SFU CMPT-307 2008-2 Lecture: Week 11

Ján Maňuch

E-mail: jmanuch@sfu.ca

Lecture on July 22, 2008, 5.30pm-8.20pm

Dynamic Programming: Matrix-Chain Multiplication

Given: a "chain" of matrices $(A_1, A_2, \ldots A_n)$, with A_i having dimension $p_{i-1} \times p_i$.

Goal: compute the product $A_1 \cdot A_2 \cdots A_n$ as fast as possible

Clearly, time to multiply two matrices depends on **dimensions**

Does the **order** of multiplication (= *parenthesization*) matter?

Example: $n = 4$. Possible orders:

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

Suppose A_1 is 10×100, A_2 is 100×5, A_3 is 5×50, and A_4 is 50×10

Assume that multiplication of a $(p \times q)$-matrix and a $(q \times r)$-matrix takes pqr steps (a straightforward algorithm)

Order 2: $(A_1((A_2A_3)A_4))$

$$100 \cdot 5 \cdot 50 + 100 \cdot 50 \cdot 10 + 10 \cdot 100 \cdot 10 = 85,000$$

Order 5: $(((A_1A_2)A_3)A_4)$

$$10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 + 10 \cdot 50 \cdot 10 = 12,500$$

Seems it might be a good idea to find a “good” order

How many orders are there? Can we just check all of them?

(we look only at fully parenthesized matrix products)

Let $P(n)$ be the number of orders of a sequence of n matrices

Clear, $P(1) = 1$ (only one matrix)

If $n \geq 2$, a matrix product is the product of two matrix subproducts. Split may occur between k-th and $(k+1)$-st position, for any $k = 1, 2, \ldots, n-1$ ("top-level multiplication")

Thus

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & \text{if } n \geq 2 \end{cases}$$

Unfortunately, $P(n) = \Omega(4^n / n^{3/2})$, and thus (easier to see) $P(n) = \Omega(2^n)$

Thus "brute-force approach" (check all parenthesization) is no good

Assignment Problem 11.1. (deadline: July 29, 5:30pm)
Show that the number of full parenthesizations of a product of n matrices, $P(n)$ is in $\Omega(2^n)$.

We will use the **Dynamic programming** approach to **optimally** solve this problem.

The four basic steps when designing Dynamic programming algorithm:

1. **Characterize the structure** of an optimal solution
2. Recursively **define the value** of an optimal solution
3. **Compute the value** of an optimal solution in a bottom-up fashion
4. **Construct an optimal solution** from computed information

1. Characterizing structure

Let $A_{i,j} = A_i \cdots A_j$ for $i \leq j$.

If $i < j$, then any parenthesization of $A_{i,j}$ must split product at some k, $i \leq k < j$, i.e., compute $A_{i,k}$, $A_{k+1,j}$, and then $A_{i,k} \cdot A_{k+1,j}$.

Hence, for some k, the cost of computing $A_{i,j}$ is

- the cost of computing $A_{i,k}$ plus
- the cost of computing $A_{k+1,j}$ plus
- the cost of multiplying $A_{i,k}$ and $A_{k+1,j}$.

Optimal substructure:

- Suppose that optimal parenthesization of $A_{i,j}$ splits the product between A_k and A_{k+1}.
- Then, parenthesizations of $A_{i,k}$ and $A_{k+1,j}$ within this optimal parenthesization must be also optimal
(otherwise, substitute the opt. parenthesization of $A_{i,k}$ (resp. $A_{k+1,j}$) to current parenthesization of $A_{i,j}$ and obtain a better solution — contradiction)

Use **optimal substructure** to construct an optimal solution:

1. split into two subproblems (choosing an optimal split),
2. find optimal solutions to subproblem,
3. combine optimal subproblem solutions.

2. A recursive solution

Let $m[i, j]$ denote minimum number of scalar multiplications needed to compute $A_{i,j} = A_i \cdot A_{i+1} \cdots A_j$ (full problem: $m[1, n]$).

Recursive definition of $m[i, j]$:

- if $i = j$, then $m[i, j] = m[i, i] = 0$ ($A_{i,i} = A_i$, no multiplication needed).
- if $i < j$, assume optimal split at k, $i \leq k < j$. Since each matrix A_i is $p_{i-1} \times p_i$, $A_{i,k}$ is $p_{i-1} \times p_k$ and $A_{k+1,j}$ is $p_k \times p_j$,

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} \cdot p_k \cdot p_j$$

- We do not know optimal value of k. There are $j-i$ possibilities, $k = i, i+1, \ldots, j-1$, hence

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{m[i,k] + m[k+1,j] & \text{if } i < j \\ +p_{i-1} \cdot p_k \cdot p_j\} & \end{cases}$$

We also keep track of optimal splits:

$$s[i,j] = k \ \Leftrightarrow m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j$$

($s[i,j]$ is a value of k at which we split the product $A_{i,j}$ to obtain an optimal parenthesization)

This can be used to write a recursive algorithm:

Recursive-Matrix-Chain(p, i, j)

if $i = j$ **then**
    return 0
end if
$m[i, j] \leftarrow \infty$
for $k \leftarrow i$ **to** $j - 1$ **do**
    $q \leftarrow$ RECURSIVE-MATRIX-CHAIN$(p, i, k) +$
    RECURSIVE-MATRIX-CHAIN$(p, k + 1, j) + p_{i-1} p_k p_j$
    if $q < m[i, j]$ **then**
        $m[i, j] \leftarrow q$
    end if
end for
return $m[i, j]$

Running time analysis:

$T(1) \geq 1$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \text{ for } n > 1$$

rewrite:

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n$$

This is still exponential in n:

we prove that $T(n) \geq 2^{n-1}$ by induction on n

$$\begin{aligned} T(n) &\geq 2\sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2\sum_{i=0}^{n-2} 2^{i} + n \\ &= 2(2^{n-1} - 1) + n \\ &= 2^n + n - 2 \\ &\geq 2^{n-1} \end{aligned}$$

Hence, $T(n) = \Omega(2^n)$.

3. Computing the optimal costs

Want to compute $m[1,n]$, minimum cost for multiplying $A_1 \cdot A_2 \cdots A_n$.

Recursively, it would take $\Omega(2^n)$ steps: the same subproblems are computed over and over again.

However, if we compute in a bottom-up fashion, we can reduce running time to polynomial in n.

The recursive equation shows that cost $m[i,j]$ (product of $j-i+1$ matrices) depends only on smaller subproblems:
for $k = 1, \ldots, j-1$,

- $A_{i,k}$ is a product of $k-i+1 < j-i+1$ matrices,
- $A_{k+1,j}$ is a product of $j-k < j-i+1$ matrices.

Algorithm should fill table m in order of increasing lengths of chains.

Matrix-Chain-Order(p)

$n \leftarrow \text{length}[p] - 1$
for $i \leftarrow 1$ **to** n **do**
    $m[i,i] \leftarrow 0$
end for
for $\ell \leftarrow 2$ **to** n **do**
    for $i \leftarrow 1$ **to** $n - \ell + 1$ **do**
        $j \leftarrow i + \ell - 1$
        $m[i,j] \leftarrow \infty$
        for $k \leftarrow i$ **to** $j - 1$ **do**
            $q \leftarrow m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j$
            if $q < m[i,j]$ **then**
                $m[i,j] \leftarrow q$
                $s[i,j] \leftarrow k$
            end if
        end for
    end for
end for
return m and s

Example.

Six matrices:

$$
\begin{array}{ll}
A_1\ (30 \times 35) & A_2\ (35 \times 15) \\
A_3\ (15 \times 5) & A_4\ (5 \times 10) \\
A_5\ (10 \times 20) & A_6\ (20 \times 25)
\end{array}
$$

Recall: multiplying A $(p \times q)$ and B $(q \times r)$ takes $p \cdot q \cdot r$ scalar multiplications.

j \ i	1	2	3	4	5	6
6						0
5					0	
4				0		
3			0			
2		0				
1	0					

j \ i	1	2	3	4	5	6
6	15,125	10,500	5,375	3,500	5,000	0
5	11,875	7,125	2,500	1,000	0	
4	9,375	4,375	750	0		
3	7,875	2,625	0			
2	15,750	0				
1	0					

4. Constructing an optimal solution

Simple with array $s[i, j]$: $s[]$ shows us an optimal split point for every subproblem.
Here is a recursive procedure to print an optimal parenthesization in linear time:

Print-Optimal-Parenthesization(s, i, j)

```
if i = j then
    print "A_i"
else
    print "("
    PRINT-OPTIMAL-PARENTHESIZATION(s, i, s[i, j])
    PRINT-OPTIMAL-PARENTHESIZATION(s, s[i, j] + 1, j)
    print ")"
end if
```

Assignment Problem 11.2. (deadline: July 29, 5:30pm)
Consider a variant of the matrix-chain multiplication problem in which the goal is to parenthesize the sequence of matrices so as to maximize, rather than minimize, the number of scalar multiplications. Perform all 4 steps to design a Dynamic Programming algorithm.

Time complexity

We have three nested loops:

1. ℓ, length, $O(n)$ iterations
2. i, start, $O(n)$ iterations
3. k, split point, $O(n)$ iterations

Body of loops: constant complexity.

Total complexity: $O(n^3)$
(compared to $\Omega(2^n)$ for brute-force approach).

In many cases, Dynamic programming approaches are more efficient than simple Divide&Conquer.

DP: Longest common subsequence

— biologists often need to find out how similar are 2 DNA sequences

— DNA sequences are strings of *bases*: A, C, T and G

— how to define similarity?

- one is a substring of another
- number of changes (mutations) needed to change one string to another
- *the longest common subsequence* of two strings S_1 and S_2: a longest sequence S_3 appearing in each of S_1 and S_2 (in the same order, but necessarily consecutively)

Definition. $Z = z_1 z_2 \dots z_k$ is a **subsequence** of $S = s_1 s_2 \dots s_n$ if there exists an increasing sequence of indexes: $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ such that $z_j = s_{i_j}$

Example.

$$\begin{array}{ccccccccccc} \text{S=} & G & G & C & A & C & T & G & T & A & C \\ & & \downarrow & \downarrow & & \downarrow & & & & \downarrow & \\ \text{Z=} & & G & C & & C & & & & A & \end{array}$$

$Z = GCCA$ is a subsequence of $S = GGCACTGTAC$

Definition. Z is **a common subsequence** of X and Y if its a subsequence of both X and Y.

A longest such Z is called **a longest common subsequence** — LCS.

Example. Consider

$$X = GGCACTGTAC$$

$$Y = CATGTCACGG$$

Then $ATAC$ and $GCAG$ are a common subsequences of X and Y. The longest common subsequence is $CATGTAC$.

"brute-force approach": list all subsequences of X and for each test if it's subsequence of Y
if X has a length m, there are 2^m subsequences of X
exponential time

"dynamic programming approach":

1. Characterizing structure

consider a string $S = s_1 s_2 \dots s_n$, then for every $1 \leq i \leq j \leq n$, we define a **substring** $S_{i,j}$ of S as follows

$$S_{i,j} = s_i s_{i+1} \dots s_{j-1} s_j$$

space of subproblems:
— inspired by "matrix-chain multiplication problem" we could consider the following subproblems: longest common subsequences of substrings $X_{i,j}$ and $Y_{k,l}$ for $i \leq j$ and $k \leq l$
— "thumb rule": keep the space of subproblems as small as possible

— class of subproblems: LCS's of prefixes $X_{1,i}$ and $Y_{1,j}$

Assignment Problem 11.3. (deadline: July 29, 5:30pm)
Give an $\mathcal{O}(n+m)$ time algorithm deciding whether a sequence $X = x_1 \ldots x_n$ is a subsequence of $Y = y_1 \ldots y_m$. Remember to explain how you algorithm works!

Note: A DP algorithm for this problem would work in time $O(n.m)$. You will only get a half of the points for such a solution.

optimal substructure of LCS

Claim. Let $Z = z_1 \dots z_k$ be a LCS of $X = x_1 \dots x_m$ and $Y = y_1 \dots y_n$. Then

1. if $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{1,k-1}$ is an LCS of $X_{1,m-1}$ and Y_{n-1};
2. if $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of $X_{1,m-1}$ and Y;
3. if $x_m \neq y_n$ and $z_k \neq y_n$, then Z is an LCS of X and $Y_{1,n-1}$.

Proof.

1. - if $z_k \neq x_m = y_n$, then Zx_m is a common subsequence of X and Y longer than Z, a contradiction
   - clearly, $Z_{1,k-1}$ is a common subsequence of $X_{1,m-1}$ and $Y_{1,n-1}$
   - if not a longest one: let W be an LCS of $X_{1,m-1}$ and $Y_{1,n-1}$; then $W z_k$ is a common subsequence of X and Y, again a contradiction ("cut-and-paste")

2. clearly, since $z_k \neq x_m$, Z is a common subsequence of $X_{1,m-1}$ and Y;
   if not a longest one: use "cut-and-paste" technique again

3. similarly as in case 2.

Hence, an LCS of two sequences contains within it an LCS of prefixes of these two sequences: **optimal substructure** property.

Example. $CATGTAC$ is an LCS of
$X = GGCACTGTAC$ and $Y = CATGTCACGG$

by 3., $CATGTAC$ is an LCS of
$X = GGCACTGTAC$ and $Y_{1,9} = CATGTCACG$

by 3., $CATGTAC$ is an LCS of
$X = GGCACTGTAC$ and $Y_{1,8} = CATGTCAC$

by 1., $CATGTA$ is an LCS of
$X_{1,9} = GGCACTGTA$ and $Y_{1,7} = CATGTCA$

etc.

2. A recursive solution

by above, to find an LCS of $X = x_1 \ldots x_m$ and $Y = y_1 \ldots y_n$:

- if $x_m = y_n$, then find an LCS of $X_{1,m-1}$ and $Y_{1,n-1}$ and append $x_m = y_n$ to it
- if $x_m \neq y_n$, then find an LCS of X and $Y_{1,n-1}$ and an LCS of $X_{1,m-1}$ and Y, and take the longer of these two

let $c[i, j]$ be the length of an LCS of $X_{1,i}$ and $Y_{1,j}$

recursive formula:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

3. Computing

a recursive algorithm based on recursive formula would be again exponential, however there are only $(m+1)(n+1)$ subproblems ("*overlapping-subproblems property*")

entries of table $c[0 \dots m, 0 \dots n]$ are filled in "**row-major order**":
the first row from left to right
the second row from left to right
etc

table $b[1 \dots m, 1 \dots n]$ - contains the information to construct the optimal solution (shows a *direction* from where we got the minimal value of the length of an LCS:
"$\leftarrow$" — $c[i,j] = c[i,j-1]$,
"$\uparrow$" — $c[i,j] = c[i-1,j]$, or
"↰" — $c[i,j] = c[i-1,j-1] + 1$.

LCS-Length(X, Y)

$m \leftarrow$ length$[X]$
$n \leftarrow$ length$[Y]$
for $i \leftarrow 1$ **to** m **do**
    $c[i, 0] \leftarrow 0$
end for
for $i \leftarrow 1$ **to** n **do**
    $c[0, i] \leftarrow 0$
end for
for $i \leftarrow 1$ **to** m **do**
    for $j \leftarrow 1$ **to** n **do**
        if $x_i = y_j$ **then**
            $c[i, j] \leftarrow c[i-1, j-1] + 1$
            $b[i, j] \leftarrow$ "$\nwarrow$"
        else
            if $c[i-1, j] \geq c[i, j-1]$ **then**
                $c[i, j] \leftarrow c[i-1, j]$
                $b[i, j] \leftarrow$ "$\uparrow$"
            else
                $c[i, j] \leftarrow c[i-1, j]$
                $b[i, j] \leftarrow$ "$\leftarrow$"
            end if
        end if
    end for
end for
return c and b

Time complexity: $\mathcal{O}(mn)$

Example.

	j	1	2	3	4	5	6	7	8	9	10
i		G	G	C	A	C	T	G	T	A	C
1	C	↑,0	↑,0	↰,1	←,1	↰,1	←,1	←,1	←,1	←,1	↰,1
2	A	↑,0	↑,0	↑,1	↰,2	←,2	←,2	←,2	←,2	↰,2	←,2
3	T	↑,0	↑,0	↑,1	↑,2	↑,2	↰,3	←,3	↰,3	←,3	←,3
4	G	↰,1	↰,1	↑,1	↑,2	↑,2	↑,3	↰,4	←,4	←,4	←,4
5	T	↑,1	↑,1	↑,1	↑,2	↑,2	↰,3	↑,4	↰,5	←,5	←,5
6	C	↑,1	↑,1	↰,2	↑,2	↰,3	↑,3	↑,4	↑,5	↑,5	↰,6
7	A	↑,1	↑,1	↑,2	↰,3	↑,3	↑,3	↑,4	↑,5	↰,6	↑,6
8	C	↑,1	↑,1	↰,2	↑,3	↰,4	←,4	↑,4	↑,5	↑,6	↰,7
9	G	↰,1	↰,2	↑,2	↑,3	↑,4	↑,4	↰,5	↑,5	↑,6	↑,7
10	G	↰,1	↰,2	↑,2	↑,3	↑,4	↑,4	↰,5	↑,5	↑,6	↑,7

4. Constructing an LCS

recursive procedure:

Print-LCS(b, X, i, j)

if $i = 0$ or $j = 0$ **then**
    return
end if
if $b[i, j] =$ "↰" **then**
    Print-LCS$(b, X, i - 1, j - 1)$
    print x_i
else
    if $b[i, j] =$ "↑" **then**
        Print-LCS$(b, X, i - 1, j)$
    else
        Print-LCS$(b, X, i, j - 1)$
    end if
end if

Time complexity: $\mathcal{O}(m + n)$ — in each step at least one of i and j is decreased by 1

Assignment Problem 11.4. (deadline: July 29, 5:30pm)
Give an $\mathcal{O}(n^2)$ time algorithm to find the longest monotonically increasing subsequence of a sequence of n distinct numbers.

All-pairs shortest paths

- Directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$, $|V| = n$
- Assume G contains no negative-weight cycles
- **Goal:** create $n \times n$ matrix of shortest path distances $\delta(u, v)$, $u, v \in V$
- Adjacency-matrix representation of graph:
  - $n \times n$ adjacency matrix $W = (w_{ij})$ of edge weights
  - assume

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

- Weight of path $p = (v_1, v_2, \ldots, v_k)$ is $w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$

Shortest paths & matrix multiplication

In the following, we only want to compute lengths of shortest paths, not construct the paths (see the textbook if you are interested in constructing the paths).

Dynamic programming approach, first 3 steps steps:

1. Structure of a shortest path

Subpaths of shortest paths are shortest paths

Lemma. Let $p = (v_1, v_2, \ldots, v_k)$ be a shortest path from v_1 to v_k, let $p_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ for $1 \le i \le j \le k$ be subpath from v_i to v_j. Then, p_{ij} is shortest path from v_i to v_j.

Proof. Decompose p into

$$v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k.$$

Then, $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Assume there is cheaper p'_{ij}

from v_i to v_j with $w(p'_{ij}) < w(p_{ij})$. Then

$$v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

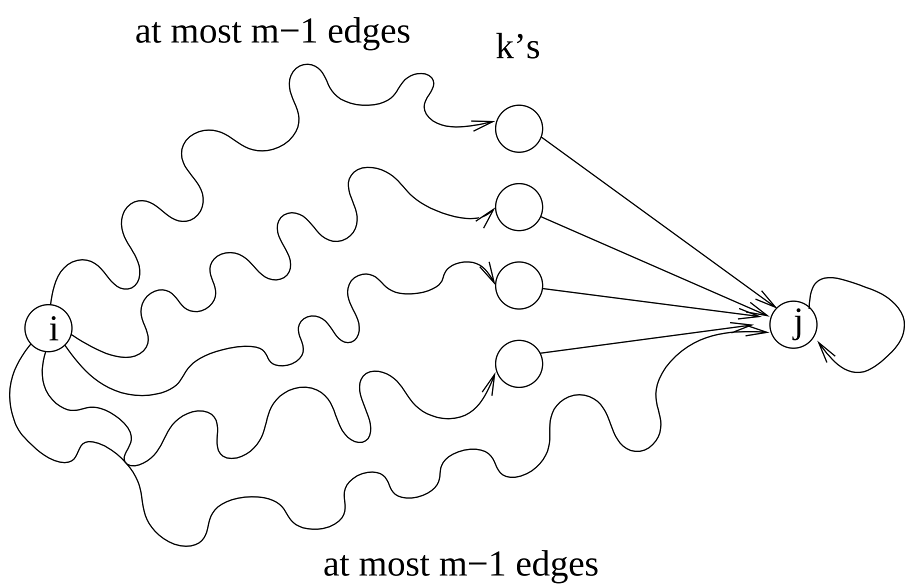
is path from v_1 to v_k whose weight $w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, a contradiction.

2. Recursive solution &
3. Compute opt. value (bottom-up)

Let $d_{ij}^{(m)}$ = weight of shortest path from i to j that uses at most m edges.

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

$$d_{ij}^{(m)} = \min_k \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}$$



Note: the shortest path from i to j can use at most $n-1$ edges (n is the number of vertices).

Hence, we're looking for

$$\delta(i,j) = d_{ij}^{(n-1)}$$

The algorithm is straightforward, running time is $O(n^4)$ ($n-1$ passes, each computing n^2 d's in $\Theta(n)$ time)

Similar to **matrix multiplication**

$C = A \cdot B$, $n \times n$ matrices, $c_{ij} = \sum_k a_{ik} \cdot b_{kj}$
$O(n^3)$ operations

Replacing "$+$" with "$\min$" and "$\cdot$" with "$+$" gives

$$c_{ij} = \min_k \{a_{ik} + b_{kj}\},$$

very similar to

$$d_{ij}^{(m)} = \min_k \{d_{ik}^{(m-1)} + w_{kj}\}$$

Hence $D^{(m)} = D^{(m-1)} \otimes W$

Note: Identity matrix for this “multiplication” $\otimes$ is

$$\bar{I} = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix} = D^{(0)} = (d_{ij})^{(0)}$$

Why? Replace 0 (identity for $+$) in real identity matrix with ∞ (identity for $\min$), and replace 1 (identity for $\cdot$) in real identity matrix with 0 (identity for $+$).

Assignment Problem 11.5. (deadline: July 29, 5:30pm)
Show that the above "multiplication" (with $\min$ instead of sum and addition instead of multiplication) of matrices is associative, i.e., that for any three matrices A, B and C, we have

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C.$$

Hence: this "multiplication" $\otimes$ is **associative**. Algebraic structure is **closed semiring** (no ring because min has no inverse).

So, we can use

$$\begin{aligned}
D^{(0)} &= \bar{I} \\
D^{(1)} &= D^{(0)} \otimes W = W \\
D^{(2)} &= D^{(1)} \otimes W = W^2 \\
D^{(3)} &= D^{(2)} \otimes W = W^3 \\
&\vdots \\
D^{(n-1)} &= D^{(n-2)} \otimes W = W^{n-1}
\end{aligned}$$

$D^{(n-1)} = (\delta(i,j))$, so that's the answer

Time: $\Theta(n \cdot n^3) = \Theta(n^4)$
$\Theta(n)$ "multiplications", each $\Theta(n^3)$

Unfortunately, no better than before...

Assignment Problem 11.6. (deadline: July 29, 5:30pm)
We are assuming that the graph doesn't contain a negative-weight cycle. What happens if we drop this assumption?

1. Show that if a graph contains a negative-weight cycle then there are two vertices with a shortest path distance $-\infty$.

2. Use matrices $D^{(1)}, \ldots, D^{(n-1)}$ to identify that the graph contains a negative-weight cycle, as well, to find the length (the number of edges) of a smallest such cycle.

But, with **repeated squaring**:

$$W^{2n} = W^n \times W^n$$

Compute

$$\underbrace{W, W^2, W^4, W^8, \ldots, W^{2^{\lceil \log(n-1) \rceil}}}_{\Theta(n) \text{ squarings}}$$

Note: $2^{\lceil \log(n-1) \rceil} \geq n-1$

OK to overshoot since product doesn't change after converging to $(\delta(i,j))$

Time: $\Theta(n^3 \log n)$
$\Theta(\log n)$ squarings, each $\Theta(n^3)$

There's something even better…

Floyd-Warshall algorithm

Also Dynamic programming, but faster (factor $\log n$)

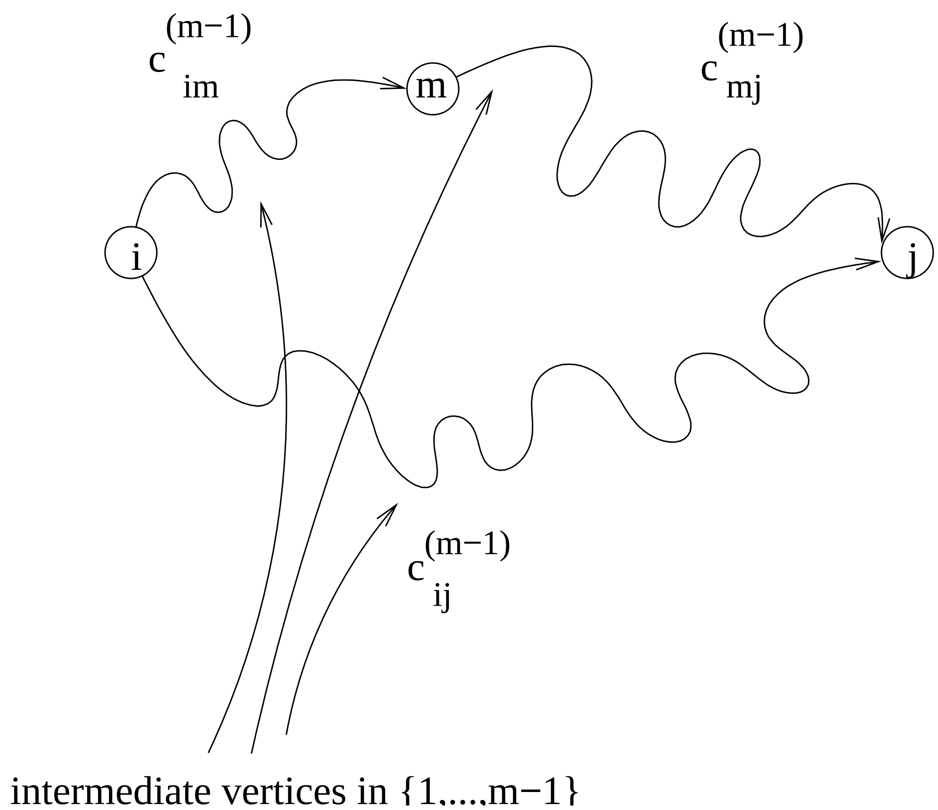
Assume $V = \{1, 2, \ldots, n\}$.

Define $c_{ij}^{(m)}$ = weight of a shortest path from i to j with **intermediate vertices** in $\{1, 2, \ldots, m\}$.

Then $\delta(i, j) = c_{ij}^{(n)}$

Compute $c_{ij}^{(n)}$ in terms of smaller ones, $c_{ij}^{(<n)}$:

$$
\begin{aligned}
c_{ij}^{(0)} &= w_{ij} \\
c_{ij}^{(m)} &= \min\left(c_{ij}^{(m-1)},\ c_{im}^{(m-1)} + c_{mj}^{(m-1)}\right)
\end{aligned}
$$



$c_{im}^{(m-1)}$
$c_{mj}^{(m-1)}$
m
i
j
$c_{ij}^{(m-1)}$
intermediate vertices in {1,....,m−1}

Difference from previous algorithm: we don't have to check *all* possible intermediate vertices. Shortest path simply either includes m or doesn't.

Pseudocode:

$C^{(0)} \leftarrow W$
for $m \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $c_{ij}^{(m)} \leftarrow \min(c_{ij}^{(m-1)}, c_{im}^{(m-1)} + c_{mj}^{(m-1)})$
    end for
  end for
end for
return $C^{(n)}$

Superscripts can be dropped: improving the space requirement to $\Theta(n^2)$.

Time: $\Theta(n^3)$, simple code

Best algorithm to date is $O(n^2 \log n + n|E|)$

4. Constructing a shortest paths

we need to compute a **predecessor matrix**:
$\Pi = (\pi_{ij})$ where

- $\pi_{ij} =$ NIL, if $i = j$ or there is no path from i to j
- $\pi_{ij} =$ predecessor of j on a shortest path from i to j, otherwise

once we have a predecessor matrix, the algorithm is easy:

Print-Shortest-Path(Π, i, j)

if $i = j$ **then**
    print i
else
    if $\pi_{ij} =$ NIL **then**
        print "no path"
    else
        Print-Shortest-Path(Π, i, π_{ij})
        print j
    end if
end if

How to compute Π?

we can compute the sequence of $\Pi^{(0)}, \ldots, \Pi^{(n)}$ at the same time as we are computing the sequence $D^{(0)}, \ldots, D^{(n-1)}$ (the first algorithm), or the sequence $C^{(0)}, \ldots, C^{(n)}$ (Floyd-Warshall algorithm), respectivelly

– in the case of first algorithm, we cannot use *squaring technique*

– let's concentrate only on **Floyd-Warshall algorithm**:

Let $\pi_{ij}^{(m)}$ be the predecessor of j on a shortest path from i to j with all intermedaite vertices in the set $\{1, 2, \ldots, m\}$

initialization:

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

recursive step: depends where the minimal weight of a shortest path comes from

$$\pi_{ij}^{(m)} = \begin{cases} \pi_{ij}^{(m-1)} & \text{if } c_{ij}^{(m-1)} \leq c_{im}^{(m-1)} + c_{mj}^{(m-1)} \\ \pi_{mj}^{(m-1)} & \text{if } c_{ij}^{(m-1)} > c_{im}^{(m-1)} + c_{mj}^{(m-1)} \end{cases}$$

the algorithm:

$C^{(0)} \leftarrow W$
for $m \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      if $c_{ij}^{(m-1)} \leq c_{im}^{(m-1)} + c_{mj}^{(m-1)}$ **then**
        $c_{ij}^{(m)} \leftarrow c_{ij}^{(m-1)}$
        $\pi_{ij}^{(m)} \leftarrow \pi_{ij}^{(m-1)}$
      else
        $c_{ij}^{(m)} \leftarrow c_{im}^{(m-1)} + c_{mj}^{(m-1)}$
        $\pi_{ij}^{(m)} \leftarrow \pi_{mj}^{(m-1)}$
      end if
    end for
  end for
end for
return $C^{(n)}$ and $\Pi^{(n)}$

Application: Transitive closure of a directed graph

given: a directed graph $G = (V, E)$

ouput: the **transitive closure** = a directed graph $G^* = (V, E^*)$, where

$$(i, j) \in E^* \quad \text{if} \quad \text{there is a path from } i \text{ to } j \text{ in } G$$

Easy to compute using the above algorithm:

- assign weight 0 to all edges in E (if $(i, j) \notin E$, then $w_{ij} = \infty$)
- run the *Floyd-Warshall* algorithm $\longrightarrow C$
- (i, j) is an edge in E^* if and only if $c_{ij} = 0$

Note: only value in matrices $C^{(m)}$ are 0 and ∞, which can be interpreted as boolean values and operations "$+$" and "$\min$" can be replaced by logical operations AND and OR