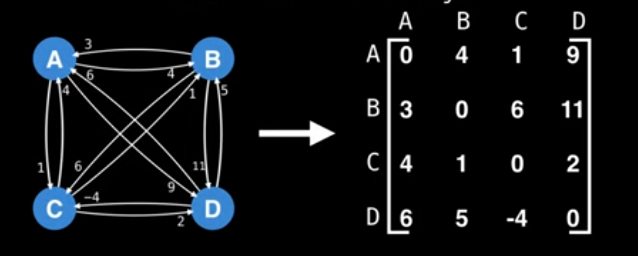
Floyd Warshall Algorithm Oleh Halim Kurniawan

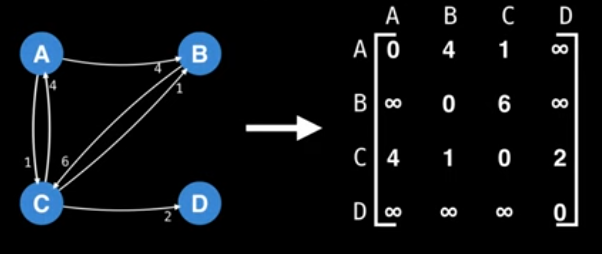
Objective is to find the shortest between all pairs of nodes in a weighted graph, All pairs shortest path. This Algorithm is good for graphs no larger than a couple hundred nodes and also this algorithm can detect negative cycles.

With FW (Floyd Warshall), the optimal way to represent our graph is with a 2D adjacency matrix m where cell m[i][j] represents the edge weight of going from node i to node j.



Note: That a distance from a node to itself is 0 so because of that the diagonal is 0 as pictured above.

If there is no edge from node I to node j then set the edge value for m [i][j] to be positive infinity.

Note: If your programming language DOES NOT support a special constant for +∞ such that ∞ + ∞ = ∞ and x + ∞ = ∞ then AVOID using 231-1 as infinity! Doing such would cause an integer overflow; use a large constant instead like 107 ..

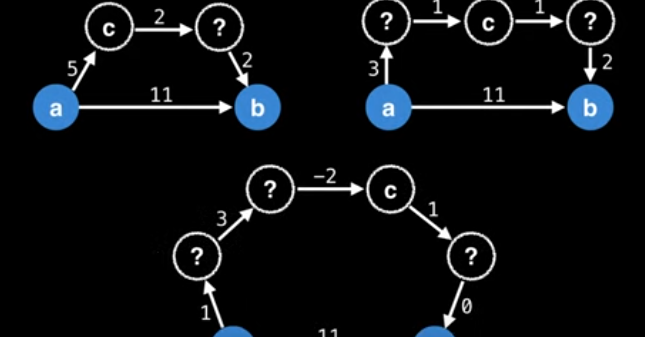
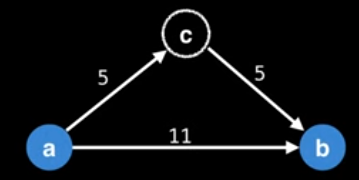
The main idea is to gradually build up all intermediate routed between nodes I and j to find the optimal path.



Suppose our adjacency matrix tells us that the distance from a to b is: m[a][b] = 11

Suppose there exists a third node, c.

If m[a][c] + m[c][b] < m[a][b] then its better to route trough c! The point is to find al possible intermediate nodes towards the destination, it doesn’t matter if it has to go trough multiple different intermediate nodes, as long as its less than the others (AKA shorter) that way!



So how do we actually calculate every intermediate nodes? Trough dynamic progamming (DP),

Let DP be a 3D matrix of size n x n x n that acts as a memo table.

dp[k][i][j] = shortest path from I to j routing trough nodes {0, 1, … k-1}

Using dynamic programing to cache previous solutions.

Start with k = 0, then k = 1, then k = 2, …. This gradually builds up the optimal solutions routing trough 0 , then all optimal solutions routing trough 0 and 1, then all optimal solutions trough 0, 1, 2, etc… up until n-1 which stores to APSP solution.

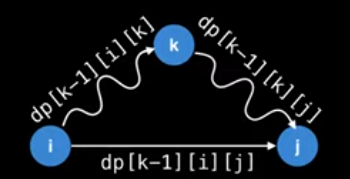
Specifically dp[n-1] is the 2D matrix solution we’re after.

In the beginning the optimal solution from I to j is simply the distance in the adjacency matrix.

dp[k][i][j] = m[i][j] if k = 0

otherwise:

dp[k][i][j] = min(dp[k-1][i][j], dp[k-1][i][j]+dp[k-1][k][j])

This is what it visually looks like

Start at I, route trough some nodes, get to K, and from K route back trough J.

|  |  |  |
| --- | --- | --- |
| Algorithm | Time Complexity | Space Complexity |
|  | |  |  |  | | --- | --- | --- | | Best | Average | Worst | |  |
| FloydWarshall | |  |  |  | | --- | --- | --- | | O(V3) | O(V3) | O(V3) | | O(V2) |