Design & Analysis of Algorithms (Dynamic Programming)

Soharab Hossain Shaikh
BML Munjal University

1

Transitive Closure

DEFINITION The *transitive closure* of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{t_{ij}\}$, in which the element in the ith row and the jth column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, t_{ij} is 0.

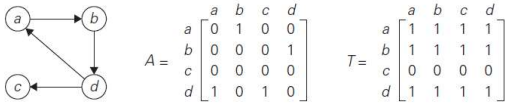


FIGURE (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

2

Warshall's Algorithm

Warshall's algorithm after Stephen Warshall, who discovered it. It is convenient to assume that the digraph's vertices and hence the rows and columns of the adjacency matrix are numbered from 1 to n. Warshall's algorithm constructs the transitive closure through a series of $n \times n$ boolean matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}.$$

Each of these matrices provides certain information about directed paths in the digraph. Specifically, the element $r_{ij}^{(k)}$ in the ith row and jth column of matrix $R^{(k)}$ $(i, j = 1, 2, \ldots, n, k = 0, 1, \ldots, n)$ is equal to 1 if and only if there exists a directed path of a positive length from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k. Thus, the series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph.

The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.

3

Warshall's Algorithm

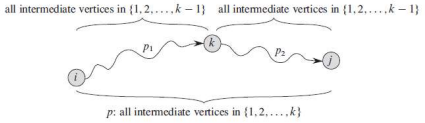
The central point of the algorithm is that we can compute all the elements of each matrix $R^{(k)}$ from its immediate predecessor $R^{(k-1)}$ in series. Let $r_{ij}^{(k)}$, the element in the ith row and jth column of matrix $R^{(k)}$, be equal to 1. This means that there exists a path from the ith vertex v_i to the jth vertex v_j with each intermediate vertex numbered not higher than k:

v_i, a list of intermediate vertices each numbered not higher than k, v_j.

Two situations regarding this path are possible. In the first, the list of its intermediate vertices does not contain the kth vertex. Then this path from v_i to v_j has intermediate vertices numbered not higher than $k-1$, and therefore $r_{ij}^{(k-1)}$ is equal to 1 as well. The second possibility is that path that does contain the kth vertex v_k among the intermediate vertices. Without loss of generality, we may assume that v_k occurs only once in that list. (If it is not the case, we can create a new path from v_i to v_j with this property by simply eliminating all the vertices between the first and last occurrences of v_k in it.) With this caveat, path can be rewritten as follows:

4

Warshall's Algorithm



v_i, vertices numbered $\leq k-1$, v_k, vertices numbered $\leq k-1$, v_j.

The first part of this representation means that there exists a path from v_i to v_k with each intermediate vertex numbered not higher than $k-1$ (hence, $r_{ik}^{(k-1)}=1$), and the second part means that there exists a path from v_k to v_j with each intermediate vertex numbered not higher than $k-1$ (hence, $r_{kj}^{(k-1)}=1$).

What we have just proved is that if $r_{ij}^{(k)}=1$, then either $r_{ij}^{(k-1)}=1$ or both $r_{ik}^{(k-1)}=1$ and $r_{kj}^{(k-1)}=1$. It is easy to see that the converse of this assertion is also true. Thus, we have the following formula for generating the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$:

$$r_{ij}^{(k)}=r_{ij}^{(k-1)} \quad \text{or} \quad \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right).$$

5

Warshall's Algorithm

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$.

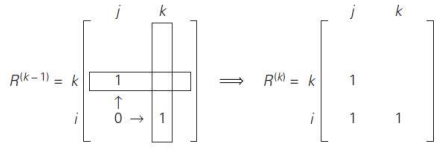


FIGURE Rule for changing zeros in Warshall's algorithm.

6

Warshall's Algorithm

Stephen Warshall in 1962

ALGORITHM *Warshall*$(A[1..n, 1..n])$

```
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
R(0) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R(k)[i, j] ← R(k−1)[i, j] or (R(k−1)[i, k] and R(k−1)[k, j])
return R(n)
```

7

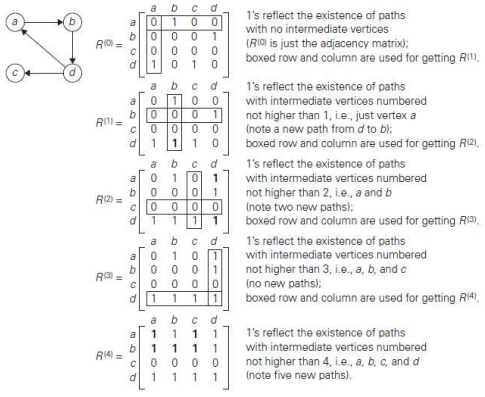


FIGURE Application of Warshall's algorithm to the digraph shown. New 1's are in bold.

8

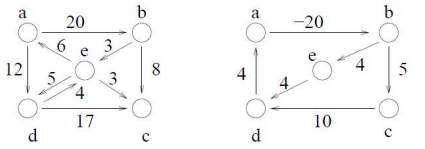
Time & Space Requirements

- The algorithm's running time is clearly $\Theta(n^3)$.
- Problem: the algorithm uses $\Theta(n^3)$ space. It is possible to reduce this down to $\Theta(n^2)$ space by keeping only one matrix instead of n.

9

All Pairs Shortest Path

Given a weighted digraph $G = (V, E)$ with a weight function $w : E \to \mathbf{R}$, where R is the set of real numbers, determine the length of the shortest path (i.e., distance) between all pairs of vertices in G. Here we assume that there are no cycle with zero or negative cost.



without negative cost cycle with negative cost cycle

10

Floyd's Algorithm

The **Floyd–Warshall** algorithm is an example of dynamic programming, and was published in its currently recognized form by **Robert Floyd** in 1962.

However, it is essentially the same as algorithms previously published by **Bernard Roy** in 1959 and also by **Stephen Warshall** in 1962 for finding the transitive closure of a graph.

Closely related to **Kleene's algorithm** (published in 1956) for converting a deterministic finite automaton into a regular expression.

The modern formulation of the algorithm as three nested for-loops was first described by **Peter Ingerman**, also in 1962.

The algorithm is also known as **Floyd's algorithm**, the **Roy–Warshall algorithm**, the **Roy–Floyd algorithm**, or the **WFI algorithm**.

11

Graph Weight Matrix

we assume that the graph is represented by an $n \times n$ matrix with the weights of the edges:

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

Output Format: an $n \times n$ distance $D = [d_{ij}]$ where d_{ij} is the distance from vertex i to j.

12

Problem Decomposition

Definition: The vertices $v_2, v_3, ..., v_{l-1}$ are called the *intermediate vertices* of the path $p = \langle v_1, v_2, ..., v_l \rangle$.

- Let $d_{ij}^{(k)}$ be the length of the shortest path from i to j such that *all* intermediate vertices on the path (if any) are in set $\{1, 2, \ldots, k\}$.

  $d_{ij}^{(0)}$ is set to be w_{ij}, i.e., no intermediate vertex.
  Let $D^{(k)}$ be the $n \times n$ matrix $[d_{ij}^{(k)}]$.

- Claim: $d_{ij}^{(n)}$ is the distance from i to j. So our aim is to compute $D^{(n)}$.

- Subproblems: compute $D^{(k)}$ for $k = 0, 1, \cdots, n$.

13

Structure of Shortest Path

Observation 1:
A shortest path does not contain the same vertex twice.
A path containing the same vertex twice contains a cycle. Removing cycle gives a shorter path.

Observation 2: For a shortest path from i to j such that any intermediate vertices on the path are chosen from the set $\{1, 2, \ldots, k\}$, there are two possibilities:

1. k is not a vertex on the path,
The shortest such path has length $d_{ij}^{(k-1)}$.

2. k is a vertex on the path.
The shortest such path has length $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

14

Structure of Shortest Path

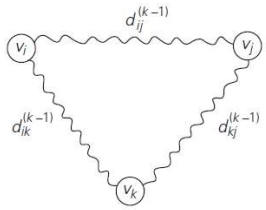


FIGURE Underlying idea of Floyd's algorithm.

15

Structure of Shortest Path

Consider a shortest path from i to j containing the vertex k. It consists of a subpath from i to k and a subpath from k to j.
Each subpath can only contain intermediate vertices in $\{1, ..., k-1\}$, and must be as short as possible, namely they have lengths $d_{ik}^{(k-1)}$ and $d_{kj}^{(k-1)}$.

Hence the path has length $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

Combining the two cases we get

$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}.$$

16

Bottom-Up Computation

- Bottom: $D^{(0)} = [w_{ij}]$, the weight matrix.
- Compute $D^{(k)}$ from $D^{(k-1)}$ using

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

for $k = 1, ..., n$.

17

Floyd-Warshall Algorithm

Floyd-Warshall(w, n)

```
{ for i = 1 to n do                          initialize
    for j = 1 to n do
    { D^0[i,j] = w[i,j];
      pred[i,j] = nil;
    }

  for k = 1 to n do                          dynamic programming
    for i = 1 to n do
      for j = 1 to n do
        if (d^(k-1)[i,k] + d^(k-1)[k,j] < d^(k-1)[i,j])
            {d^(k)[i,j] = d^(k-1)[i,k] + d^(k-1)[k,j];
             pred[i,j] = k;}
        else d^(k)[i,j] = d^(k-1)[i,j];
  return d^(n)[1..n, 1..n];
}
```

18

Floyd-Warshall Algorithm

- The algorithm's running time is clearly $\Theta(n^3)$.
- The predecessor pointer `pred[i, j]` can be used to extract the final path (see later).
- Problem: the algorithm uses $\Theta(n^3)$ space. It is possible to reduce this down to $\Theta(n^2)$ space by keeping only one matrix instead of n. Algorithm is on next page. Convince yourself that it works.

19

Floyd-Warshall Algorithm (Revised)

Floyd-Warshall(w, n)

```
{ for i = 1 to n do                          initialize
    for j = 1 to n do
    { d[i,j] = w[i,j];
      pred[i,j] = nil;
    }

  for k = 1 to n do                          dynamic programming
    for i = 1 to n do
      for j = 1 to n do
        if (d[i,k] + d[k,j] < d[i,j])
            {d[i,j] = d[i,k] + d[k,j];
             pred[i,j] = k;}
  return d[1..n, 1..n];
}
```

20

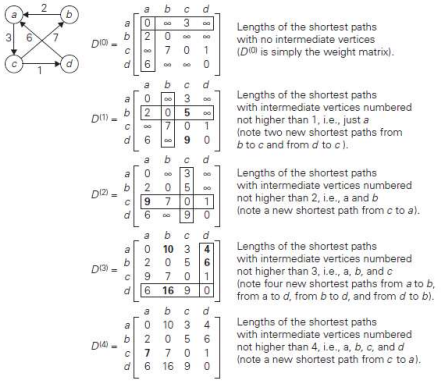


FIGURE Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

21

Extract the Shortest Path

The predecessor pointers `pred[i,j]` can be used to extract the final path. The idea is as follows.

Whenever we discover that the shortest path from i to j passes through an intermediate vertex k, we set $pred[i,j] = k$.

If the shortest path does not pass through any intermediate vertex, then $pred[i,j] = nil$.

To find the shortest path from i to j, we consult $pred[i,j]$. If it is nil, then the shortest path is just the edge (i,j). Otherwise, we recursively compute the shortest path from i to $pred[i,j]$ and the shortest path from $pred[i,j]$ to j.

22

Extract the Shortest Path

```
Path(i, j)
{
    if (pred[i, j] = nil)    single edge
        output (i, j);
    else          compute the two parts of the path
    {
        Path(i, pred[i, j]);
        Path(pred[i, j], j);
    }
}
```

23

End of Lecture

24