Lecture 41

All pairs shortest path

Dijkstra's algorithm gave a solution to the single source shortest path problem. That is, it calculated the shortest path from some start vertex to every other vertex in the graph.

Last time, we started to look at an all pairs shortest path solution. Floyd-Warshall uses a $|V|x|V|$ matrix to keep track of shortest path distances between all pairs of nodes. The algorithm then looks for a better path by looking at all possible intermediate nodes.

That is, For every path from V_i to V_j, it checks to see if it knows of a cheaper path by going through an intermediate node V_k. So, if the path distance from V_i to V_k plus the distance from V_k to V_j is less than the known distance from V_i to V_j, then the algorithm updates the matrix with the new information.

The initialization step can be thought of as the shortest paths of length 1. On the next step, all searches for an intermediate node will result in improvement if there is a two edge path that is better than any of the 1 edge paths. Similarly, on the third iteration, it has all two edge path information, and looks at all three edge paths.

The way we recognize a negative weight cycle is to look at the diagonal of the matrix. If the shortest path from a vertex to itself is less than 0, then we know that there is a negative weight cycle in the graph.

Floyd-Warshall with path reconstruction

At each step, we update the better path distances, and we store the intermediate vertex in the `next` matrix. This allows us to store the path for all pairs of vertices in a single 2-D array.

The `next` matrix holds the highest index vertex on a path between V_i and V_j. If the shortest path between V_i and V_j is the edge E_{ij}, then we store a `null` value, or some similar indicator.

Here's how we rebuild the path. If we want to rebuild the path from V_i to V_j, we look at `next[i][j]`, and get the index. If there is an index k stored, then we know that the shortest path from V_i to V_j went through V_k. Next, we look up `next[i][k]` and `next[k][j]`. For each of these paths, if there is an intermediate vertex in the table, we know the path went through that vertex as well. We continue this recursive process until all lookups in `next` result in the `no parent` indicator.

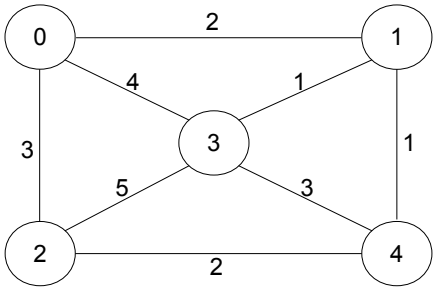
The pseudocode for Floyd-Warshall with path reconstruction appears below.

```
procedure FloydWarshallWithPathReconstruction ()
    for k := 1 to n
        for i := 1 to n
            for j := 1 to n
               if path[i][k] + path[k][j] < path[i][j] then
                  path[i][j] := path[i][k]+path[k][j];
                  next[i][j] := k;

procedure GetPath (i,j)
    if path[i][j] equals infinity then
        return "no path";

    int intermediate := next[i][j];
    if intermediate equals 'null' then
        return " ";   /* there is an edge from i to j, with no vertices between */
    else
        return GetPath(i,intermediate) + intermediate + GetPath(intermediate,j);
```

Example: Below is an example undirected graph used in class to demonstrate F-W:



Here is the initialization matrix:

	0	1	2	3	4
0	0	2	3	4	INF
1	2	0	INF	1	1
2	3	INF	0	5	2
3	4	1	5	0	3
4	INF	1	2	3	0

The below is the resulting next matrix after running the algorithm:

	0	1	2	3	4
0				1	1
1			4		
2		4		4	
3	1		4		1
4	1			1	

We want to rebuild the path from vertex 3 to vertex 2. We begin by looking up `next[3][2]`, which gives us 4. So we look up `next[3][4]`, which gives us 1, and `next[4][2]`, which gives us no parent. So, we know the path from 3 to 2 uses edge $4-2$. Now, we look up `next[3][1]` and `next[1][4]`. Both of these look-ups give no parent. So we are using edges $3-1$ and $1-4$ and we are done because we have no more paths to check.

A few more definitions and terms

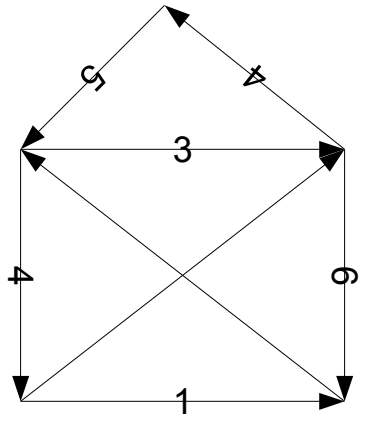
An **Eulerian circuit** or **Euler tour** is an Euler walk that starts and ends at the same vertex. The graph above does not have an Euler tour.

We know this by staring at it for a while and trying to find one. But we can do better. Euler realized (but didn't prove) that:

A graph has an Euler circuit if and only if it is connected and every vertex has even degree.

An **Eulerian trail** or **Euler walk** is a path through a graph that uses every edge exactly once.

The following graph has an euler walk, but not an Euler tour.



Graphs with an Euler walk are said to be **traversable**.

Thm: A graph G is traversable if and only if G is connected and has exactly two vertices of odd degree.

Furthermore, an Euler walk of G begins at one of the odd degree vertices and ends at the other.

A **Hamiltonian path** is a path through a graph that visits each vertex exactly once.

A **Hamiltonian cycle** is a Hamiltonian path that is also a cycle.

A **Complete Graph** is a graph in which all vertices are adjacent to all other vertices.

The number of different Hamiltonian cycles in a complete undirected graph on n vertices is $(n-1)!/2$

Finding the minimum weight Hamiltonian cycle in a graph is known as the traveling salesperson problem.

Traveling Salesperson Problem

We're going to finish up by looking at a graph problem known as the Traeling Salesperson Problem (TSP).

The essence of TSP is this:

Given a list of cities and their pairwise distances, the task is to find the shortest possible route that visits each city exactly once and returns to the origin city.

There is currently no polynomial time algorithm for solving TSP.