**Graph Terminology**

V = all the vertices in the graph.

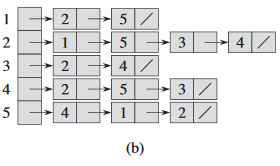
E = All connections between the vertices in the graph. See the list example for more info.

**What does it mean to search a graph?**

It means you trace the edges of it to find the vertices (Points where multiple lines/curves meet). Methods for doing this consist of using input graphs to analyse the nature of the output graphs, or just through other methods.

**How do we represent these graphs?**

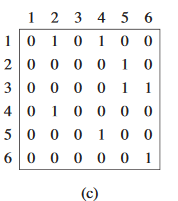
Assuming they follow the format (Don’t ask what that means), they can either be represented as adjacency lists or adjacency matrices. Lists are generally used as they are compact (I guess in memory?), and function well with sparse graphs. If the graph is dense however, matrices are recommended. The difference between sparse and dense graphs is based on E’s relation to V2. The closer E is to V2, the denser the graph is.

**More on the lists please**

In list representations, you have an array *Adj* of lists which size is determined by |V|. Now assume we have some variable u, then the array Adj[u] contains all the vertices v that fulfil the condition . To make it easier to understand, look at the image to the right. There, a bunch of arrays are listed in an orderly fashion. The first number in each array correspond to u, and all the following numbers are the ones that u can go to / become / are adjacent to.

In the book on figure 22.1(a) and 22.2(b), are respectively examples of undirected graphs and directed graphs. If G is a directed graph, the sum of the lengths of all adjacency lists (the arrays in the image above) will equal |E|. If the graph is undirected however, then it will be 2|E| instead.

Fun fact: the memory cost of an adjacency list representation is always going to be .

**Now gimme matrices!**

They are 2 dimensional arrays, which uses even more memory. Instead of having a set of number representing adjacency, this method instead uses a coordinate representation. If you want to check if a value v is adjacent to a value u, then you check if the vth value on the uth row is a 1. This can be seen in the image of a directed graph matrix to the right. Mind you, if the graph was undirected, you would also be able to tell the adjacency through the uth value on the vth row.

Fun fact: Unlike the list representation, the memory cost of a matrix is always .

**Weighted graphs**

So this is just a side note, but you can actually use a variety of functions to change adjacency, one of which being the weighted function. It’s pretty simple, you have some function w which takes inputs u and v. When calculating the adjacency of v, you use the function w to alter v, which can alter whether it is adjacent. At least, I think that is how it works, the book doesn’t delve too heavily into it, so it does kind of sound like arcane bullshit.

**Breadth-first search**

A breadth-first search algorithm takes inputs and a single source vertex s. The algorithm finds the distance from the source vertex to each other vertex in reach. It should be noted that the distance is defined as smallest number of edges between the source and the target vertex.

From what I can tell, the algorithm finds all vertices in *intervals* of distances, starting from the closest. For example, say you have a source vertex 8 in a graph of 30 vertices. The algorithm can then try first find all vertices within a distance of 1 edge. Assuming source s = 8, and if 8 points to 2 and 17, it will find vertex 2 and 17. The algorithm then expands to find all vertices within 1 to 2 edges. That means it then finds all vertices pointed to by either 2 and/or 17. The algorithm then repeats the process with all vertices found.

To keep track of which vertices have already been found (and prevent possible infinite iterations), the algorithm connects all vertices with 1 of 3 colours. For the sake of this example, undiscovered vertices are marked white, while discovered vertices are either grey or black. Grey represents discovered vertices containing edges pointing to undiscovered vertices. Black vertices only connect to other discovered vertices. When a vertex is discovered, it is highly recommended that you store the distance to it, though I’m not sure if that’s supposed to be inside the vertex itself or if it should be in a hash map.

Oh yeah btw, trees are a thing for this shit. Basically, when doing the search, you can create a tree of which vertices are connected to which. This can pose a few problems because depending on how the tree is structured, it will misrepresent the graph results. The problem you want to avoid when making these trees occur when a vertex has multiple parent vertices (pointed to by other vertices), but also share child vertices with one of the parents. In this case, you are going to want to make the tree in such a way that the vertex is connected with the parent vertex with whom it shares a child, as the vertex and the child can then exist as neighbour vertices.

Side note: this algorithm gives 0 fucks whether the graph is directed or not. Also, the use of G and s are not exclusive to the breadth-first search algorithm. When implementing a graph searching algorithm I highly recommend doing it using the dynamic programming style.

**Depth-first search**

The depth-first search algorithm functions a bit differently from the breadth-first, as it instead of finding all vertices within a specific distance interval, it picks a connected vertex and does the same for the vertices the target vertex is connected to. This repeats until it comes to a vertex with no undiscovered vertices connected, after which it backtracks and tries to find all undiscovered vertices connected to previous vertices. Once all connected vertices have been fully explored, it checks if there still are any undiscovered vertices. If not, great, terminate that shit. If there are, it selects one of them and passes it as a source vertex to repeat the entire process all over.

With regards to the lit af trees, it is possible for a single algorithm to form multiple trees due to the source changing. In such a scenario, the original run of the program forms a forest of trees. In order to help structure the trees, the algorithm gives discovered vertices two timestamps: one for when the vertex itself is discovered, and once again when all of its vertices have also been discovered.

**Directed Acyclic Graph**

So, assume you have a directed graph of some sort. Let’s say with 3 vertices, a, b, and c. A points to both b and c. b points to c. Congrats, you now have a DAG graph, as the graph does not have a looping connection. If it were to be a non DAG graph, a would point to b, b to c, and c, to a, creating and endless loop.

**Topological sorting**

It’s fucking magic. But no really, it involves some shit with numbering vertices based on distance using DFS, and then at the end of one tree, you go all the way back, giving all the vertices on the way values which include the added distance for going back. This can then in some way prevent cycles or some shit.