All-pairs shortest path with Floyd-Warshall

Sometimes, we have a graph and want to find the shortest path from u to v for all pairs of vertices (u, v). Note that we already know how to do this in $O(n^3)$ time – we can run the quadratic version of Dijkstra's algorithm from each of the n vertices. However, there is a 4-line algorithm that will do the same job.

Example 1: Floyd-Warshall

```
int graph[128][128], n;                // a weighted graph and its size

void floydWarshall() {
    for( int k = 0; k < n; k++ )
    for( int i = 0; i < n; i++ )
    for( int j = 0; j < n; j++ )
        graph[i][j] = min( graph[i][j], graph[i][k] + graph[k][j] );
}

int main {
    // initialize the graph[][] adjacency matrix and n
    // graph[i][i] should be zero for all i.
    // graph[i][j] should be "infinity" if edge (i, j) does not exist
    // otherwise, graph[i][j] is the weight of the edge (i, j)

    floydWarshall();

    // now graph[i][j] is the length of the shortest path from i to j
}
```

The prettiest way to format the 'for' loops and the best number of curly braces to use are hot topics for discussion because the algorithm itself is so simple. First, let's look at the initialization. graph[i][j] should be set to the weight of the edge (i, j). For non-existent edges, it should be set to "infinity". However, because in the last line of floydWarshall() we add together two entries in the graph[i][j] matrix, the "infinity" value we pick should not be larger than INT_MAX/2 to avoid overflow. The algorithm itself takes two inputs – the adjacency matrix, graph[][], and the number of vertices, n. After the execution of floydWarshall(), graph[][] is modified to contain the lengths of the shortest paths between all pairs of vertices. This works for both directed and undirected graphs.

The order of the 3 'for' loops is important. Doing them in the order "i, j, k" will not work. Here is a simple proof that the order "k, i, j" does work. Define D(k, i, j) to be the length of the shortest path from i to j that only uses vertices 0 through k (in addition to vertices i and j themselves). For example, D(3, 6, 7) is the length of the shortest path from 6 to 7 if we are only allowed to start at 6, go through vertices 0, 1, 2, and 3 in some order and end up at 7. We might not need to visit all of 0, 1, 2 and 3.

To prove correctness of Floyd-Warshall, we will demonstrate the following loop invariant: *after the k-th iteration of the outer loop has finished, graph[i][j] contains D(k, i, j) for all i and j.* Before the start of the algorithm, the invariant is true because graph[i][j] contains the length of the shortest path from i to j that only uses vertices i and j, with no intermediate vertices. Suppose the invariant is true after k-1 iterations. Then during iteration k, graph[i][j] will either not change (if the shortest path does not need to use vertex k), or it will change to graph[i][k] + graph[k][j] if it improves the distance. graph [i][k] is the length of the shortest path form i to k that uses vertices 0 through k, and graph[k][j] is the length of the shortest path from k to j that uses vertices 0 through k. Hence, graph[i][j] becomes the best of two alternatives – its old value (if vertex k does not help) or the shortest path from i to k to j (if vertex k does help).

By induction, the loop invariant holds during each iteration of the outer loop. Hence, after the outer loop has finished, graph[i][j] contains the length of the shortest path from i to j that is allowed to use **all** existing vertices from 0 to n-1. This shows the correctness of Floyd-Warshall.

What if we have an unweighted graph and are simply interested in the question, "Is there a path from i to j?" We can use Floyd-Warshall to solve this question easily.

Example 2: Graph reachability with Floyd-Warshall

```
bool graph[128][128], n;

int reachability() {
    for( int k = 0; k < n; k++ )
    for( int i = 0; i < n; i++ )
    for( int j = 0; j < n; j++ )
        graph[i][j] = graph[i][j] || (graph[i][k] && graph[k][j]);
}
```

If you are a minimalist, the parentheses around "&&" are unnecessary. After reachability() returns, graph[i][j] will be true if there exists a path from i to j. We will come back to the reachability problem later and show that it can solved much faster, but the advantage of this method is its extreme brevity.

Now we know how to get the length of the shortest paths, but what if we want to recover the path itself? We can figure it out from the resulting graph[][] matrix. Suppose that we are at i and we want to go to j. There is an edge from i to k of length L[i][k]. Furthermore, suppose that

graph[i][j] = L[i][k]+ graph[k][j].

This guarantees that if we go from i to k, we will be following a shortest path to j. Now we simply scan all neighbours of i and find such a vertex k, and recurse on that vertex. This method requires no additional memory, but is a bit slow.

Alternately, we can store a parent[][] array much like in Dijkstra's or in Bellman-Ford.

Example 3: Floyd-Warshall with path recovery

```
const int Inf = INT_MAX/2 – 1;      // graph[i][j] = Inf if no edge
int graph[128][128], n;             // a weighted graph and its size
int parent[128][128];

void floydWarshall() {
    for( int i = 0; i < n; i++ )
    for( int j = 0; j < n; j++ )
        if ( i == j || graph[i][j] == Inf ) parent[i][j] = -1;
        else parent[i][j] = i;

    for( int k = 0; k < n; k++ )
    for( int i = 0; i < n; i++ )
    for( int j = 0; j < n; j++ ) {
        int newD = graph[i][k] + graph[k][j];
        if( newD < graph[i][j] ) {
            graph[i][j] = newD;
            parent[i][j] = parent[k][j];
        }
    }
}
```

parent[i][j] contains a vertex number. The meaning of parent[i][j] is this: "For some shortest path from i to j, the vertex right before j is parent[i][j]." We initialize all entries parent[i][j] to i when we have an edge, and if not we set it to -1. The update process of the parent array basically mean the following: "If going from i to j through k (i → k → j) is an improvement, then we set the parent of j in the new shortest path to the parent of the path k → j." The rest of the code is the same as before. To recover the path from i to j, simply recurse on parent[i][j]'s entries.

Powers of the adjacency matrix

Suppose that we have an unweighted graph represented as an adjacency matrix of zeroes and ones. It is a square matrix, so we can multiply it by itself. What will that give us? Matrix multiplication works like this: if we have n-by-n matrices A and B, then C=AB is computed by the following algorithm.

Example 4: Matrix multiplication

```
for( int i = 0; i < n; i++ )
for( int j = 0; j < n; j++ ) {
   C[i][j] = 0;
   for( int k = 0; k < n; k++ )
       C[i][j] += A[i][k] * B[k][j];
}
```

If both A and B are our adjacency matrix M, then C is the square of M, and its entries are

$$C[i][j]=\sum_k M[i][k]*M[k][j] \quad .$$

If we think of M[u][v] as the number of walks of length 1 from u to v, then C[i][j] becomes the number of walks of length exactly 2 from i to j. This is because we try all intermediate vertices k and add up the number of walks of length 1 from i to k times the number of walks of length 1 from k to j. This extends to higher powers of M. In general, $M^p[i][j]$ is the number of walks of length exactly p from i to j. Note that these are walks, not paths. Nowhere is it guaranteed that we are not using the same vertex twice. We can extend this definition to the 0th power of M. Which vertices are reachable from vertex i via a path of length 0? Only vertex i itself. Hence, M^0 is the identity matrix.

Powers of adjacency matrix are useful, for example, if we need to compute the set of vertices reachable from some source vertex s in exactly p steps. To do this, we can calculate M^p and find all v for which $M^p[s][v]$ is non-zero (we have at least one walk of length p from s to v). In this case, it is better to use a boolean matrix and replace addition by "boolean or" (the || operator) when computing matrix products. Otherwise, even for relatively small values of p, the number of paths can be so huge that it will cause overflow. Consider for instance a complete graph. Its adjacency matrix contains all ones (except the zeroes on the diagonal). The p'th power of M may contain entries larger than $(n-1)^{p-1}$.

It takes $O(n^3)$ time to multiply two matrices (just look at example 4). Therefore, computing the p'th power of M using the naive method requires $O(pn^3)$ time. If we are only interested in vertices reachable in p steps from s, then we only need to compute the s'th row of M^p . To do this, we can take the s'th row of the identity matrix (M^0) and multiply it p times by M. A single row is a 1-by-n matrix, so each multiplication will require $O(n^2)$ time for a total of $O(pn^2)$. Please note that there are faster methods of multiplying matrices, and indeed much research has been dedicated to multiplying matrices efficiently. However, these methods are out of scope for our course, and we will not discuss them here.