Introducation to Graph Theory

Definition and Representation

In mathematical and computer science terms, a graph is a collection of values (known as nodes or vertices) and a collection of connections, known as edges. This is frequently represented by endpoints and line segments connecting them. Mathematically a graph is a tuple of two sets, $G(E, V)$, an edge set and a vertex set. Graph functions as a template to abstract many problems, it represents relationships between similar constructs.

For the purposes of complexity analysis, the runtime is often expressed in terms of the size of the E and V sets, similar to how complexities of operations with 1D data structures is often in terms of the length. We know that $|E| \leq |V|$ as in a fully connected graph, the size of the edge set is equal to $|V|(|V| - 1)$. But in programming contests, if unspecified, one should assume $|E| \approx |V|^2$ as they will often throw the worst case at you. (Sidenote: a graph in which $|E| \approx |V|^2$ is known as a *dense* graph, a graph in which $|E| << |V|^2$ is known as a *sparse* graph)

There are many types of graphs, and each of them are used to represent different relationships between values, and their difference is mostly on what the edges can be. To start off there's directed and undirected graphs, in directed graphs the edges are ordered pairs (i.e. an edge from i to j is distinct from an edge from j to i) whereas in undirected graphs an edge is just a connection. One can imagine an undirected edge between i and j to be consisted of two directed edges, one from i to j and the other from j to i. There are further restrictions on whether the edge length is specified, as well as the allowed range of the length, for example, some graphs call for negative edge lengths while others restrict the edge lengths to only positive. For the purposes of this lecture, we are going to deal with directed graphs with positive edge lengths. Further distinction can be made by the nature of the paths, or connected collections of edges, within a graph. A cycle is a path that starts and ends at the same node. An acyclic graph is a graph without cycles. A special type of graph that is often studied is the direct acyclic graph (DAG). Another type of graph that comes up a lot is a tree, which is a directed graph where there exists exactly one path between any two vertices.

I know that we usually don't go into implementation in this class. But the implementation of a graph is important and can sometimes heavily affects the runtime. Graphs are usually represented in two ways:

- Adjacency Matrix: A 2D array of booleans (or edge lengths) where the indices are the nodes connecting the edge. This method has space complexity $O(|V|^2)$
- Edge List: An array of linked lists, where the indices are the vertices and at each index is a list of its neighbors (and optionally the edge length) (i and j are known as neighbors if there exists an edge from i to j). This method has space complexity $O(|V| + |E|)$

There are advantage and disadvantage of both, from a contest perspective, an adjacency matrix is easy to write up, but sometimes consume much more memory than an adjacency list.

Depth First Search and Applications

The most basic algorithm we will study today is the depth first search, otherwise known as depth first transversal. The point of this algorithm is to find a path from one node to another. Let's look through an example together.

We are going to solve a maze. A maze can be represented as a graph where each grid point represent a vertex and if two points are "adjacent," i.e. If you can reach one square from another, then they are connected by an edge in our representation. This is not the best representation of a maze, we can simplify this to only represent the intersections as vertices, but this representation works for our purposes.

The algorithm goes as follows, we keep a list of booleans corresponding to the list of vertices. This list marks if a square has been visited or not. We start by the beginning of the maze.

At each vertex (aka each vertex with more than one edge coming out of it). We will recursively search along each of the paths until it reaches either an exit, in which case we're done, or a dead end, or a visited node. In either of the later cases we go one the next branch, hence earning the algorithm its name **depth first search (DFS)**, because it always going the full depth along each path.

If we are looking to recover the path the algorithm took to get from the source to the destination, we keep a list of predecessor nodes `pred`, where `pred[u] = v` if in our path we got to `u` from `v`.

In psudocode, this is

```
mark all nodes as unvisited
for all node, set pred to -1

DFS(source, target):
    if source == target
        print path by tracing back pred until one reaches -1
    mark source as visited
        for n in neighbors of source where n is unvisited
            pred[n] = source
            DFS(n, target)
```

This algorithm has the complexity of $O(E+V)$, as in the worst case we are guaranteed to visit every vertex, and in doing so we have to traverse every edge. Using the earlier substitution for contests (assuming the graph is dense), the worse case scenario can be estimated to be about $O(V^2)$.

This algorithm is actually very versatile and has profound implications, for example, it can be used to detect if a graph is connected or have several disconnected pieces. This can be done by trying to traverse the graph with no endpoint, only seeking to mark all nodes as visited. If the call returns and there still exists unvisited nodes, those nodes are disconnected from the starting point, allowing us to subdivide an unconnected graph into connected pieces.

Another modification of this algorithm is cycle detection. Recall the few terminating conditions, another one we haven't used yet is reaching a previously visited node. Think about what this means. If we hit a previously visited node, that means there is a cycle in our graph starting at that node. This will also come in handy in problems.

Breath First Search and Meet In The Middle

An alternative method of transversal is known as breath first search or transversal. As its name suggests, unlike depth first search this method takes one step in all possible directions. This is a different algorithm but the basic tenement is the same. As a transversal algorithm, it can do everything depth first search can, including connectedness and cycle detection. Unlike DFS, BFS guarantees an optimal path, as it takes one step in every direction before taking 2 steps in any direction. Therefore the first path it finds is guaranteed to be the shortest.

In breath first search, if the average degree of the graph is p and the true distance between the starting and ending positions is d, then $O(p^d)$ nodes will be traversed, whereas DFS is much more erratic and "luck based." The choice of when to use BFS over DFS is highly dependent on the nature of the graph and d, if d is small but we're looking for one solution in a large graph, BFS is usually preferred. But if we are looking for many solutions (searching based on a criteria) but d is relatively large, then DFS would be preferred.

Meet in the Middle

Let's talk about an interview question. Given the name of two people on Facebook, how does one find their degrees of separation.

This is one of those cases where DFS would not solve the problem for rather obvious reasons. But let's talk about BFS, starting from friend A, if we were to find the distance to friend B via BFS, we will have to traverse $O(p^d)$ people as previously stated.

Sociology theory says that d is bounded by 6, and let's put a lower bound on p as, say, 150. This means that we will need to traverse about $150^6 = 11390625000000$ people. This is not good enough.

Consider, however, that we concurrently conduct BFS from both A and B. Each with their own unique visited-A and visited-B markings. Then if they encounter the visited marking of the other person, we are done. This effectively reduced the complexity of $O(p^d)$ to $O(2 \times p^{d/2})$, which further reduces to $O(p^{d/2})$, a drastic decrease. In our example it reduces the number of people traversed to $2 * 150^3 = 6750000$, which is much, much less.

Although we introduced this technique in the context of graph theory, this technique can be used in a variety of settings. However, this technique is usually known as the "smart brute-force" method, and you should only try to do it when there is no other algorithms.

Let's see an example outside the context of graphs, and yet another popular interview question, the four number problem!

The problem is as follows, you are given a list of integers with length n, find if there exists 4 number a, b, c, d s.t. $a + b + c + d = 0$.

The brute force solution is to try all combination of 4 numbers, which is $O(n^4)$. For those interview oldies, you probably know that "a hashtable makes everything better," so a slightly better solution is to hash all possible $-(a + b + c)$ and check that against the original array.

That last approach is incredibly close to the correct one, but as the name suggests, meet in the **middle** requires using symmetry to our maximum advantage. The insight is that $-(a + b) = (c + d)$. We still use a hashset, and store all possible $-(a + b)$'s, then we iterate through all possible $(c + d)$'s and check them against the subset. This has complexity of $O(n^2)$, which is better than either of the previous ones.