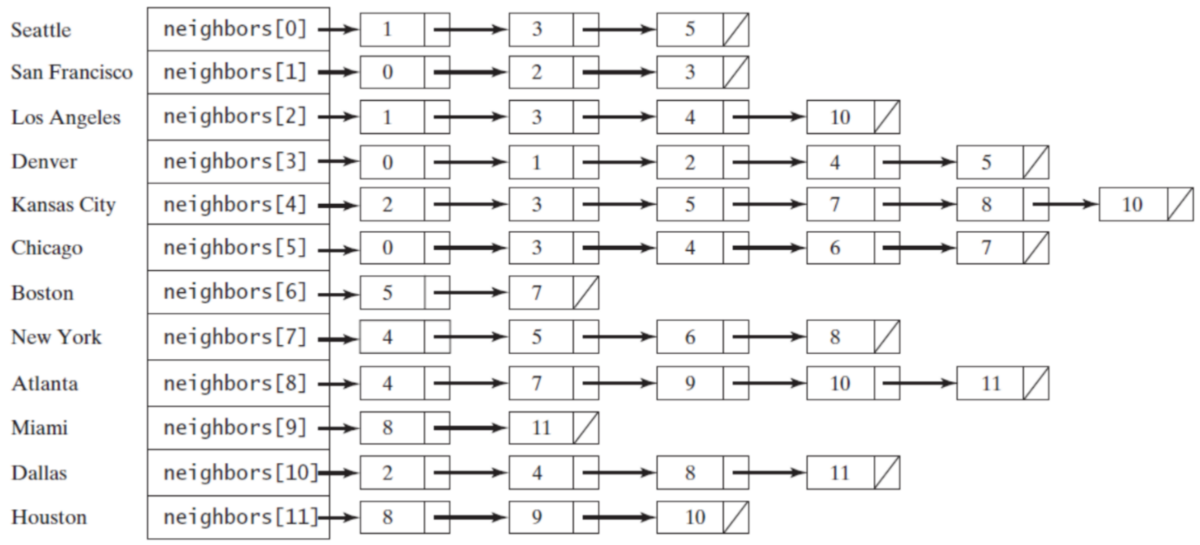
### 1.2 Vertices & Edges Lists of Objects:

**Adjacency independent Lists** is similar to list of objects, but instead of having the Edge list as neighbours encapsulated in the node/vertex data structure, Edge List can be an independent list of objects, such that a given edge is represented once. For example to represent the graph in Figure 1, instead of using Node, we will use String array for vertices and linked list for all neighbours:

```
String [] V = {"Seattle", "San Francisco", "Los Angeles",
"Denver", "Kansas City", "Chicago", "Boston", "New York",
"Atlanta", "Miami", "Dallas", "Houston"};
```

```
java.util.LinkedList[] neighbors = new java.util.LinkedList[12];
```

neighbors[0] contains all vertices pointed from vertex 0 via directed edges, neighbors[1] contains all vertices pointed from vertex 1 via directed edges, and so on. The graph in Figure 1 is represented as shown in as shown in Figure 2. Wendy does not point to any vertex, so neighbors[4] is null.



Seattle neighbors[0] 1 3 5
San Francisco neighbors[1] 0 2 3
Los Angeles neighbors[2] 1 3 4 10
Denver neighbors[3] 0 1 2 4 5
Kansas City neighbors[4] 2 3 5 7 8 10
Chicago neighbors[5] 0 3 4 6 7
Boston neighbors[6] 5 7
New York neighbors[7] 4 5 6 8
Atlanta neighbors[8] 4 7 9 10 11
Miami neighbors[9] 8 11
Dallas neighbors[10] 2 4 8 11
Houston neighbors[11] 8 9 10

*Figure 2: Edges in the graph in Figure 1 are represented using linked lists*

You cannot make edge lists in such a representation weighted or attributed, unless it becomes a linked list of objects.

### 1.3 Vertices & Edge Incidence Lists/Arrays:

Using a class to create node objects as in the first data structure, we can also represent edges as **incidence list** of Edge 2D arrays rather than linked list, as first dimension is the number of edges, and the second dimension is 2 to identify the 2 vertices incident on the edge. For example define a City class and create array of objects rather than list of objects, and edges are 2D array as in the example below. This can be directed or undirected based on how you process it. However, it cannot be weighted or further attributed, unless it is a 2D array of objects and not of integer values.

```
City city0 = new City("Seattle", 608660, "Mike McGinn");
...
City city11 = new City("Houston", 2099451, "Annise Parker");
City[] vertices = {city0, city1, ... , city11};

int[][] edges = {
{0, 1}, {0, 3}, {0, 5},
{1, 0}, {1, 2}, {1, 3},
{2, 1}, {2, 3}, {2, 4}, {2, 10},
{3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
{4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
{5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
{6, 5}, {6, 7},
{7, 4}, {7, 5}, {7, 6}, {7, 8},
{8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
{9, 8}, {9, 11},
{10, 2}, {10, 4}, {10, 8}, {10, 11},
{11, 8}, {11, 9}, {11, 10}
};
```

### 1.4 Adjacency Matrix:

Assume that the graph has *n* vertices defined as above, you can use a two-dimensional matrix, say adjacencyMatrix, to represent the edges. Each element in the array is 0 or 1. adjacencyMatrix[i][j] is 1 if there is an edge from vertex *i* to vertex *j*; otherwise, adjacencyMatrix[i][j] is 0. Otherwise, it can have a value to represent a weighted graph. If the graph is undirected, the matrix is symmetric, because adjacencyMatrix[i][j] is the same as adjacencyMatrix[j][i]. For example, the edges in the graph in Figure 1 can be represented using an *adjacency matrix* as follows:

```
int[][] adjacencyMatrix = {
{ , 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0}, // Seattle
{1, , 1, 1, 0, 0, 0, 0, 0, 0, 0, 0}, // San Francisco
{0, 1, , 1, 1, 1, 0, 0, 0, 0, 0, 0}, // Los Angeles
{1, 1, 1, , 1, 1, 0, 0, 0, 0, 0, 0}, // Denver
{0, 0, 1, 1, , 1, 0, 1, 1, 0, 1, 0}, // Kansas City
{1, 0, 0, 1, 1, , 1, 1, 0, 0, 0, 0}, // Chicago
{0, 0, 0, 0, 0, 1, , 1, 0, 0, 0, 0}, // Boston
{0, 0, 0, 0, 1, 1, 1, , 1, 0, 0, 0}, // New York
{0, 0, 0, 1, 1, 0, 0, 1, , 1, 1, 1}, // Atlanta
{0, 0, 0, 0, 0, 0, 0, 0, 1, , 0, 1}, // Miami
{0, 0, 1, 0, 1, 0, 0, 0, 1, 0, , 1}, // Dallas
{0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, } // Houston
};
```

## Basic Graph Operations:

**Graph Declaration:** Various constructors are implemented to read from one data structure and generate the others. You can implement and remove what is not useful to your problem at hand. Other methods to define the graph incrementally are Add/Remove/Update Vertex, and Add/Remove/Update Edge. Only the `addVertex` and the `addEdge` methods are implemented currently. `clear` method is also implemented to clear the memory allocated for all defined data structures.

**Graph Properties:**

- **Degree:** `getDegree` is a method to return the degree (number of edges incident on it, and if directed, there is an outdegree and indegree that can be defined seperately).
- **Cycles:** Detecting the cycles is implemented in `isCyclic` method. Finding the Cycles in implemented in the recursive methods: `getACycle` that searches for a cycle from a given vertex. The method returns a List that contains all the vertices in a cycle starting from u. If the graph doesn't have any cycles, the method returns null.
- **Connected Components:** `getConnectedComponents` is a method that returns a List of Lists of vertices that are connected in different components.

**Graph Traversal:**

**BFS** is breadth first search starting from one vertex in the graph as a root of the generated tree, then all its adjacent vertices, then all the vertices adjacent to those vertices, and so on until all vertices are visited.

**DFS** is depth first search starting from one vertex in the graph as a root, then recursively visits the subtrees of the root, keeping track of visited vertices to avoid infinite cycles.

*Applications of BFS include:*

- Testing whether a graph is bipartite. (A graph is bipartite if the vertices of the graph can be divided into two disjoint sets such that no edges exist between vertices in the same set.)

*Applications of the DFS include:*

- Finding a Hamiltonian path/cycle. A Hamiltonian path in a graph is a path that visits each vertex in the graph exactly once. A Hamiltonian cycle visits each vertex in the graph exactly once and returns to the starting vertex.

*Applications of BFS or DFS include:*

- Detecting whether a graph is connected. A graph is connected if there is a path between any two vertices in the graph. Search the graph starting from any vertex. If the number of vertices searched is the same as the number of vertices in the graph, the graph is connected. Otherwise, the graph is not connected
- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.
- Detecting whether there is a path between two vertices.
- Finding the shortest path between two vertices. You can prove that the path between the root and any node in the BFS tree is the shortest path between the root and the node.
- Detecting whether there is a cycle in the graph
- Finding a cycle in the graph

You will find examples of BFS and DFS methods in the `Graph` class.

**Minimum spanning tree**: In COM1029, you were provided with the Prim's and Kruskal's algorithms for constructing a minimal spanning tree for a given graph. You will find example of Prim's using adjacency lists in the `prims_al` method and another using adjacency matrix in the `prims_am` method in the `Graph` class. You will also find an example of Kruskal's algorithm using adjacency matrix in the `kruskal_am` method, and another greedy algorithm using adjacency lists in the `Kruskal_greedy` method. Testing showed that the Prims implementation using adjacency matrix does not work with bipartite graphs because the limitation of edges' count in the minimum spanning tree to be the vertices count -1 does not hold. Also, the Kruskal algorithm using adjacency matrix is not efficient for big graphs.

**Path Finding:** There are various approaches to find path between 2 nodes without optimising the search to shortest path. A `getPath` is implemented in the `Graph` template using BFS traversal, and `getPath_dfs` using DFS traversal. The method returns a List<Integer> that contains all the vertices in a path from u to v in this order. Using the BFS approach, you can obtain the shortest path from u to v. If there isn't a path from u to v, the method returns null.

**Shortest Path**: In COM1029 you studied Dijkstra's algorithm for shortest paths between all nodes and a source node. You will find two implementations in the template class using adjacency matrices in the `dijkstra_am` method and another using adjacency lists in the `dijkstra_al` method.

**Network flow**: Max flow min cut theorem template is provided in a simplified template `Graph_mincut_maxflow` class using adjacency matrix. Fill in the required TODO method as explained in the lab sheet. Sample solution will be provided next week.

**Other problems include:**

- **Graph Enumeration:** describes a class of combinatorial enumeration problems in which one must count undirected or directed graphs of certain types, typically as a function of the number of vertices of the graph.
- **Subgraph Isomorphism:** finding a fixed graph as a subgraph in a given graph, for example finding the largest complete subgraph is called the clique problem (NP-complete).
- Finding **induced subgraphs** in a given graph, for example: Finding the largest edgeless induced subgraph or independent set is called the independent set problem (NP-complete).
- **Graph colouring:** colouring a graph so that no two adjacent vertices have the same colour, or with other similar restrictions. Or colouring edges such that no two coincident edges are the same colour), or other variations.
- **Visibility problems:** Museum guard problem
- **Decomposition problems:** decompose a graph into subgraphs isomorphic to a fixed graph; for instance, decomposing a complete graph into Hamiltonian cycles.