Welcome

Hello, welcome to competitive programming. Today we are going to talk about representing graphs. There are three common representations.

Objectives

The three graph representations you will want to know are

- adjacency matrix
- adjacency list
- edge list

You will want to know the time complexities and tradeoffs for each of these. Different problems will require different representations depending on your memory and time constraints.

Graphs

Graph Vocabulary

As you remember from data structures, a graph is a structure consisting of nodes (or vertices) and edges, each edge connecting two vertices.

Here are some common terms you will need to remember about graphs.

A loop is when an edge connects a vertex to itself.

A graph is a multigraph if there are two vertices with more than one edge between them. We won't see those as often just yet, but they will show up later.

A path is a sequence of connected edges. If there is a path between any two vertices then the graph is said to be connected.

A connected graph with no loops is said to be simple.

Graphs can be directed or undirected, depending on whether the edges are considered to be one way or two way.

A weighted graph is one in which the edges of the graph have a value associated with them. This usually represents cost or distance.

Here is a weird graph for you to look at, and we'll use it in our examples. It is directed and weighted.

Adjacency Matrix

The first representation is an adjacency matrix. If you suppose that each vertex has a distinct index i, then a matrix element (i,j) indicates whether there is an edge between vertex i and j.

If the graph is undirected, then the matrix will be symmetrical, that is entry (i,j) will be the same as entry (j,i).

If the graph is unweighted, then usually we use a 1 for the entries.

The advantage of this representation is that we can check an individual pair for an edge in ${\cal O}(1)$ time. We pay for that though because we have to use ${\cal O}(V^2)$ memory.

If you need to access individual vertexes quickly, and your memory limit is relatively high, then this is likely a good choice. This is also a good choice for dense matrices; those with a lot of edges.

One disadvantage of this structure is that to enumerate the edges you have to make a traversal of the matrix, which will be a ${\cal O}(V^2)$ operation.

Adjacency List

You can save some memory by using a one dimensional vector indexed by the vertices, but the elements are vectors of pairs: each element being a neighboring vertex and the weight of the edge. The memory requirement is ${\cal O}(V + E)$.

This representation allows quick access to individual vertices and also enumerate a vertices neighbors quickly. Since this is a very common operation in our graph algorithms, you should think of this as your default graph representation, and you should work to be efficient at coding this.

One other advantage of this representation is that it allows for multigraphs.

Edge List

The second representation is the edge list. You can keep a linked list or vector of vertex pairs. In this representation, the memory is proportional to the number of edges instead of the number of vertices.

This works well if your matrix is sparse. If you have a problem that involves iterating over edges then this could be a good choice, especially if you need the edges to be sorted by weight. Algorithms like minimum spanning tree work well with this.

The price of this representation is that you no longer have direct access to the vertices; you have to search if you want to access a particular vertex.

Next
Transcript — Bellman Ford and Shortest Path F...