CMSC 351: Floyd's Algorithm

Justin Wyss-Gallifent

November 10, 2023

1 Introduction

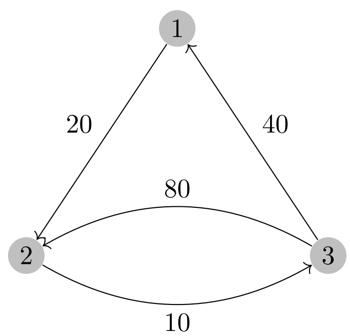
Suppose we have a directed, weighted, simple and connected graph in which both positive and negative (and zero) weights are allowed and we wish to find a shortest path between any two vertices.

The word "path" in this context is a bit different from our definition in that it is allowing repeated edges and vertices. Consequently because we permitting negative weights we could possibly get paths with arbitrarily low (negative) distances if the graph had a cycle with total negative weight since we could just follow the cycle arbitrarily many times.

Since this adds a layer of confusion to the situation we will insist that the graph has no cycles with negative total weight. By doing this the term "shortest path" aligns with our use of the word "path".

2 A Dynamic Example

Consider the following directed, weighted, simple and connected graph:



Here is the adjacency matrix for that graph which we'll denote by d (rather than the typical A). One way to think about the adjacency matrix is that $d[i, j]$ equals the length of a shortest path from i to j not permitting any intermediate vertices.

$$d = \begin{bmatrix} 0 & 20 & \infty \\ \infty & 0 & 10 \\ 40 & 80 & 0 \end{bmatrix}$$

Suppose now for each i, j we also allow 1 to be an intermediate vertex. Observe that the length of a shortest path from 3 to 2 shrinks because we may use 1 as an intermediate vertex. More formally we observe that:

$$d[3, 1] + d[1, 2] < \text{previous } d[3, 2]$$

Note that none of the other shortest paths change because 1 doesn't help anything else. Let's update the matrix accordingly:

$$d[3,2] = d[3,1] + d[1,2]$$

Now for all entries $d[i,j]$ equals the length of a shortest path from i to j permitting intermediate vertex 1. We call this - Pass By 1:

$$d = \begin{bmatrix} 0 & 20 & \infty \\ \infty & 0 & 10 \\ 40 & 60 & 0 \end{bmatrix}$$

Suppose now for each i, j we also allow $1, 2$ to be intermediate vertices. Observe that the length of a shortest path from 1 to 3 shrinks because we may use 2 as an intermediate vertex. More formally we observe that:

$$d[1,2] + d[2,3] < \text{previous } d[1,3]$$

Note that none of the other shortest paths change because 2 doesn't help anything else. Let's update the matrix accordingly:

$$d[1,3] = d[1,2] + d[2,3]$$

Now for all entries $d[i,j]$ equals the length of a shortest path from i to j permitting intermediate vertices $1, 2$. We call this - Pass By (1 and) 2:

$$d = \begin{bmatrix} 0 & 20 & 30 \\ \infty & 0 & 10 \\ 40 & 60 & 0 \end{bmatrix}$$

Suppose now for each i, j we also allow $1, 2, 3$ to be intermediate vertices. Observe that the length of a shortest path from 2 to 1 shrinks because we may use 3 as an intermediate vertex. More formally we observe that:

$$d[2,3] + d[3,1] < \text{previous } d[2,1]$$

Note that none of the other shortest paths change because 3 doesn't help anything else. Let's update the matrix accordingly:

$$d[2,1] = d[2,3] + d[3,1]$$

Now for all entries $d[i,j]$ equals the length of a shortest path from i to j permitting intermediate vertices $1, 2, 3$. We call this - Pass By (1 and 2 and) 3:

$$d = \begin{bmatrix} 0 & 20 & 30 \\ 50 & 0 & 10 \\ 40 & 60 & 0 \end{bmatrix}$$

But now since we're allowing all vertices as intermediate vertices what we've actually obtained is a matrix of shortest paths between all pairs of vertices.

3 Path Storage and Reconstruction

Before we turn this into an algorithm note that the matrix doesn't actually give us the paths, just the lengths of those paths. To obtain the paths let's go back to the start and take another matrix along for the ride.

We define the matrix p which we initialize by setting $p[u, v] = u$ for each edge u to v and by setting $p[v, v] = v$ for each vertex v. All other entries are NULL. Observe that in general $p[u, v]$ is the predecessor of v in a shortest path from u to v not permitting any intermediate vertices:

$$p = \begin{bmatrix} 1 & 1 & NULL \\ NULL & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}$$

During our first update of d above we saw that it was quicker to get from 3 to 2 via 1. The value of $p[3, 2]$ needs to be the precedessor of 2 on the path from 3 to 2 but since we go via 1 we should reassign this to be $p[1, 2]$:

$$p[3, 2] = p[1, 2]$$

Now:

$$p = \begin{bmatrix} 1 & 1 & NULL \\ NULL & 2 & 2 \\ 3 & 1 & 3 \end{bmatrix}$$

Likewise during our second update of d we update:

$$p[1, 3] = p[2, 3]$$

Now:

$$p = \begin{bmatrix} 1 & 1 & 2 \\ NULL & 2 & 2 \\ 3 & 1 & 3 \end{bmatrix}$$

And during our third upate of d we also update:

$$p[2, 1] = d[3, 1]$$

Now:

$$p = \begin{bmatrix} 1 & 1 & 2 \\ 3 & 2 & 2 \\ 3 & 1 & 3 \end{bmatrix}$$

Now that we are done let's look at how to reconstruct a shortest path from p. Consider a shortest path from 3 to 2. Start by initializing $path = [2]$. Then observe that $p[3,2] = 1$ which means 1 is the predecessor of 2 along a shortest path from 3 to 2 and so we *prepend* 1 and now $path = [1,2]$. Then observe that $p[3,1] = 3$ which means 3 is the predecessor of 1 along a shortest path from 3 to 1 and so we prepend 2 and now $path = [3,1,2]$.

4 Algorithm

Each of these updates only uses data from the previous d and so it forms an algorithm nicely.

```
d = adjacency matrix for a graph
p = n x n array all null
for each edge (u,v):
    p[u,v] = u
end for
for each vertex v:
    p[v,v] = v
end for
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            if d[i,k] + d[k,j] < d[i,j]:
                d[i,j] = d[i,k] + d[k,j]
                p[i,j] = p[k,j]
            end if
        end for
    end for
end for
```

The first two `for` loops simply initialize p. Each iteration of k adds another vertex to the list of permissible intermediate vertices and updates the matrices d and p correspondingly.

5 Time Complexity

Starting with an adjacency matrix the first `for` loop is $\Theta(V^2)$ (scanning all entries) and the second `for` loop is $\Theta(V)$. The remaining code is $\Theta(V^3)$ and so overall we have $\Theta(V^3)$.

While this may seem slow this can actually be faster than Dijkstra, for example, if the graph has many many edges.