Graphs

Graphs

A graph is a very powerful data structure and allows for the solving of many unintuitive questions. Unfortunately, its implementation can also be equally unintuitive.

Terminologies

A graph shares many common terminologies with BSTs, such as:

- Vertex – a point in the graph
  - In Degree: Number of edges going in to this vertex
  - Out Degree: Number of edges going out of this vertex
- Edge – connection between two vertices

It also has the following:

- Weight – can be seen as the cost of taking a certain edge
- Direction – from which vertex to which vertex is the edge going?
  - Undirected – can go both ways with the same weight
    - Adjacent nodes
    - Neighbors
  - Directed – can only go one way
    - Neighbors only in a certain direction
  - Bidirected – can go both ways but with different weights

However, there is no notion of parent, child, root or ancestor/descendant in a graph.

A **simple** graph is a set of vertices where some [0…NC2] pairs of the vertices are connected by edges.

A **sparse** graph and a **dense** graph are just ways to describe the number of edges compared to the number of vertices. There is no specification on how many edges would cause a sparse graph to become dense.

A **complete** graph has every vertex connected to all other vertices in the graph. In other words, it is a simple graph with N vertices and ${}^{N}C_{2}$ edges.

A **simple** path is a sequence vertices connected by a sequence of edges with no repeated vertex.

A **simple directed** path is a simple path connected by directed edges which are all in one direction.

The **path length/cost** of a path is the cost of traversing a path. In an unweighted graph, this is the number of edges in the path. In a weighted path, this would be the sum of edge weights in the path.

A **simple cycle** is a path that starts and ends with the same vertex. There are no other repeated vertices except for the start/end vertex.

A **simple directed cycle** is a simple cycle with directed edges all in one direction.

A **component** is a group of vertices in an **undirected** graph that can visit each other via some path.

A **connected graph** is an undirected graph with one component.

A **reachable vertex** is a vertex that can be reached from a certain starting vertex. An **unreachable vertex** is one that cannot be reached.

A **subgraph** is a subset of vertices and their connecting edges of the original graph. For example, we can express it as {7-6-8 5} (2 components) or {3-4 6-8 5} (3 components).

A **directed acyclic graph (DAG)** is a directed graph with no cycle.

A **tree** is a connected and undirected graph with V vertices and E = V-1 edges, with one unique path between any pair of vertices.

A **bipartite graph** is an undirected graph with vertices that can be partitioned into two sets such that there are no edges between members of the same set.

Graph Implementation #1: Adjacency Matrix
The concept of an adjacency matrix is to keep a record of all edges using a 2D array.

Adjacency Matrix Implementation: 2D Array
With a V x V sized array, where V is the number of vertices, we can find out if there's an edge from vertex i to vertex j if AdjMatrix[i][j] = 1, else if it equals 0 then there's no edge.

- Can account for directed graphs
  - AdjMatrix[i][j] = 1 => i->j exists
  - AdjMatrix[j][i] = 0 => j->i does not exist
- Can account for weighted graphs
  - AdjMatrix[i][j] = weight of edge from i to j
- Space Complexity: $O(V^2)$, where V = number of vertices

Possible way of implementing:

```
int i, V = NUM_V; // NUM_V has been set before
int[][] AdjMatrix = new int[V][V];
```

Graph Implementation #2: Adjacency List
Since an Adjacency Matrix takes up too much space, another way is to store V lists in an array, with a list of neighbors for each vertex. For a weighted graph, we can store a pair of numbers, one for the neighbor and one for the weight.

Adjacency List Implementation #1: Array of Linked Lists
One way is to use linked lists to store the neighbours, since every vertex will have a different number of neighbours. A linked list may help to save on unused memory, but it will suffer from the overhead of storing references and cache misses.

Adjacency List Implementation #2: Array of Arrays
Another way is to have an array storing arrays. This can help with cache locality, but may face issues with unused memory and the need for resizing.

The overall space complexity for both implementation is O(V+E).

- E is the number of edges in the graph
- Worse case for E is $O(V^2)$
- O(V+E) therefore approximates to O(max(V, E))

Possible way of implementing:

```
ArrayList <ArrayList<IntegerPair>> AdjList =
      new ArrayList<ArrayList<IntegerPair>>();
```

Graph Implementation #3: Edge List
The idea is to avoid the need to store the edges with respect to the vertices, but simply keep the information on the vertices with the edge itself.

Edge List Implementation: Array
Simply store an array of IntegerTriples

- Outgoing Vertex, Incoming Vertex, Edge Weight
- If the graph is unweighted, an IntegerPair will work as well
- Overall space complexity: O(E) = O(V^2) (worst case)

Possible way of implementing:

```
ArrayList<IntegerTriple> EdgeList = new ArrayList<IntegerTriple>();
```

	Adjacency Matrix	**Adjacency List**	**Edge List**
	2D Array	Array of Arrays	Array of Edges
Count Vertices	O(1) – Number of rows	O(1) – Number of rows	May be impossible, else O(E)
Enumerating Neighbours	O(V) – Iterate through all V items in row v	O(k) – Iterate through all k items in row v	O(E log V + k) – Sort edges and binary search before scanning k neighbours
Count Edges	$O(V^2)$ – Count all non-zero entries	O(V) – Sum length of V lists	O(1) – Return size of array
Presence of Edge	O(1) – Check AdjMatrix[u][v]	O(k) – Scan through k neighbours of u to find v	O(E) – Search all E edges. If sorted, potentially O(log E + k).
BFS	$O(V^2)$ – Outer loop V times, inner loop V times to check for neighbours May be preferred for dense graphs due to better cache performance	O(V + E) – V vertices get queued once, and with k edges examined each time where Vk = E	Impossible to initialise
DFS	$O(V^2)$ – Outer loop V times, inner loop V times to check for neighbours May be preferred for dense graphs due to better cache performance	O(V + E) – V vertices get queued once, and with k edges examined each time where Vk = E	Impossible to initialise
Prim's	$O(V^2)$ – Eventually, all vertices are added, and all edges are enqueued once. This requires V enumerations, resulting in $O(V^2)$ performance May be preferred for dense graphs due to better cache performance, as O(E) tends to $O(V^2)$	O(E) – All edges are added once, with all V vertices having their neighbours enumerated	O(VE) – If linear search is employed. With V vertices, V searches are needed over E edges. O(E log V + E) – E log V for sorting and binary search, and E for eventual V retrievals of k edges each, summing up to E.
Kruskal's	Difficult due to inability to sort by edge weights easily	Difficult due to inability to sort by edge weights easily	Very good – enables O(E log V) sorting
Pros	O(1) edge existence checking Good for dense graphs	O(k) enumeration Good for sparse graphs, BFS, DFS, Dijkstra's O(V+E) space	O(E log V) sorting Good for Kruskal's O(1) edge counting O(E) space
Cons	$O(V^2)$ space O(V) enumeration	O(k) edge existence checking	O(E) edge existence checking

		Overhead for array maintenance for sparse graph	O(E) enumeration