CMSC 451: Lecture 13
All-Pairs Shortest Paths and the Floyd-Warshall Algorithm
Tuesday, Oct 17, 2017

Reading: Section 6.6 in DPV. Not covered in KT.

All-Pairs Shortest Paths: Earlier, we saw that Dijkstra's algorithm and the Bellman-Ford algorithm both solved the problem of computing shortest paths in graphs from a single source vertex. Suppose that we want instead to compute shortest paths between *all pairs* of vertices. We could do this applying either Dijkstra or Bellman-Ford using every vertex as a source, but today we will consider an alternative approach, which is based on dynamic programming.

Let $G = (V, E)$ be a directed graph with edge weights. For each edge (u, v) E, let $w(u, v)$ denote its weight. The *cost* of a path is the sum of its edge weights, and the *distance* between two vertices, denoted $\delta(u, v)$, minimum cost of any path between u and v. As in Bellman-Ford, we allow negative cost edges, but no negative-cost cycles. (Recall that negative-cost cycles imply that shortest paths are not defined.)

The algorithm that we will present is called the *Floyd-Warshall algorithm*. It runs in $O(n^3)$ time, where $n = |V|$. It dates back to the early 1960's. The algorithm can be adapted for use in a number of related applications as well.

Transitive Closure: You are given a *binary relation* R on a set X, by which we mean that R is a subset of ordered pairs $(x, y) \subseteq X \times X$. A relation is said to be *transitive* if for any $x, y, z \in X$, if $(x, y) \in R$ and $(y, z) \in R$ then $(x, z) \in R$. The *transitive closure* of R, denoted R^* is the smallest extension of R that is transitive.

We can think of (X, R) as a directed graph, where X are the vertices and the pairs of R are edges. The transitive closure of R is effectively the same as the *reachability* relation in this graph, that is, $(x, y) \in R^*$ if and only if there exists a path from x to y in R. The Floyd-Warshall algorithm can be modified to compute the transitive closure in time $O(n^3)$, where $n = |X|$.

All-Pairs Max-Capacity Paths: Let $G = (V, E)$ be a directed graph with positive edge capacities $c(u, v)$. Think of each edge as a pipe, and the capacity $c(u, v)$ as the amount of flow that can be pushed through this pipe per unit interval. The *capacity* of a path is the minimum capacity of any edge along the path. (Intuitively, the minimum capacity edge forms a bottleneck, which limits the total amount of flow along the path. By the way, the capacity of the trivial path from u to u is $+\infty$.)

Given any two nodes u and v, we would like to know the maximum capacity path between them. It is easy to modify the Floyd-Warshall algorithm to compute the maximum capacity path between every pair of vertices in $O(n^3)$ time, where $n = |V|$.

Input/Output Representation: We assume that the digraph is represented as an adjacency matrix, rather than the more common adjacency list. (Adjacency lists are generally more efficient for sparse graphs, but storing all the inter-vertex distances will require $\Omega(n^2)$ storage anyway.) Because the algorithm is matrix-based, we will employ common matrix notation, using i, j and k to denote vertices rather than u, v, and w as we usually do.

The input is an $n \times n$ matrix w of edge weights, which are based on the edge weights in the digraph. We let w_{ij} denote the entry in row i and column j of w.

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ +\infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

Setting $w_{ij} = \infty$ if there is no edge, intuitively means that there is no direct link between these two nodes, and hence the direct cost is infinite. The reason for setting $w_{ii} = 0$ is that there is always a trivial path of length 0 (using no edges) from any vertex to itself.

The output will be an $n \times n$ distance matrix $D = d_{ij}$ where $d_{ij} = \delta(i,j)$, the shortest path cost from vertex i to j. In order to recover the shortest path, we will also compute a helper matrix helper$[i,j]$. The value of helper$[i,j]$ will be any vertex that is midway along the shortest path from i to j. If the shortest path travels directly from i to j without passing through any other vertices, then helper$[i,j]$ will be set to *null*. (Later we will see how to use these values to compute the final path.)

Floyd-Warshall Algorithm: As with any DP algorithm, the key is reducing a large problem to smaller problems. A natural way of doing this is to follow the approach of Bellman-Ford of constructing shortest paths based on the number of edges in the path. The first iteration finds all shortest paths of length one, the next finds shortest paths of length two, and so on. It turns out that this does not lead to the fastest algorithm, however. (A more efficient variant finds shortest paths by doubling, first paths of length one, then two, then four, etc. However, this is still not the fastest.)

Rather than restricting the number of edges on the path, the trick is to restrict the set of vertices through which the path is allowed to pass. Given a path $p = \langle v_1, v_2, \ldots, v_\ell \rangle$ we refer to the vertices $v_2, \ldots, v_{\ell-1}$ as the *intermediate vertices* of this path. Note that a path consisting of a single edge has no intermediate vertices.

- Define $d_{ij}^{(k)}$ to be the shortest path from i to j such that any intermediate vertices on the path are chosen from the set $\{1, \ldots, k\}$.

In other words, we consider a path from i to j which either consists of the single edge (i,j), or it visits some intermediate vertices along the way, but these intermediate can only be chosen from among $\{1, \ldots, k\}$. (It does not need to visit all of these vertices, and it may visit none of them.) The path is free to visit any subset of these vertices, and to do so in any order. For example, in the digraph shown in the Fig. 1(a), notice how the value of $d_{5,6}^{(k)}$ changes as k varies.

How do we compute $d_{ij}^{(k)}$ assuming that we have already computed the previous matrix $d^{(k-1)}$? For the basis case ($k = 0$) the path cannot go through any intermediate vertices, and so $d_{ij}^{(0)} = w(i,j)$ for all i, j. For the induction step ($k \geq 1$), there are two cases, depending on the ways that we might get from vertex i to vertex j, assuming that the intermediate vertices are chosen from $\{1, 2, \ldots, k\}$:

Don't go through k at all: The shortest path from node i to node j uses intermediate vertices $\{1, \ldots, k-1\}$, and hence the length of the shortest path is $d_{ij}^{(k-1)}$.

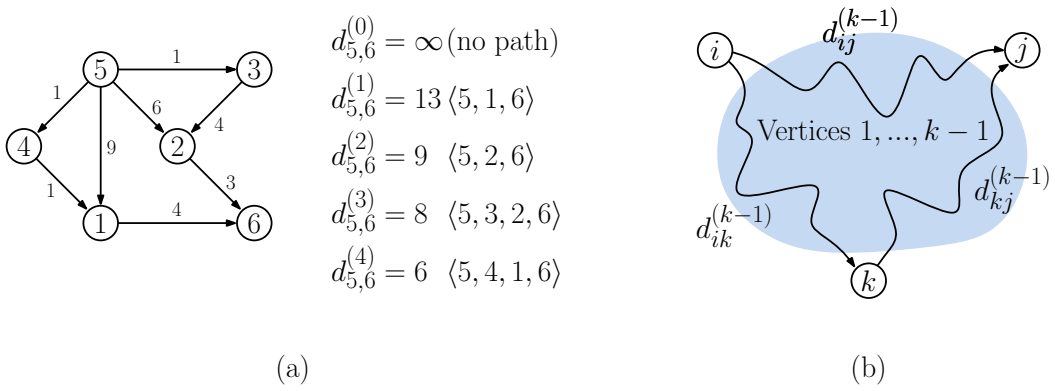

Fig. 1: Limiting intermediate vertices. For example $d^{(3)}_{5,6}$ can go through any combination of the intermediate vertices $\{1, 2, 3\}$, of which $\langle 5, 3, 2, 6\rangle$ has the lowest cost of 8.

Go through k: First observe that a shortest path does not pass through the same vertex twice, so we can assume that we pass through k exactly once. (The assumption that there are no negative cost cycles is being used here.) That is, we go from i to k, and then from k to j. In order for the overall path to be as short as possible we should take the shortest path from i to k, and the shortest path from k to j. Since both of these paths use intermediate vertices only in $\{1, \ldots, k-1\}$, the length of the path is $d^{(k-1)}_{ik} + d^{(k-1)}_{kj}$.[1]

From the above discussion, we have the following recursive rule (the DP formulation) for computing $d^{(k)}$, which is illustrated in Fig. 1(b).

$$\begin{aligned} d^{(0)}_{ij} &= w_{ij}, \\ d^{(k)}_{ij} &= \min\left(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj}\right) \qquad \text{for } k \geq 1. \end{aligned}$$

The final answer is $d^{(n)}_{ij}$, as this allows all possible vertices as intermediate vertices.

As with other DP problems, a recursive implementation of this rule would be prohibitively slow because the same values may be reevaluated many times. Instead, we store the computed values in a table. An honest implementation of the above rule would involve a 3-dimensional array, $d[i, j, k]$. However, a careful analysis of the algorithm reveals that the third dimension does not need to be explicitly stored. We will present the simpler version of the algorithm (which omits the kth dimension) and leave it as an exercise that the algorithm is still a correct implementation of the above rule.

The complete algorithm is presented in the code fragment below. We have also included helper values, helper$[i, j]$, that will be useful for extracting the final shortest paths. Recall that it stores any vertex along the shortest path from i to j. If the path goes through k, then we can store k as the helper. An example of the algorithm's execution is shown in Fig. 2.

[1] Although the figure suggests that $i \neq j \neq k$, you should convince yourself that this holds even if some combination of i, j, and k are equal to each other. For example, if $j = k$, then $d^{(k-1)}_{kj} = d^{(k-1)}_{jj} = w(j, j) = 0$, and so the formula degenerates to $d^{(k-1)}_{ik} + d^{(k-1)}_{kj} = d^{(k-1)}_{ik} = d^{(k-1)}_{ij}$.

Floyd-Warshall Shortest-Path Algorithm

```
floyd-warshall(n, w) {
    d = array [1..n, 1..n]                  // distance matrix
    for (i = 1 to n) {                      // initialize
        for (j = 1 to n) {
            d[i, j] = w[i, j]
            helper[i, j] = null
        }
    }
    for (k = 1 to n) {                      // use intermediates {1..k}
        for (i = 1 to n) {                  // ...from i
            for (j = 1 to n) {              // ...to j
                if (d[i, k] + d[k, j]) < d[i, j]) {
                    d[i, j] = d[i, k] + d[k, j] // new shorter path length
                    helper[i, j] = k          // new path is through k
                }
            }
        }
    }
    return d                                // d[i,j] holds the distance from i to j
}
```

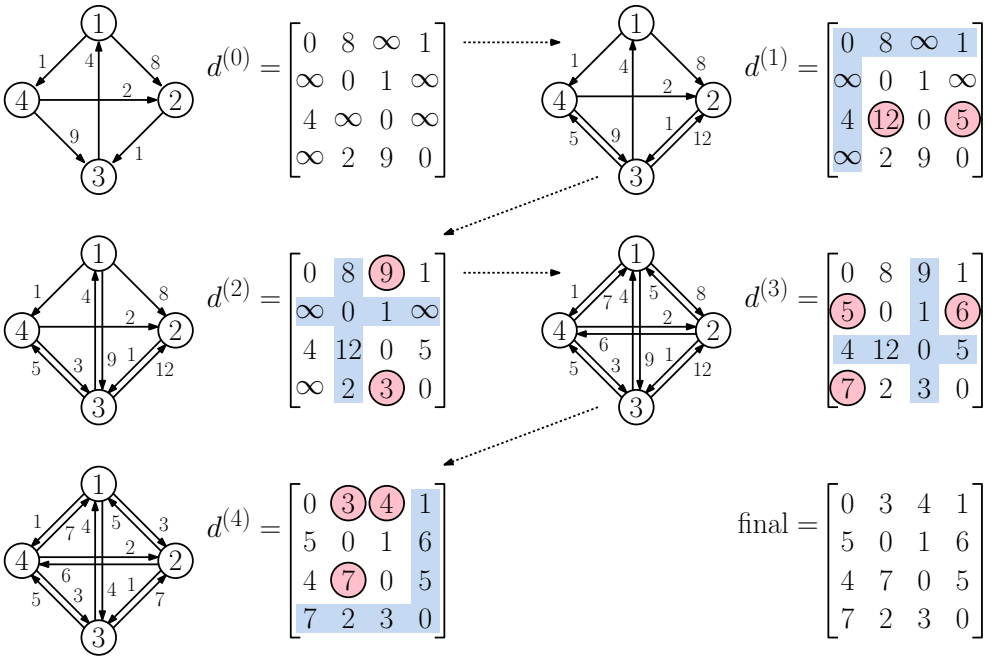

Fig. 2: Floyd-Warshall example. Newly updates entries are circled.

Clearly the algorithm's running time is $O(n^3)$. The space used by the algorithm is $O(n^2)$. Observe that we deleted all references to the superscript (k) in the code. It is left as an exercise that this does not affect the correctness of the algorithm. (Hint: The danger is that values may be overwritten and then used later in the same phase. Consider which entries might be overwritten and then reused, they occur in row k and column k. It can be shown that the overwritten values are equal to their original values.)

Extracting the Shortest Path: Let's next see how to use the helper values helper$[i, j]$ to extract the shortest path. Recall that whenever we discover that the shortest path from i to j passes through an intermediate vertex k, we set helper$[i, j] = k$. If the shortest path does not pass through any intermediate vertex, then helper$[i, j] = \textit{null}$. To find the shortest path from i to j, we consult helper$[i, j]$. If it is *null*, then the shortest path is just the edge (i, j). Otherwise, we recursively compute the shortest path from i to helper$[i, j]$ and concatenate this with the shortest path from helper$[i, j]$ to j.

Printing the Shortest Path

```
get-path(i, j) {
    if (helper[i, j] == null)               // path is a single edge
        output(i, j)
    else {                                  // path goes through helper
        get-path(i, helper[i, j])           // output the path from i to helper
        get-path(helper[i, j], j)           // output the path from helper to j
    }
}
```