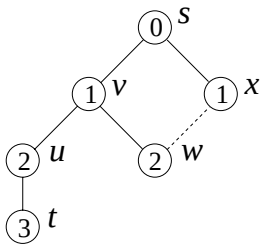

Figure 27: BFS tree.

For an directed graph the analysis is essentially the same.

Lecture 23: All-Pairs Shortest Paths

(Tuesday, April 21, 1998)

Read: Chapt 26 (up to Section 26.2) in CLR.

All-Pairs Shortest Paths: Last time we showed how to compute shortest paths starting at a designated source vertex, and assuming that there are no weights on the edges. Today we talk about a considerable generalization of this problem. First, we compute shortest paths not from a single vertex, but from every vertex in the graph. Second, we allow edges in the graph to have numeric *weights*.

Let $G = (V, E)$ be a directed graph with edge weights. If (u, v) E, is an edge of G, then the weight of this edge is denoted $W(u, v)$. Intuitively, this weight denotes the distance of the road from u to v, or more generally the cost of traveling from u to v. For now, let us think of the weights as being positive values, but we will see that the algorithm that we are about to present can handle negative weights as well, in special cases. Intuitively a negative weight means that you get paid for traveling from u to v. Given a path $\pi = \langle u_0, u_1, \ldots, u_k \rangle$, the *cost* of this path is the sum of the edge weights:

$$cost(\pi) = W(u_0, u_1) + W(u_1, u_2) + \cdots W(u_{k-1}, u_k) = \sum_{i=1}^{k} W(u_{i-1}, u_i).$$

(We will avoid using the term *length*, since it can be confused with the number of edges on the path.) The *distance* between two vertices is the cost of the minimum cost path between them.

We consider the problem of determining the cost of the shortest path between all pairs of vertices in a weighted directed graph. We will present two algorithms for this problem. The first is a rather naive $\Theta(n^4)$ algorithm, and the second is a $\Theta(n^3)$ algorithm. The latter is called the *Floyd-Warshall algorithm*. Both algorithms is based on a completely different algorithm design technique, called *dynamic programming*.

For these algorithms, we will assume that the digraph is represented as an adjacency matrix, rather than the more common adjacency list. Recall that adjacency lists are generally more efficient for sparse graphs (and large graphs tend to be sparse). However, storing all the distance information between each pair of vertices, will quickly yield a dense digraph (since typically almost every vertex can reach almost every other vertex). Therefore, since the output will be dense, there is no real harm in using the adjacency matrix.

Because both algorithms are 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. Let $G = (V, E, w)$ denote the input digraph and its edge weight function. The edge weights may be positive, zero, or negative, but we assume that

there are no cycles whose total weight is negative. It is easy to see why this causes problems. If the shortest path ever enters such a cycle, it would never exit. Why? Because by going round the cycle over and over, the cost will become smaller and smaller. Thus, the shortest path would have a weight of $-\infty$, and would consist of an infinite number of edges. Disallowing negative weight cycles will rule out the possibility this absurd situation.

Input Format: 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 intuitively the cost of getting from any vertex to should be 0, since we have no distance to travel. Note that in digraphs it is possible to have self-loop edges, and so $W(i,j)$ may generally be nonzero. Notice that it cannot be negative (otherwise we would have a negative cost cycle consisting of this single edge). If it is positive, then it never does us any good to follow this edge (since it increases our cost and doesn't take us anywhere new).

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. Recovering the shortest paths will also be an issue. To help us do this, we will also compute an auxiliary matrix $pred[i,j]$. The value of $pred[i,j]$ will be a vertex that is somewhere 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 $pred[i,j]$ will be set to *null*. We will see later than using these values it will be possible to reconstruct any shortest path in $\Theta(n)$ time.

Dynamic Programming for Shortest Paths: The algorithm is based on a technique called *dynamic programming*. Dynamic programming problems are typically optimization problems (find the smallest or largest solution, subject to various constraints). The technique is related to divide-and-conquer, in the sense that it breaks problems down into smaller problems that it solves recursively. However, because of the somewhat different nature of dynamic programming problems, standard divide-and-conquer solutions are not usually efficient. The basic elements that characterize a dynamic programming algorithm are:

Substructure: Decompose the problem into smaller subproblems.

Optimal substructure: Each of the subproblems should be solved optimally.

Bottom-up computation: Combine solutions on small subproblems to solve larger subproblems, and eventually to arrive at a solution to the complete problem.

The question is how to decompose the shortest path problem into subproblems in a meaningful way. There is one very natural way to do this. What is remarkable, is that this does *not* lead to the best solution. First we will introduce the natural decomposition, and later present the Floyd-Warshall algorithm makes use of a different, but more efficient dynamic programming formulation.

Path Length Formulation: We will concentrate just on computing the *cost* of the shortest path, not the path itself. Let us first sketch the natural way to break the problem into subproblems. We want to find some parameter, which constrains the estimates to the shortest path costs. At first the estimates will be crude. As this parameter grows, the shortest paths cost estimates should converge to their correct values. A natural way to do this is to restrict the number of edges that are allowed to be in the shortest path.

For $0 \leq m \leq n-1$, define $d_{ij}^{(m)}$ to be the cost of the shortest path from vertex i to vertex j that contains at most m edges. Let $D^{(m)}$ denote the matrix whose entries are these values. The idea is to

compute $D^{(0)}$ then $D^{(1)}$, and so on, up to $D^{(n-1)}$. Since we know that no shortest path can use more than $n-1$ edges (for otherwise it would have to repeat a vertex), we know that $D^{(n-1)}$ is the final distance matrix. This is illustrated in the figure (a) below.

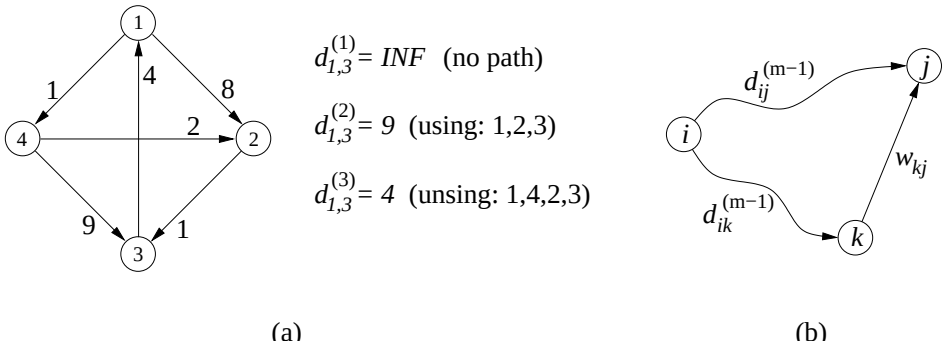

Figure 28: Dynamic Programming Formulation.

The question is, how do we compute these distance matrices? As a basis, we could start with paths of containing 0 edges, $D^{(0)}$ (as our text does). However, observe that $D^{(1)} = W$, since the edges of the digraph are just paths of length 1. It is just as easy to start with $D^{(1)}$, since we are given W as input. So as our basis case we have

$$d^{(1)}_{ij} = w_{ij}.$$

Now, to make the induction go, we claim that it is possible to compute $D^{(m)}$ from $D^{(m-1)}$, for $m \geq 2$. Consider how to compute the quantity $d^{(m)}_{ij}$. This is the length of the shortest path from i to j using at most m edges. There are two cases:

Case 1: If the shortest path uses strictly fewer than m edges, then its cost is just $d^{(m-1)}_{ij}$.

Case 2: If the shortest path uses exactly m edges, then the path uses $m-1$ edges to go from i to some vertex k, and then follows a single edge (k, j) of weight w_{kj} to get to j. The path from i to k should be shortest (by the principle of optimality) so the length of the resulting path is $d^{(m-1)}_{ik} + w_{ij}$. But we do not know what k is. So we minimize over all possible choices.

This is illustrated in the figure (b) above.

This suggests the following rule:

$$d^{(m)}_{ij} = \min \left\{ \begin{array}{l} d^{(m-1)}_{ij} \\ \min_{1 \leq k \leq n} \left(d^{(m-1)}_{ik} + w_{kj} \right) \end{array} \right\}.$$

Notice that the two terms of the main min correspond to the two cases. In the second case, we consider all vertices k, and consider the length of the shortest path from i to k, using $m-1$ edges, and then the single edge length cost from k to j.

We can simplify this formula a bit by observing that since $w_{jj} = 0$, we have $d^{(m-1)}_{ij} = d^{(m-1)}_{ij} + w_{jj}$. This term occurs in the second case (when $k = j$). Thus, the first term is redundant. This gives

$$d^{(m)}_{ij} = \min_{1 \leq k \leq n} \left(d^{(m-1)}_{ik} + w_{kj} \right),$$

The next question is how shall we implement this rule. One way would be to write a recursive procedure to do it. Here is a possible implementation. To compute the shortest path from i to j, the initial call would be $Dist(n-1, i, j)$. The array of weights

Recursive Shortest Paths

```
Dist(int m, int i, int j) {
    if (m == 1) return W[i,j]                       // single edge case
    best = INF
    for k = 1 to n do
        best = min(best, Dist(m-1, i, k) + w[k,j])  // apply the update rule
    return best
}
```

Unfortunately this will be *very slow*. Let $T(m, n)$ be the running time of this algorithm on a graph with n vertices, where the first argument is m. The algorithm makes n calls to itself with the first argument of $m - 1$. When $m = 1$, the recursion bottoms out, and we have $T(1, n) = 1$. Otherwise, we make n recursive calls to $T(m - 1, n)$. This gives the recurrence:

$$T(m,n) = \begin{cases} 1 & \text{if } m = 1, \\ nT(m-1,n) + 1 & \text{otherwise.} \end{cases}$$

The total running time is $T(n - 1, n)$. It is a straightforward to solve this by expansion. The result will be $O(n^n)$, a huge value. It is not hard to see why. If you unravel the recursion, you will see that this algorithm is just blindly trying all possible paths from i to j. There are exponentially many such paths.

So how do we make this faster? The answer is to use *table-lookup*. This is the key to dynamic programming. Observe that there are only $O(n^3)$ different possible numbers $d_{ij}^{(m)}$ that we have to compute. Once we compute one of these values, we will store it in a table. Then if we want this value again, rather than recompute it, we will simply look its value up in the table.

The figure below gives an implementation of this idea. The main procedure $ShortestPath(n, w)$ is given the number of vertices n and the matrix of edge weights W. The matrix $D^{(m)}$ is stored as $D[m]$, for $1 \le m \le n - 1$. For each m, $D[m]$ is a 2-dimensional matrix, implying that D is a 3-dimensional matrix. We initialize $D^{(1)}$ by copying W. Then each call to $ExtendPaths()$ computes $D^{(m)}$ from $D^{(m-1)}$, from the above formula.

Dynamic Program Shortest Paths

```
ShortestPath(int n, int W[1..n, 1..n]) {
    array D[1..n-1][1..n, 1..n]
    copy W to D[1]                                  // initialize D[1]
    for m = 2 to n-1 do
        D[m] = ExtendPaths(n, D[m-1], W)            // comput D[m] from D[m-1]
    return D[n-1]
}

ExtendShortestPath(int n, int d[1..n, 1..n], int W[1..n, 1..n]) {
    matrix dd[1..n, 1..n] = d[1..n, 1..n]           // copy d to temp matrix
    for i = 1 to n do                               // start from i
        for j = 1 to n do                           // ...to j
            for k = 1 to n do                       // ...passing through k
                dd[i,j] = min(dd[i,j], d[i,k] + W[k,j])
    return dd                                       // return matrix of distances
}
```

The procedure $ExtendShortestPath()$ consists of 3 nested loops, and so its running time is $\Theta(n^3)$. It is called $n - 2$ times by the main procedure, and so the total running time is $\Theta(n^4)$. Next time we will see that we can improve on this. This is illustrated in the figure below.

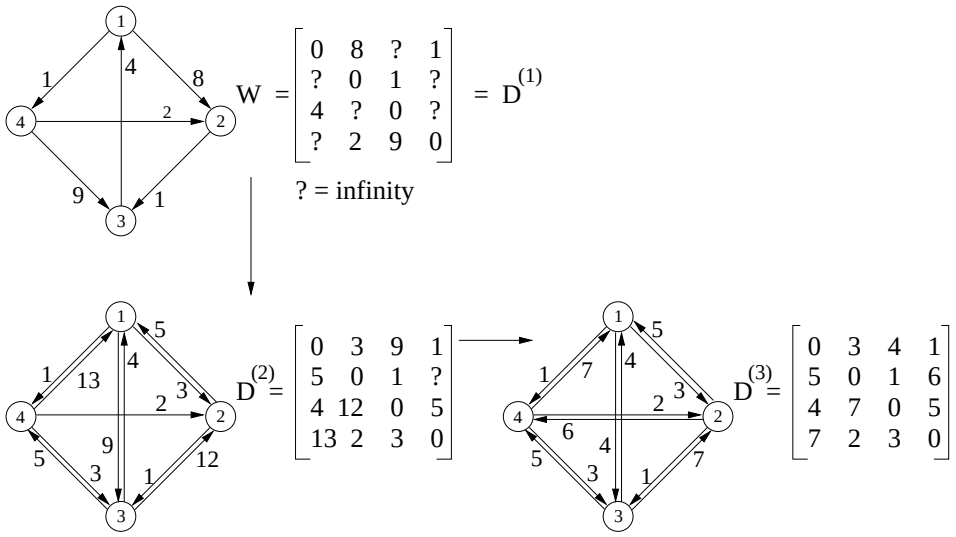

Figure 29: Shortest Path Example.

Lecture 24: Floyd-Warshall Algorithm

(Thursday, April 23, 1998)

Read: Chapt 26 (up to Section 26.2) in CLR.

Floyd-Warshall Algorithm: We continue discussion of computing shortest paths between all pairs of vertices in a directed graph. The Floyd-Warshall algorithm dates back to the early 60's. Warshall was interested in the weaker question of reachability: determine for each pair of vertices u and v, whether u can reach v. Floyd realized that the same technique could be used to compute shortest paths with only minor variations.

The Floyd-Warshall algorithm improves upon this algorithm, running in $\Theta(n^3)$ time. The genius of the Floyd-Warshall algorithm is in finding a different formulation for the shortest path subproblem than the path length formulation introduced earlier. At first the formulation may seem most unnatural, but it leads to a faster algorithm. As before, we will compute a set of matrices whose entries are $d_{ij}^{(k)}$. We will change the *meaning* of each of these entries.

For a path $p = \langle v_1, v_2, \ldots, v_\ell \rangle$ we say that the vertices $v_2, v_3, \ldots, v_{\ell-1}$ are the *intermediate vertices* of this path. Note that a path consisting of a single edge has no intermediate vertices. We 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 $\{1, 2, \ldots, k\}$. 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 $\{1, 2, \ldots, k\}$. The path is free to visit any subset of these vertices, and to do so in any order. Thus, the difference between Floyd's formulation and the previous formulation is that here the superscript (k) restricts the set of vertices that the path is allowed to pass through, and there the superscript (m) restricts the number of edges the path is allowed to use. For example, in the digraph shown in the following figure, notice how the value of $d_{32}^{(k)}$ changes as k varies.

Floyd-Warshall Update Rule: How do we compute $d_{ij}^{(k)}$ assuming that we have already computed the previous matrix $d^{(k-1)}$? As before, there are two basic cases, depending on the ways that we might get from vertex i to vertex j, assuming that the intermediate vertices are chosen from $\{1, 2, \ldots, k\}$: