Algorithms & Data Structures I
CSC 225

Ali Mashreghi

Fall 2018

Department of Computer Science, University of Victoria

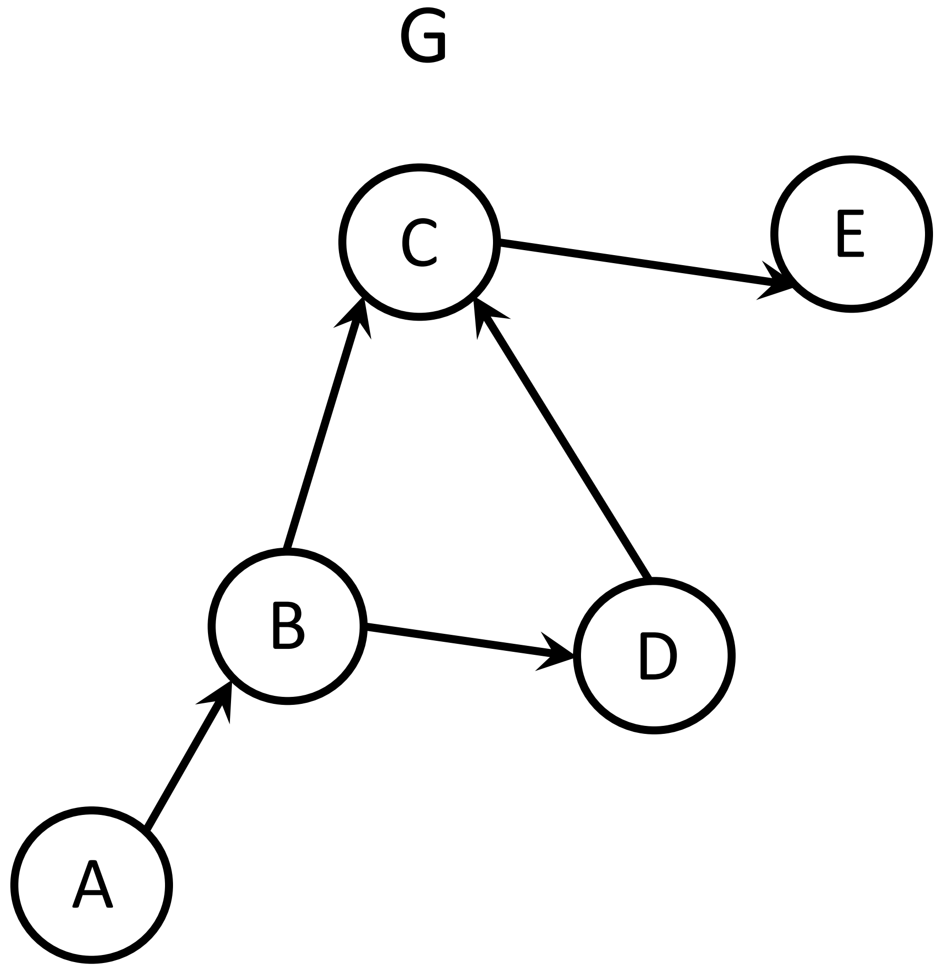
Transitive closure

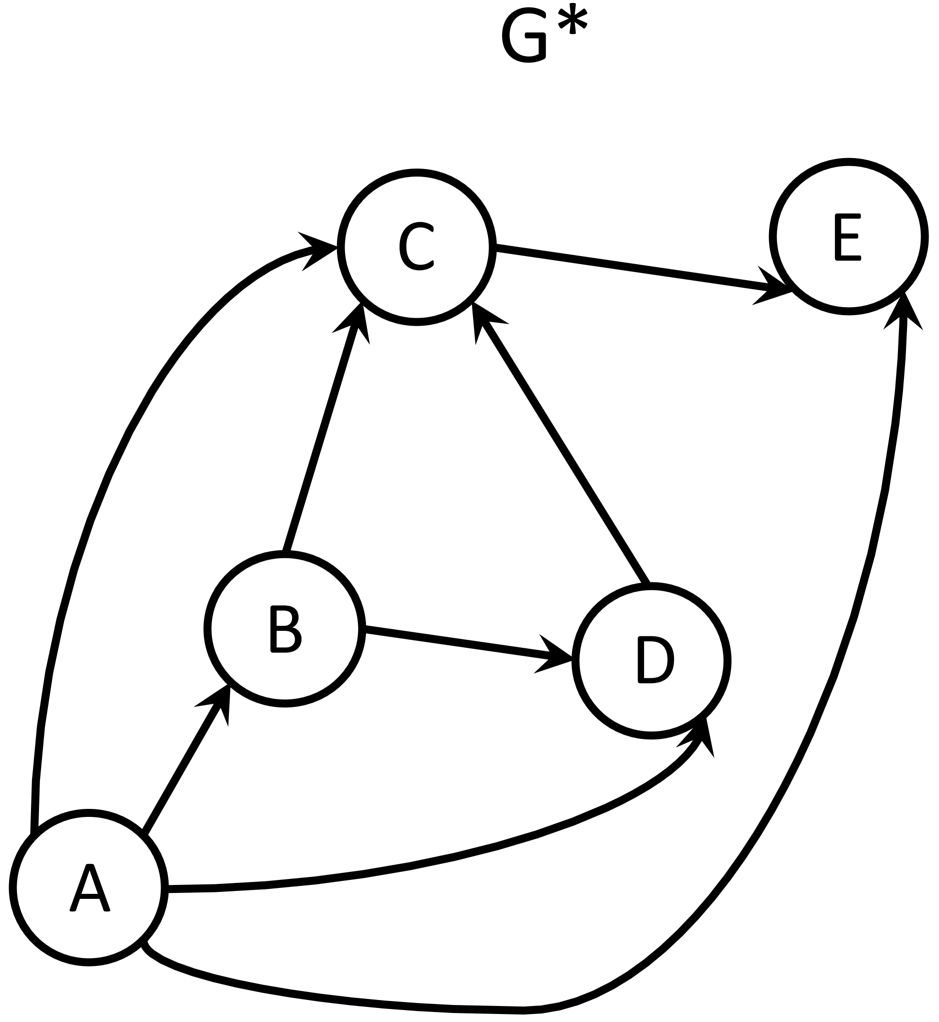
- Transitive closure of a graph $G = (V, E)$, is another graph $G^* = (V, E^*)$ such that:

1. The set of vertices of G and G^* are the same.
2. In G^* there is an **edge** from u to v, if and only if there is a **path** from u to v in G.

✓Having G^*, we can answer to all queries of the form, **“is $\boldsymbol{x}$ reachable from $\boldsymbol{y}$ in $\boldsymbol{G}$?”**

Transitive closure





Transitive closure

- In fact, most of the time we don't just want to know which nodes are reachable from one another.
- We also want to know what is the length of the shortest path between all pairs of nodes.
- This is known as the **all-pairs shortest path problem** (APSP).

Transitive closure

- We present four solutions to this problem:

1. Using BFS
2. Using strongly connected components (SCCs)
3. Using matrix multiplication
4. Using dynamic programming

Transitive closure

- **Question:** How can we use BFS to find to solve the all-pairs shortest path problem?

Transitive closure

- **Question:** How can we use BFS to find to solve the all-pairs shortest path problem?
- **Answer:**

Allocate a 2D integer array A to store the the value of the shortest paths between each pairs of nodes. (initialized with ∞)

Pick each node as the source, and run a BFS from that node. Each time you find $\delta(u, v)$, update the value of $A[u][v]$.

Transitive closure

- **Question:** What is the time complexity of the algorithm?

Transitive closure

- **Question:** What is the time complexity of the algorithm?
- **Answer:** $n \cdot O(m+n) = O(mn+n^2)$, since $m = O(n^2)$, the solution could be $O(n^3)$.

Transitive closure

- **Question:** What is the time complexity of the algorithm?
- **Answer:** $n \cdot O(m+n) = O(mn+n^2)$, since $m = O(n^2)$, the solution could be $O(n^3)$.

Transitive closure

- We present four solutions to this problem:

1. Using BFS ✔:

   (1) works for directed and undirected graphs

   (2) not only it determines reachability but it also finds the shortest paths.

1. Using strongly connected components (SCCs)
2. Using matrix multiplication
3. Using dynamic programming

Using SCCs

1. We run the **strongly connected components** algorithm on a graph G.
2. Then, if we have $\boldsymbol{k}$ SCCs, we obtain a component **graph $\boldsymbol{G'}$ with $\boldsymbol{k}$ nodes**.
3. We run the **BFS** algorithm from each of **these $\boldsymbol{k}$ nodes** to see which nodes in G' can reach each other.

- To see whether a node u can reach node v in G, we check whether u's component can reach v's component.

Using SCCs

- Step 1 takes $O(n + m)$.

- For step 2, for each component we can take an arbitrary node in that component and run a BFS to see which other components are reachable. This takes $O\left(k(n + m)\right)$.

- Step 3, needs k BFSs on a graph with k nodes and $O(k^2)$ edges, which will be $O(k^3)$ overall.

Using SCCs

✓ So, the total complexity is $O(k(n+m)+k^3)$.

- This approach is **very efficient** if we know that **$\boldsymbol{k}$ is much smaller than $\boldsymbol{n}$**.

- In fact, if $k=o(n)$ this approach will take $o(n^3)$.

Transitive closure

- We present four solutions to this problem:

1. Using BFS ✔:
2. Using strongly connected components (SCCs) ✔:

   (1) Very efficient if we know that the number of SCCs is much smaller than the number of nodes.

   (2) If k is close to n, it will still be as bad as n^3.

1. Using matrix multiplication
2. Using dynamic programming

Using matrix multiplication

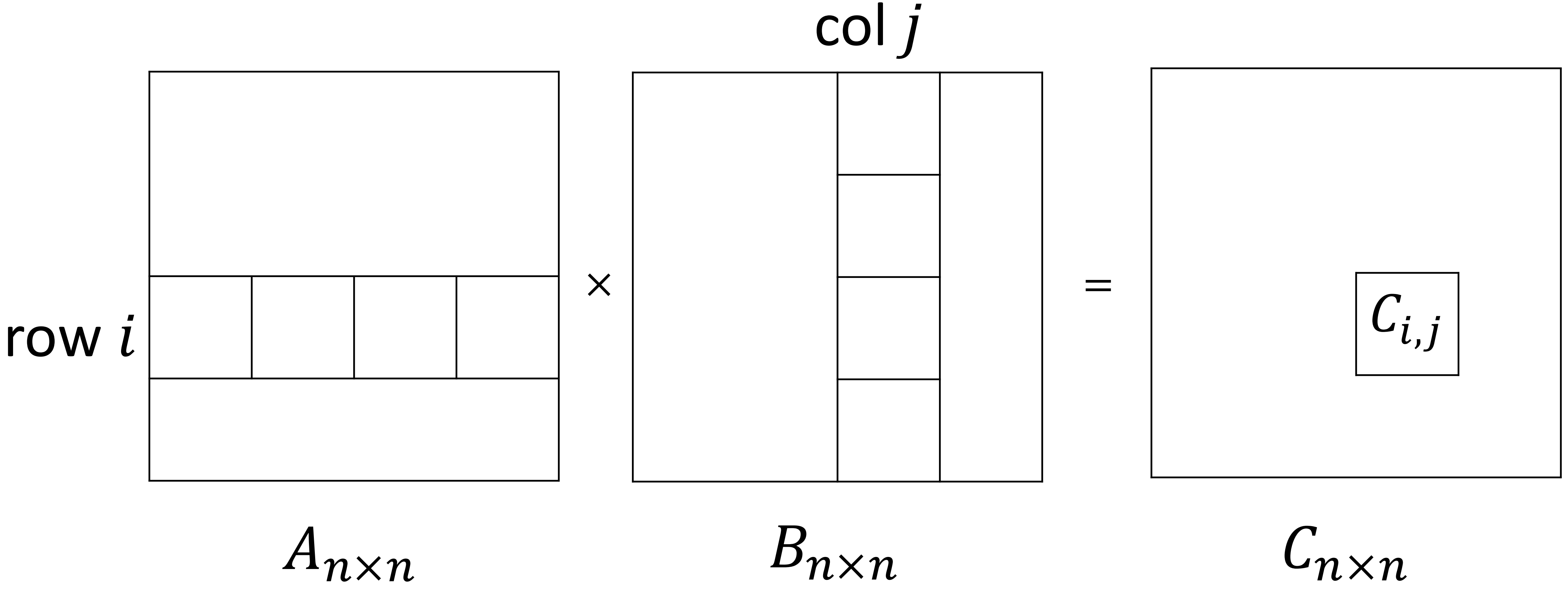
- In this approach we intend to use a matrix operation that **resembles matrix multiplication** to solve the reachability problem.

- In this method, not only we determine the reachability of nodes, but **we obtain the shortest paths** for all pairs too.

- This method can compute the shortest paths **even if the graph is weighted**.

Matrix multiplication

- We multiply two $n \times n$ matrices by multiplying the row i of A by col j of B to form the entry $C_{i,j}$ of the result.



Matrix multiplication

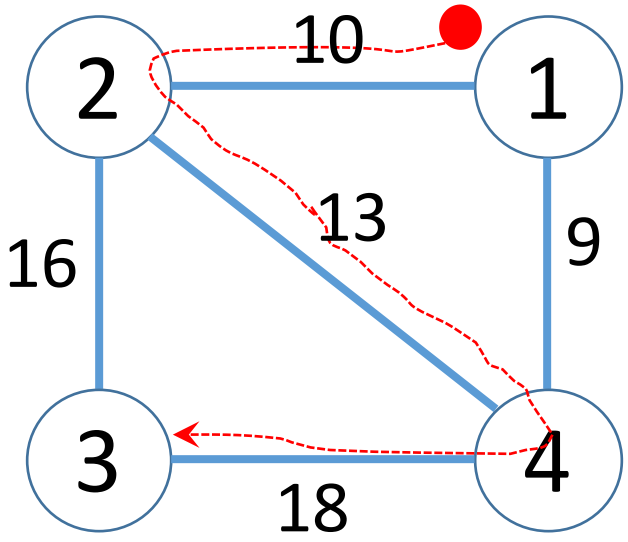
- The algorithm will have time complexity of $\Theta(n^3)$

MATRIXMULTIPLICATION(A, B) //two $n \times n$ matrices (as 2D arrays)

1. Let C be a new $n \times n$ matrix initialized with 0's
2. **for** $i = 0$ **to** n-1
3.   **for** $j = 0$ **to** n-1
4.   **for** $k = 0$ **to** n-1
5.   C[i][j] = C[i][j] + (A[i][k]*B[k][j])
6. **return** C

Weighted shortest path

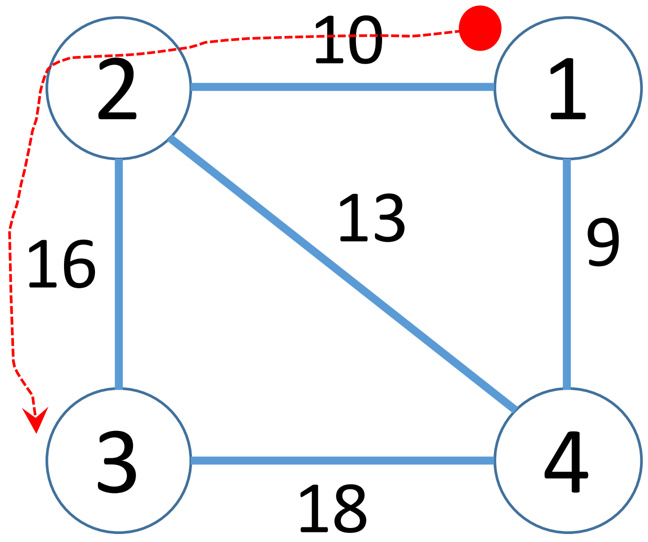
- If a graph is weighted we say that the weight of a path is sum of the weights on the edges of the path.



A path of length 10+13+18=41

Weighted shortest path

- The shortest path between two nodes u and v, is a path whose weight is the smallest among all $u - v$ paths.

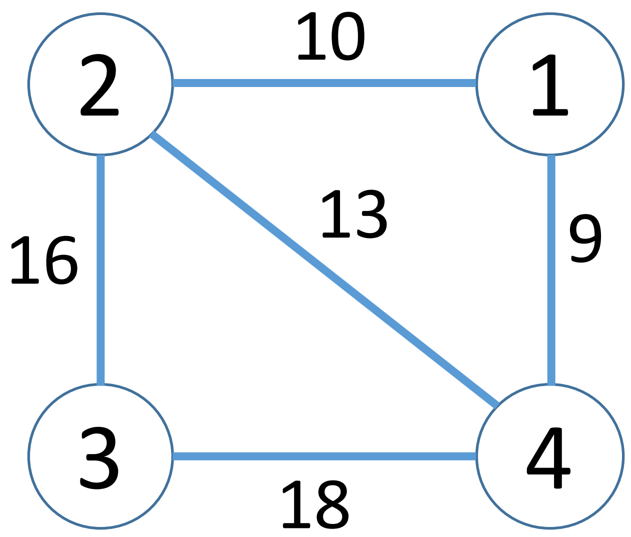


The shortest path from node 1 to node 3, has a weight of 26

Weighted shortest path

- In a weighted graph we deal with a weight matrix W.

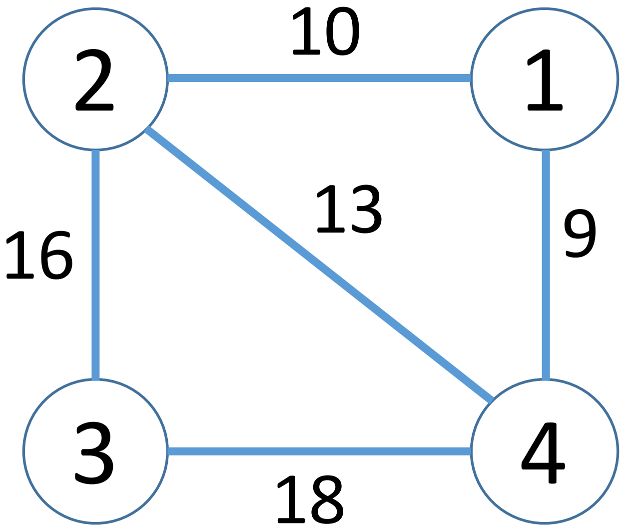
	1	2	3	4
1	0	10	∞	9
2	10	0	16	13
3	∞	16	0	18
4	9	13	18	0



Weighted shortest path

- $W[i][j]$ shows the weight of the edge (i, j), or ∞ if the edge does not exist. $W[i][i]$ is 0.

	1	2	3	4
1	0	10	∞	9
2	10	0	16	13
3	∞	16	0	18
4	9	13	18	0



Weighted shortest path

- Note that the shortest path in a weighted graph **does not necessarily** have the least number of edges.

- We want to see have we can **compute these shortest paths** by **performing an operation** similar to matrix multiplication **on the weight matrix**.

Weighted shortest path

- Let $\boldsymbol{L_i}$ be a matrix that contains the value of **all shortest paths** in the graph that use **at most $\boldsymbol{i}$ edges**.

- With this definition we have $\boldsymbol{L_1 = W}$, since W has the value of all shortest paths that use **at most 1 edge**.

- We want to form matrices $L_1, L_2, \ldots, L_{n-1}$, by computing L_{i+1} from L_i.

Weighted shortest path

- Once we compute L_{n-1}, since no shortest path uses more than $n-1$ edges, L_{n-1} will have the value of all possible shortest paths in the graph.

- To obtain the **transitive closure** and answer to **reachability queries** for two nodes u and v:

1. $L_{n-1}[u][v] = \infty$ means u can't reach v
2. $L_{n-1}[u][v] \neq \infty$ is the length of the shortest path

Weighted shortest path

- The following procedure can extend L_i to L_{i+1}:

//L is an $n \times n$ matrix that has the current shortest paths
//W is the weight matrix of the graph
EXTEND-SHORTEST-PATHS(L, W)
1. Let R be a new $n \times n$ matrix, initialized with ∞
2. **for** i = 0 **to** n-1
3. **for** j = 0 **to** n-1
4. **for** k = 0 **to** n-1
5. R[i][j] = min(R[i][j], L[i][k] + W[k][j])
6. **return** R

Weighted shortest path

- The following procedure can extend L_i to L_{i+1}:

```
//L is an n×n matrix that has the current shortest paths
//W is the weight matrix of the graph
EXTEND-SHORTEST-PATHS(L, W)
Let R be a new n×n matrix, initialized with ∞
for i = 0 to n-1
    for j = 0 to n-1
        for k = 0 to n-1
            R[i][j] = min(R[i][j], L[i][k] + W[k][j])
return R
```

k is an intermediate vertex we use to find new paths of with one more edge

Weighted shortest path

- The complexity of this method is $\Theta(n^3)$.

//L is an $n \times n$ matrix that has the current shortest paths
//W is the weight matrix of the graph
EXTEND-SHORTEST-PATHS(L, W)
1. Let R be a new $n \times n$ matrix, initialized with ∞
2. **for** i = 0 **to** n-1
3.   **for** j = 0 **to** n-1
4.   **for** k = 0 **to** n-1
5.   R[i][j] = min(R[i][j], L[i][k] + W[k][j])
6. **return** R

Weighted shortest path

- Now, we can obtain the shortest path between all pairs of nodes as follows:

ALL-PAIRS-SHORTEST-PATHS(W) // an $n \times n$ weight matrix

1. Let $L_1 = W$
2. **for** i = 2 **to** n-1
3. $\quad L_{i+1}$ = EXTEND-SHORTEST-PATHS(L_i, W)
4. **return** L_{n-1}

Weighted shortest path

- The time complexity of the algorithm is $\Theta(n^4)$, since we have to compute $L_1, L_2, \ldots, L_{n-1}$.
- This algorithm works for both directed and undirected graphs.

ALL-PAIRS-SHORTEST-PATHS(W) // an $n \times n$ weight matrix

1. Let $L_1 = W$
2. **for** $i = 2$ **to** n-1
3. $\quad L_{i+1} =$ EXTEND-SHORTEST-PATHS(L_i, W)
4. **return** L_{n-1}

Improving time

- We make the following two observations:

1. All we care about is $\boldsymbol{L_{n-1}}$**!**
2. Moreover, we are applying **the same operation** which is extending using matrix W $\boldsymbol{n}$ **times**.

Improving time

- We make the following two observations:

1. All we care about is $\boldsymbol{L_{n-1}}$**!**
2. Moreover, we are applying **the same operation** which is extending using matrix W $\boldsymbol{n}$ **times**.

- In fact, in the context of matrix multiplication what we are doing is similar to computing:

$$A^x = \underbrace{A \times A \times \cdots \times A}_{\text{x times}}$$

Improving time

- Let's see how we can compute A^x **with less than x multiplications!**

- Then, we apply the same idea to our all-pairs shortest paths algorithm.

Improving time

- Matrix multiplication has the **associative property**:

$$(A \times B) \times C = A \times (B \times C)$$

- In other words, the order of multiplication **doesn't matter**.

- As a result:

$$A^x = \underbrace{A \times A \times \cdots \times A}_{\text{x times}} = \underbrace{(A \times \cdots \times A)}_{\text{x/2 times}} \times \underbrace{(A \times \cdots \times A)}_{\text{x/2 times}} = A^{x/2} \times A^{x/2}$$

Improving time

- So, if we raising a matrix to a power we can do as follows:

MATRIX-POWER(A, x)
1. **if** $x = 1$
2. **return** A
3. B = MATRIX-POWER(A, $\left\lfloor \frac{x}{2} \right\rfloor$)
4. B = B $\times$ B
5. **if** x is odd
6. B = B $\times$ A
7. **return** B

Improving time

- So, if we raising a matrix to a power we can do as follows:

MATRIX-POWER(A, x)
1. **if** $x = 1$
2. **return** A
3. B = MATRIX-POWER(A, $\left\lfloor \frac{x}{2} \right\rfloor$)
4. B = B × B
5. **if** x is odd
6. B = B × A
7. **return** B

We need one more multiplication in case x is not even

Improving time

- So, if we raising a matrix to a power we can do as follows:

```
MATRIX-POWER(A, x)
    if x = 1
        return A
    B = MATRIX-POWER(A, ⌊x/2⌋)
    B = B × B
    if x is odd
        B = B × A
    return B
```

Question: How many multiplications do we need to compute A^x?

Improving time

- So, if we raising a matrix to a power we can do as follows:

MATRIX-POWER(A, x)
1. **if** $x = 1$
2. **return** A
3. B = MATRIX-POWER(A, $\left\lfloor \frac{x}{2} \right\rfloor$)
4. B = B × B
5. **if** x is odd
6. B = B × A
7. **return** B

Question: How many multiplications do we need to compute A^x?

Answer: $\mathrm{O}(\log x)$, since each time the power is divided by 2, and in each call we do at most 2 multiplications.

Improving time

- So, if we raising a matrix to a power we can do as follows:

MATRIX-POWER(A, x)
1. **if** $x = 1$
2.   **return** A
3. B = MATRIX-POWER(A, $\left\lfloor \frac{x}{2} \right\rfloor$)
4. B = B × B
5. **if** x is odd
6.   B = B × A
7. **return** B

Note that this idea is not limited to matrices. You can use it even if you want to compute x^y where x is an integer!

Improving time

- It’s possible to show that the extend operation of shortest paths **is also associative:**

A ***extended by*** (**B** ***extended by*** **C**) = (**A** ***extended by*** **B**) ***extended by*** **C**

- As a result, we can use the same idea to improve the running time.

Improving time

FAST-SHORTEST-PATHS(W, x)
1. **if** $x = 1$
2. **return** W
3. B = FAST-SHORTEST-PATHS (W, $\left\lfloor \frac{x}{2} \right\rfloor$)
4. B = EXTEND-SHORTEST-PATHS(B, B)
5. **if** x is odd
6. B = EXTEND-SHORTEST-PATHS(B, W)
7. **return** B

Improving time

- We initially call FAST-SHORTEST-PATHS(W, n-1)

FAST-SHORTEST-PATHS(W, x)
1. **if** $x = 1$
2. **return** W
3. B = FAST-SHORTEST-PATHS (W, $\left\lfloor \frac{x}{2} \right\rfloor$)
4. B = EXTEND-SHORTEST-PATHS(B, B)
5. **if** x is odd
6. B = EXTEND-SHORTEST-PATHS(B, W)
7. **return** B

Improving time

- With a similar argument, now the algorithm takes $\Theta(n^3 \log n)$

Transitive closure

- We present four solutions to this problem:

1. Using BFS ✔
2. Using strongly connected components (SCCs) ✔
3. Using matrix multiplication ✔:

- Has a time complexity of $\Theta(n^3 \log n)$
- Computes the shortest paths even if the graph is weighted

4. Using dynamic programming

Transitive closure

- We present four solutions to this problem:

1. Using BFS ✔
2. Using strongly connected components (SCCs) ✔
3. Using matrix multiplication ✔
4. Using dynamic programming: we will talk about this approach after a comprehensive discussion on dynamic programming

Faster matrix multiplication

- Matrix multiplication is a **fundamental problem** on its own and also an active area of research.

- Faster matrix multiplication will improve the running time of a lot of algorithms directly.

- Here, we show how we can multiply matrices faster using the **divide-and-conquer** technique.

Faster matrix multiplication

- Each Aij, Bij and Cij is an $\boldsymbol{\frac{n}{2}} \times \boldsymbol{\frac{n}{2}}$ **matrix**.

$$\underset{A_{n\times n}}{\begin{array}{|c|c|}\hline A11 & A12 \\ \hline A21 & A22 \\ \hline\end{array}} \times \underset{B_{n\times n}}{\begin{array}{|c|c|}\hline B11 & B12 \\ \hline B21 & B22 \\ \hline\end{array}} = \underset{C_{n\times n}}{\begin{array}{|c|c|}\hline C11 & C12 \\ \hline C21 & C22 \\ \hline\end{array}}$$

Faster matrix multiplication

- $C11 = (A11 \times B11) + (A12 \times B21)$

$$\underset{A_{n\times n}}{\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix}} \times \underset{B_{n\times n}}{\begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix}} = \underset{C_{n\times n}}{\begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix}}$$

Faster matrix multiplication

- Similarly, we can compute

$$C11 = (A11 \times B11) + (A12 \times B21)$$
$$C12 = (A11 \times B12) + (A12 \times B22)$$
$$C21 = (A21 \times B11) + (A22 \times B21)$$
$$C12 = (A21 \times B12) + (A22 \times B22)$$

- Where **each $\times$** corresponds to **a recursive call** of the algorithm.

Faster matrix multiplication

- Similarly, we can compute

$$C11 = (A11 \times B11) + (A12 \times B21)$$
$$C12 = (A11 \times B12) + (A12 \times B22)$$
$$C21 = (A21 \times B11) + (A22 \times B21)$$
$$C12 = (A21 \times B12) + (A22 \times B22)$$

- **For each $+$** we need **to compute the sum** of two $\frac{n}{2} \times \frac{n}{2}$ matrices.

Faster matrix multiplication

$$C11 = (A11 \times B11) + (A12 \times B21)$$
$$C12 = (A11 \times B12) + (A12 \times B22)$$
$$C21 = (A21 \times B11) + (A22 \times B21)$$
$$C12 = (A21 \times B12) + (A22 \times B22)$$

- **Question:** If $\boldsymbol{T(n)}$ is the running of the recursive algorithm for multiplying **two $\boldsymbol{n \times n}$ matrices**, what is the time for **each** $\times$?

Faster matrix multiplication

$$C11 = (A11 \times B11) + (A12 \times B21)$$
$$C12 = (A11 \times B12) + (A12 \times B22)$$
$$C21 = (A21 \times B11) + (A22 \times B21)$$
$$C12 = (A21 \times B12) + (A22 \times B22)$$

- **Question:** If $\boldsymbol{T(n)}$ is the running of the recursive algorithm for multiplying **two $\boldsymbol{n \times n}$ matrices**, what is the time for **each** $\times$?
- **Answer:** Each recursive multiplication of $\frac{n}{2} \times \frac{n}{2}$ matrices takes $\boldsymbol{T(\frac{n}{2})}$ time.

Faster matrix multiplication

$$C11 = (A11 \times B11) + (A12 \times B21)$$
$$C12 = (A11 \times B12) + (A12 \times B22)$$
$$C21 = (A21 \times B11) + (A22 \times B21)$$
$$C12 = (A21 \times B12) + (A22 \times B22)$$

- Moreover, **computing the sum** of two $n \times n$ takes as much as the number of entries in the matrices which is $\boldsymbol{n^2}$.

- So, **each** $+$ needs $\left(\frac{n}{2}\right)^2 = \frac{n^2}{4}$ operations. Since we have 4 such additions, they take $\boldsymbol{\Theta(n^2)}$ **time all together**.

Faster matrix multiplication

- All in all, the time complexity will be

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

- Which using the Master theorem is …

Faster matrix multiplication

- All in all, the time complexity will be

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

- Which using the Master theorem is $\boldsymbol{T(n) = \Theta(n^3)}$

Faster matrix multiplication

- A much more efficient way to use divide-and-conquer is by **Strassen's method:**

$$M1 = (A11 + A22)\times(B11 + B22)$$
$$M2 = (A21 + A22)\times B11$$
$$M3 = A11\times(B12 - B22)$$
$$M4 = A22\times(B21 - B11)$$
$$M5 = (A11 + A12)\times B22$$
$$M6 = (A21 - A11)\times(B11 + B12)$$
$$M7 = (A12 - A22)\times(B21 + B22)$$

$$C11 = M1 + M4 - M5 + \mathrm{M7}$$
$$C12 = M3 + M5$$
$$C21 = M2 + M4$$
$$C22 = M1 - M2 + M3 + \mathrm{M6}$$

Faster matrix multiplication

- This requires 7 multiplications (×)
- And 18 addition or subtraction $(+ \; or \; -)$

$$M1 = (A11 + A22) \times (B11 + B22)$$
$$M2 = (A21 + A22) \times B11$$
$$M3 = A11 \times (B12 - B22)$$
$$M4 = A22 \times (B21 - B11)$$
$$M5 = (A11 + A12) \times B22$$
$$M6 = (A21 - A11) \times (B11 + B12)$$
$$M7 = (A12 - A22) \times (B21 + B22)$$

$$C11 = M1 + M4 - M5 + \mathrm{M7}$$
$$C12 = M3 + M5$$
$$C21 = M2 + M4$$
$$C22 = M1 - M2 + M3 + \mathrm{M6}$$

Faster matrix multiplication

- 7 multiplications take $\boldsymbol{7T(\frac{n}{2})}$
- 18 addition or subtraction take $18\left(\frac{n^2}{4}\right) = \boldsymbol{\Theta(n^2)}$

- So, the running time could be described as

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

- Using Master theorem we have

$$T(n) =$$

Faster matrix multiplication

- 7 multiplications take $\boldsymbol{7T(\frac{n}{2})}$
- 18 addition or subtraction take $18\left(\frac{n^2}{4}\right) = \boldsymbol{\Theta(n^2)}$

- So, the running time could be described as
$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$
- Using Master theorem we have
$$T(n) = \Theta\left(n^{\log_2 7}\right) \approx \Theta(n^{2.8074})$$

Faster matrix multiplication

- If n is not a power of 2, we increase the size to the first power of 2 by adding extra rows and columns that are 0.

$$\left[\begin{array}{c|c} A & 0 \\ \hline 0 & 0 \end{array}\right] \times \left[\begin{array}{c|c} B & 0 \\ \hline 0 & 0 \end{array}\right] = \left[\begin{array}{c|c} A\times B & 0 \\ \hline 0 & 0 \end{array}\right]$$

History of matrix multiplication

- Strassen (1969) $O(n^{2.8074})$

History of matrix multiplication

- Strassen (1969) $O(n^{2.8074})$
- Coppersmith–Winograd (1990) $O(n^{2.375477})$

History of matrix multiplication

- Strassen (1969) $O(n^{2.8074})$
- Coppersmith–Winograd (1990) $O(n^{2.375477})$
- Davie-Stothers (2010) $O(n^{2.37369})$

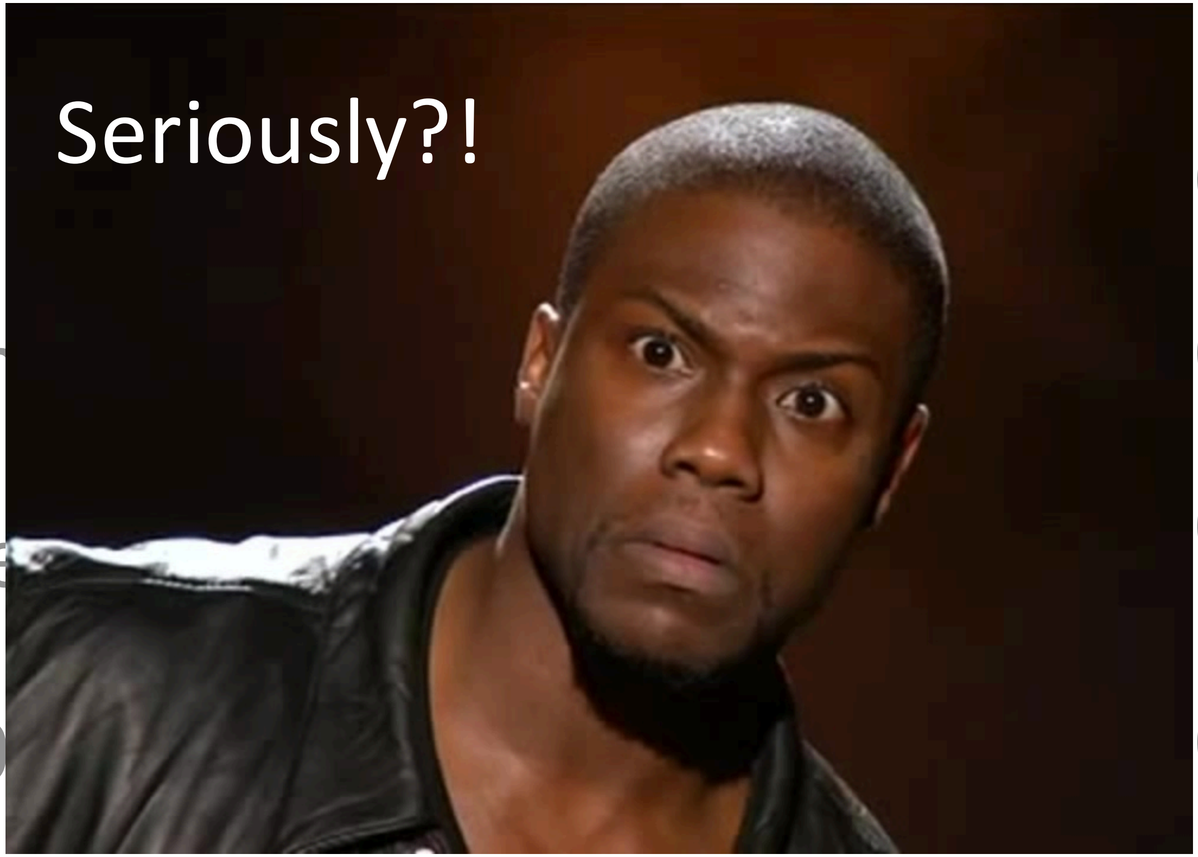
History of matrix multiplication

- Strassen (1969) $O(n^{2.8074})$
- Coppersmith–Winograd (1990) $O(n^{2.375477})$
- Davie-Stothers (2010) $O(n^{2.37369})$
- Williams (2011) $O(n^{2.3728642})$

History of matrix multiplication

- Strassen (1969) $O(n^{2.8074})$
- Coppersmith–Winograd (1990) $O(n^{2.375477})$
- Davie-Stothers (2010) $O(n^{2.37369})$
- Williams (2011) $O(n^{2.3728642})$
- François Le Gall (2014) $O(n^{2.3728639})$

History of matrix multiplication



- Strassen (19 $(n^{2.8074})$
- Coppersmith $(n^{2.375477})$
- Davie-Stothe $(n^{2.37369})$
- Williams (20 $(n^{2.3728642})$
- François Le Gall (2014) $O(n^{2.3728639})$

Faster matrix multiplication

- Note that even though the operation of extending shortest paths is similar to matrix multiplication, the Strassen's method can only be used for multiplying matrices and not for all arbitrary operations on matrices.