COMP90038
Algorithms and Complexity

Lecture 19: Warshall and Floyd

(with thanks to Harald Søndergaard & Michael Kirley)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

Recap

- **Dynamic programming** is a bottom-up problem solving technique. The idea is to divide the problem into smaller, overlapping ones. The results are tabulated and used to find the complete solution.
  - Solutions often involves recursion.
- Dynamic programming is often used on **Combinatorial Optimization** problems.
  - We are trying to find the **best** possible **combination** subject to some **constraints**
- Two classic problems
  - Coin row problem
  - Knapsack problem

The coin row problem

- You are shown a group of coins of different denominations ordered in a row.

- **You can keep some of them**, as long as you **do not pick two adjacent ones**.
  - Your objective is to **maximize your profit** , i.e., you want to take the largest amount of money.

- The solution can be expressed as the recurrence:

$$S(n) = \max\left(c_n + S\left(n-2\right), S\left(n-1\right)\right) \text{ for } n > 1$$

$$S(1) = c_1$$

$$S(0) = 0$$

The coin row problem

- Let’s quickly examine each step for [20 10 20 50 20 10 20]:

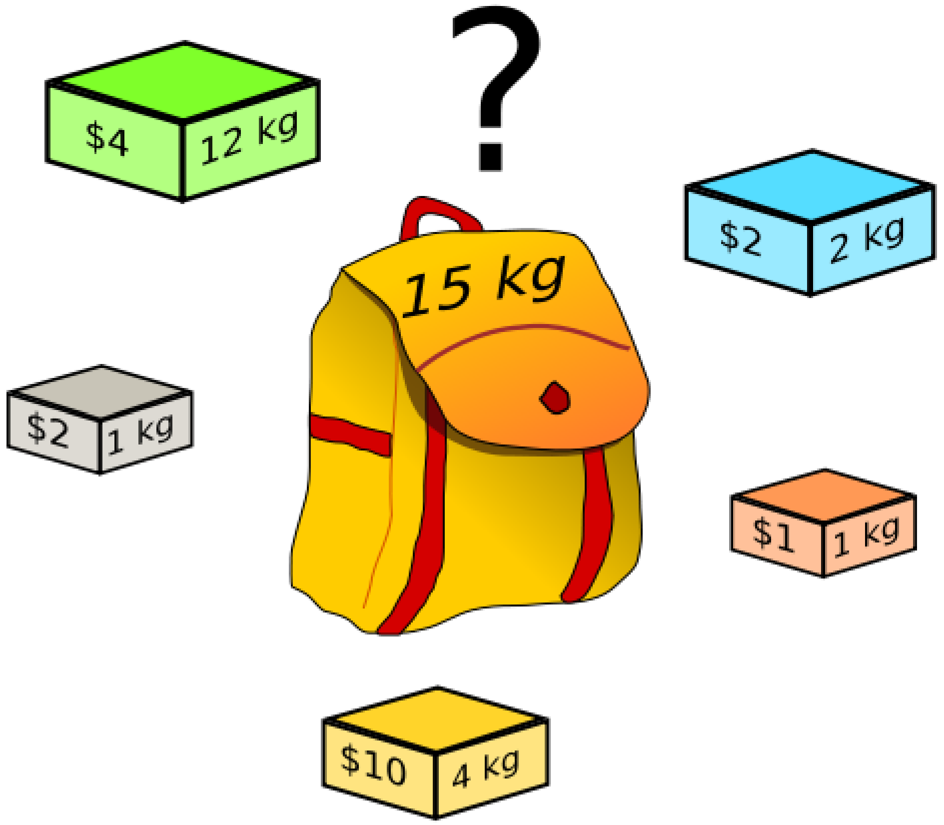
- S[0] = 0
- S[1] = 20
- S[2] = max(**S[1] = 20**, S[0] + 10 = 0 + 10) = 20
- S[3] = max(S[2] = 20, **S[1] + 20 = 20 + 20 = 40**) = 40
- S[4] = max(S[3] = 40, **S[2] + 50 = 20 + 50 = 70**) = 70
- S[5] = max(**S[4] = 70**, S[3] + 20 = 40 + 20 = 60) = 70
- S[6] = max(S[5] = 70, **S[4] + 10 = 70 + 10 = 80**) = 80
- S[7] = max(S[6] = 80, **S[5] + 20 = 70 + 20 = 90**) = 90

		1	2	3	4	5	6	7
	0	20	10	20	50	20	10	20
STEP 0	0							
STEP 1	0	20						
STEP 2	0	20	20					
STEP 3	0	20	20	40				
STEP 4	0	20	20	40	70			
STEP 5	0	20	20	40	70	70		
STEP 6	0	20	20	40	70	70	80	
STEP 7	0	20	20	40	70	70	80	90

		1	1	1	1	1	1	1
SOLUTION				3	4	4	4	4
							6	7

The knapsack problem

- We also talked about the **knapsack problem**:
- Given a list of n items with:
  - Weights $\{w_1, w_2, \ldots, w_n\}$
  - Values $\{v_1, v_2, \ldots, v_n\}$
- and a knapsack (container) of capacity W
- Find the **combination** of items with the **highest value** that would **fit into the knapsack**
- All values are positive integers



The knapsack problem

- The critical step is to find a good answer to the question: **what is the smallest version of the problem that I could solve first?**
  - Imagine that I have a knapsack of capacity 1, and an item of weight 2. **Does it fit?**
  - What if the capacity was 2 and the weight 1. Does it fit? **Do I have capacity left?**

- Given that we have **two variables**, the recurrence relation is formulated over **two parameters**:
  - the **sequence of items considered so far** {1, 2, … *i*}, and
  - the **remaining capacity** $w \leq W$.

- Let $K(i,w)$ be the value of the best choice of items amongst the first *i* using knapsack capacity *w*.
  - Then we are after $K(n,W)$.

The knapsack problem

- By focusing on $K(i,w)$ we can express a recursive solution.

- Once a new item i arrives, we can either pick it or not.
    - **Excluding *i*** means that the solution is $K(i\text{-}1,w)$, that is, which items were selected before i arrived with the same knapsack capacity.

    - **Including *i*** means that the solution also includes the subset of previous items **that will fit into a bag of capacity** $\boldsymbol{w\text{-}w_{\mathrm{i}}} \geq 0$, i.e., $K(i\text{-}1,w\text{-}w_i) + v_i$.

The knapsack problem

- This was expressed as a recursive function, with a base **state**:

$$K(i, w) = 0 \text{ if } i = 0 \text{ or } w = 0$$

- And a general case:

$$K(i,w) = \begin{cases} \max(K(i-1,w), K(i-1,w-w_i)+v_i) & \text{if } w \geq w_i \\ K(i-1,w) & \text{if } w < w_i \end{cases}$$

- Our example was:
  - The knapsack capacity W = 8
  - The values are {42, 12, 40, 25}
  - The weights are {7, 3, 4, 5}

The knapsack problem

- Did you complete the table?

```
for i ← 0 to n do
    K[i, 0] ← 0
for j ← 1 to W do
    K[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j < w_i then
            K[i, j] ← K[i − 1, j]
        else
            K[i, j] ← max(K[i − 1, j], K[i − 1, j − w_i] + v_i)
return K[n, W]
```

			j	0	1	2	3	4	5	6	7	8
v	*w*	*i*										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0	42	42
12	3	2		0	0	0	12	12	12	12	42	42
40	4	3		0	0	0	12	40	40	40		
25	5	4		0								

- $i = 3$
- $j = 7$
- $K[3\text{-}1,7] = K[2,7] = 42$
- $K[3\text{-}1,7\text{-}4] + 40 = K[2,3] + 40 = 12 + 40 = 52$

Solving the Knapsack Problem with Memoing

- To some extent the bottom-up (table-filling) solution is overkill:
  - It finds the solution to **every conceivable sub-instance**, most of which are unnecessary

- In this situation, a top-down approach, with **memoing**, is preferable.
  - There are many implementations of the memo table.
  - We will examine a simple array type implementation.

The knapsack problem

- Lets look at this algorithm, step-by-step

- The data is:
  - The knapsack capacity W = 8
  - The values are {42, 12, 40, 25}
  - The weights are {7, 3, 4, 5}

- *F* is initialized to all -1, with the exceptions of *i*=0 and *j*=0, which are initialized to 0.

```
function MFKNAP(i, j)
    if i < 1 or j < 1 then
        return 0
    if F(i, j) < 0 then
        if j < w(i) then
            value = MFKNAP(i − 1, j)
        else
            value = max(MFKNAP(i − 1, j), v(i) + MFKNAP(i − 1, j − w(i)))
        F(i, j) = value
    return F(i, j)
```

The knapsack problem

- We start with *i*=4 and *j*=8

			j	0	1	2	3	4	5	6	7	8
v	*w*	*i*										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		0	-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		0	-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		0	-1	-1	-1	-1	-1	-1	-1	**-1**

function $\text{MFK}\textsc{nap}(i, j)$
    if $i < 1$ or $j < 1$ **then**
        return 0
    if $F(i, j) < 0$ **then**
        if $j < w(i)$ **then**
            $value = \text{MFK}\textsc{nap}(i-1, j)$
        else
            $value = \max(\boxed{\text{MFK}\textsc{nap}(i-1, j)}, \boxed{v(i) + \text{MFK}\textsc{nap}(i-1, j-w(i)))}$
        $F(i, j) = value$
    return $F(i, j)$

- *i* = 4
- *j* = 8
- *K*[4-1,8] = K[3,8]
- *K*[4-1,8-5] + 25 = *K*[3,3] + 25

The knapsack problem

- Next is *i*=3 and *j*=8

			j	0	1	2	3	4	5	6	7	8
v	*w*	*i*										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		0	-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		0	-1	-1	-1	-1	-1	-1	-1	**-1**
25	5	4		0	-1	-1	-1	-1	-1	-1	-1	-1

```
function MFKnap(i, j)
    if i < 1 or j < 1 then
        return 0
    if F(i, j) < 0 then
        if j < w(i) then
            value = MFKnap(i − 1, j)
        else
            value = max(MFKnap(i − 1, j), v(i) + MFKnap(i − 1, j − w(i)))
        F(i, j) = value
    return F(i, j)
```

- *i* = 3
- *j* = 8
- *K*[3-1,8] = K[2,8]
- *K*[3-1,8-4] + 40 = *K*[2,4] + 40

The knapsack problem

- Next is i=2 and j=8

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		0	-1	-1	-1	-1	-1	-1	-1	**-1**
40	4	3		0	-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		0	-1	-1	-1	-1	-1	-1	-1	-1

```
function MFKnap(i, j)
    if i < 1 or j < 1 then
        return 0
    if F(i, j) < 0 then
        if j < w(i) then
            value = MFKnap(i − 1, j)
        else
            value = max(MFKnap(i − 1, j), v(i) + MFKnap(i − 1, j − w(i)))
        F(i, j) = value
    return F(i, j)
```

- i = 2
- j = 8
- K[2-1,8] = K[1,8]
- K[2-1,8-3] + 12 = K[1,5] + 12

The knapsack problem

- Next is *i*=1 and *j*=8
- Here we reach the bottom of this recursion

			j	0	1	2	3	4	5	6	7	8
v	*w*	*i*										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	-1	-1	-1	-1	-1	-1	-1	42
12	3	2		0	-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		0	-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		0	-1	-1	-1	-1	-1	-1	-1	-1

function $\text{MFK}\small{\text{NAP}}(i, j)$
  if $i < 1$ or $j < 1$ **then**
    return 0
  if $F(i, j) < 0$ **then**
    if $j < w(i)$ **then**
      $value = \text{MFK}\small{\text{NAP}}(i-1, j)$
    else
      $value = \max\left(\boxed{\text{MFKNAP}(i-1, j)}, \boxed{v(i) + \text{MFKNAP}(i-1, j-w(i))}\right)$
    $F(i, j) = value$
  return $F(i, j)$

- *i* = 1
- *j* = 8
- *K*[1-1,8] = K[0,8] = 0
- *K*[1-1,8-7] + 42 = *K*[0,1] + 42 = 0 + 42 = 42

The knapsack problem

- Next is *i*=1 and *j*=5.
- As before, we also reach the bottom of this branch.

			j	0	1	2	3	4	5	6	7	8
v	*w*	*i*										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	-1	-1	-1	-1	0	-1	-1	42
12	3	2		0	-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		0	-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		0	-1	-1	-1	-1	-1	-1	-1	-1

function $\text{MFKNAP}(i, j)$
  if $i < 1$ or $j < 1$ **then**
    return 0
  if $F(i, j) < 0$ **then**
    if $j < w(i)$ **then**
      $value = \text{MFKNAP}(i-1, j)$
    else
      $value = \max(\boxed{\text{MFKNAP}(i-1, j)}, v(i) + \boxed{\text{MFKNAP}(i-1, j-w(i))})$
    $F(i, j) = value$
  return $F(i, j)$

- *i* = 1
- *j* = 5
- *K*[1-1,5] = K[0,5] = 0
- *j* - *w*[1] = 5-8 < 1 → return 0

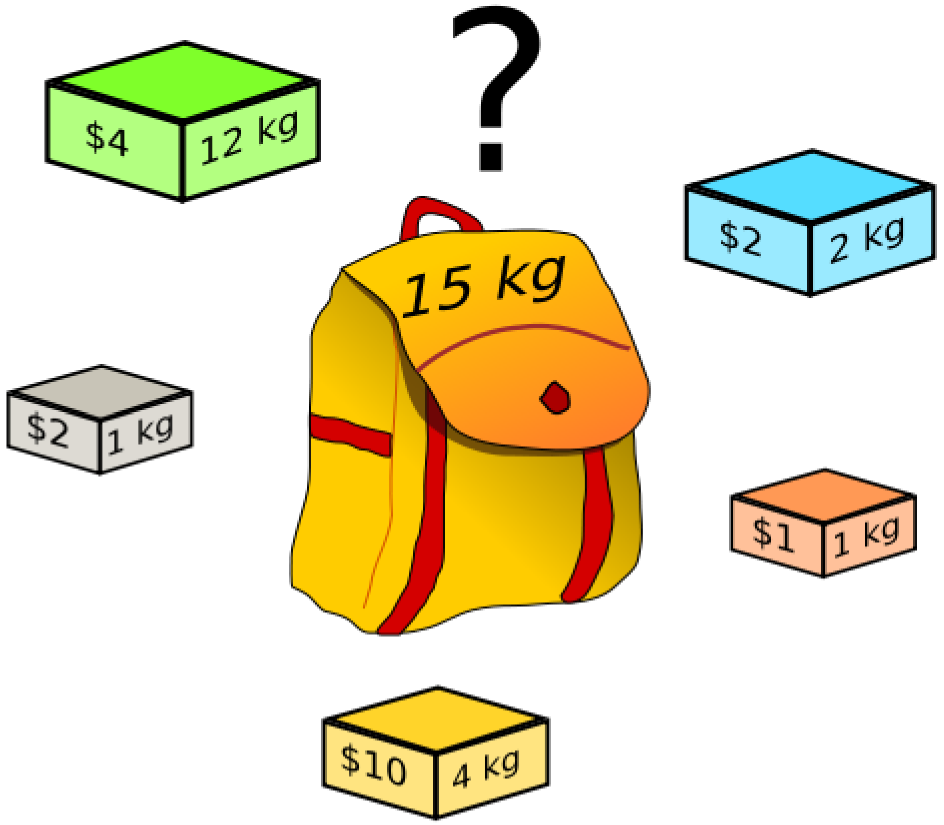
The knapsack problem

- We can trace the complete algorithm, until we find our solution.
- The states visited (18) are shown in the table.
  - Unlike the bottom-up approach, in which we visited all the states (40).
- Given that there are a lot of places in the table never used, the algorithm is less space-efficient.
  - You may use a hash table to improve space efficiency.

i	j	value
0	8	0
0	1	0
1	8	42
0	5	0
1	5	0
2	8	42
0	4	0
1	4	0
0	1	0
1	1	0
2	4	12
3	8	52
0	3	0
1	3	0
1	0	0
2	3	12
3	3	12
4	8	52

A practice challenge

- Can you solve the problem in the figure?
  - W = 15
  - w = [1 1 2 4 12]
  - v = [1 2 2 10 4]
- Because it is a larger instance, **memoing** is preferable.
  - How many states do we need to evaluate?
- FYI the answer is $15/15Kg

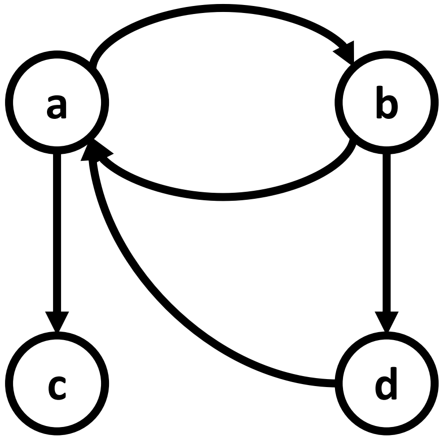


Dynamic Programming and Graphs

- We now apply dynamic programming to two graph problems:
  - Computing the **transitive closure** of a directed graph; and
  - **Finding shortest distances** in weighted directed graphs.

Warshall's algorithm

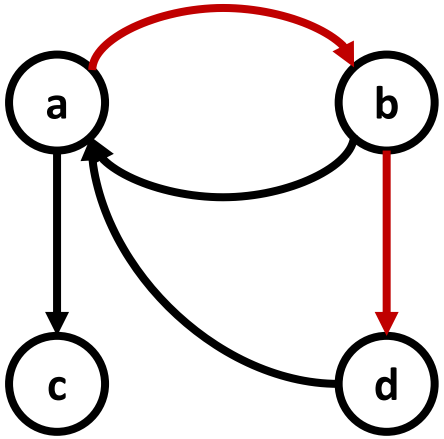
- Warshall's algorithm computes the **transitive closure** of a directed graph.
    - An edge (*a*,*d*) is in the transitive closure of graph *G* iff there is a path in *G* from *a* to *d*.
- Transitive closure is important in applications where we need to reach a "goal state" from some "initial state".
- Warshall's algorithm was not originally thought of as an instance of dynamic programming, but it fits the bill



$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Warshall’s algorithm

- Warshall's algorithm computes the **transitive closure** of a directed graph.
  - An edge (a,d) is in the transitive closure of graph G iff there is a path in G from a to d.
- Transitive closure is important in applications where we need to reach a “goal state” from some “initial state”.
- Warshall's algorithm was not originally thought of as an instance of dynamic programming, but it fits the bill



$$\begin{bmatrix} 0 & 1 & 1 & \boxed{1} \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Warshall's Algorithm

- Assume the nodes of graph *G* are numbered from 1 to *n*.
- **Is there a path** from node *i* to node *j* using nodes [1 … *k*] as "stepping stones"?
- Such path will exist if and only if we can:
  - step from *i* to *j* using only nodes [1 … *k*-1], or
  - step from *i* to *k* using only nodes [1 … *k*-1], and then step from *k* to *j* using only nodes [1 … *k*-1].

Warshall's Algorithm

- If *G*'s adjacency matrix is *A* then we can express the recurrence relation as:

$$R[i, j, 0] = A[i, j]$$

$$R[i, j, k] = R[i, j, k-1] \textbf{ or } (R[i, k, k-1] \textbf{ and } R[k, j, k-1])$$

- This gives us a dynamic programming algorithm:

$$
\begin{array}{l}
\textbf{function } \textsc{Warshall}(A[\cdot,\cdot], n) \\
\quad R[\cdot,\cdot,0] \leftarrow A \\
\quad \textbf{for } k \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad \textbf{for } i \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad\quad \textbf{for } j \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad\quad\quad R[i,j,k] \leftarrow R[i,j,k-1] \textbf{ or } (R[i,k,k-1] \textbf{ and } R[k,j,k-1]) \\
\quad \textbf{return } R[\cdot,\cdot,n]
\end{array}
$$

Warshall's Algorithm

- If we allow input *A* to be used for the output, we can simplify things.
  - If *R*[*i*,*k*,*k*-1] (that is, *A*[*i*,*k*]) is 0 then the assignment is doing nothing.
  - But if *A*[*i*,*k*] is 1 and if A[*k*,*j*] is also 1, then *A*[*i*,*j*] gets set to 1.

$$
\begin{array}{l}
\textbf{for } k \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad \textbf{for } i \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad \textbf{for } j \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad\quad \textbf{if } A[i,k] \textbf{ then} \\
\quad\quad\quad\quad \textbf{if } A[k,j] \textbf{ then} \\
\quad\quad\quad\quad\quad A[i,j] \leftarrow 1
\end{array}
$$

- But now we notice that *A*[*i*,*k*] does not depend on *j*, so testing it can be moved outside the innermost loop.

Warshall's Algorithm

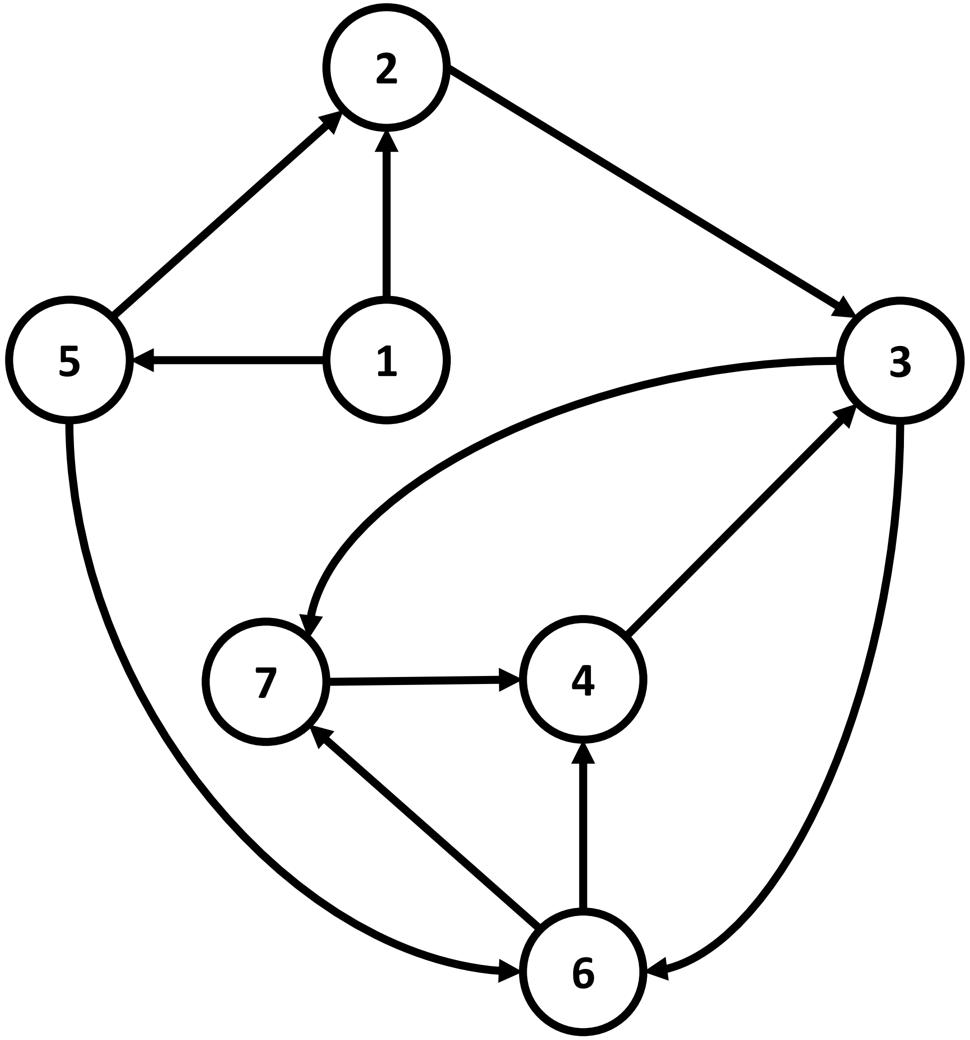
- This leads to a simpler version of the algorithm.

$$
\begin{array}{l}
\textbf{for } k \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad \textbf{for } i \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad \textbf{if } A[i,k] \textbf{ then} \\
\quad\quad\quad \textbf{for } j \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad\quad\quad \textbf{if } A[k,j] \textbf{ then} \\
\quad\quad\quad\quad\quad A[i,j] \leftarrow 1
\end{array}
$$

- If each row in the matrix is represented as a bit-string, then the innermost for loop (and *j*) can be gotten rid of – instead of iterating, just apply the "bitwise or" of row *k* to row *i*.

Warshall's Algorithm

- Let's examine this algorithm. Let our graph be.



- Then, the adjacency matrix is:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Warshall's Algorithm

- For k=1, all the elements in the column are zero, so this **if** statement does nothing.

$$
\begin{aligned}
&\textbf{for } k \leftarrow 1 \text{ to } n \textbf{ do}\\
&\quad \textbf{for } i \leftarrow 1 \text{ to } n \textbf{ do}\\
&\quad\quad \boxed{\textbf{if } A[i,k] \textbf{ then}}\\
&\quad\quad\quad \textbf{for } j \leftarrow 1 \text{ to } n \textbf{ do}\\
&\quad\quad\quad\quad \textbf{if } A[k,j] \textbf{ then}\\
&\quad\quad\quad\quad\quad A[i,j] \leftarrow 1
\end{aligned}
$$

$$
\begin{bmatrix}
0 & 1 & 0 & 0 & 1 & 0 & 0\\
0 & 0 & 1 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 1 & 1\\
0 & 0 & 1 & 0 & 0 & 0 & 0\\
0 & 1 & 0 & 0 & 0 & 1 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 1\\
0 & 0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix}
$$

Warshall's Algorithm

- For k=2, we have **A[1,2] = 1** and **A[5,2] = 1**, and **A[2,3]=1**

$$
\begin{aligned}
&\textbf{for } k \leftarrow 1 \text{ to } n \textbf{ do}\\
&\quad \textbf{for } i \leftarrow 1 \text{ to } n \textbf{ do}\\
&\quad\quad \textbf{if } A[i,k] \textbf{ then}\\
&\quad\quad\quad \textbf{for } j \leftarrow 1 \text{ to } n \textbf{ do}\\
&\quad\quad\quad\quad \textbf{if } A[k,j] \textbf{ then}\\
&\quad\quad\quad\quad\quad A[i,j] \leftarrow 1
\end{aligned}
$$

$$
\begin{bmatrix}
0 & 1 & 0 & 0 & 1 & 0 & 0\\
0 & 0 & 1 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 1 & 1\\
0 & 0 & 1 & 0 & 0 & 0 & 0\\
0 & 1 & 0 & 0 & 0 & 1 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 1\\
0 & 0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix}
$$

Warshall's Algorithm

- For k=2, we have **A[1,2] = 1** and **A[5,2] = 1**, and **A[2,3]=1**
  - Then, we can make **A[1,3] = 1** and **A[5,3] = 1**

$$\begin{array}{l}
\textbf{for } k \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad \textbf{for } i \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad \textbf{if } A[i,k] \textbf{ then} \\
\quad\quad\quad \textbf{for } j \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad\quad\quad \boxed{\begin{array}{l} \textbf{if } A[k,j] \textbf{ then} \\ \quad A[i,j] \leftarrow 1 \end{array}}
\end{array}$$

$$\begin{bmatrix}
0 & 1 & \boxed{1} & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & \boxed{1} & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix}$$

Warshall's Algorithm

- For k=3, we have **A[1,3]**, **A[2,3]**, **A[4,3]**, **A[5,3]**, A[3,6] and A[3,7] equal to 1

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

for $k \leftarrow 1$ to n **do**
  for $i \leftarrow 1$ to n **do**
    if $A[i,k]$ **then**
      for $j \leftarrow 1$ to n **do**
        if $A[k,j]$ **then**
          $A[i,j] \leftarrow 1$

Warshall's Algorithm

- For k=3, we have **A[1,3]**, **A[2,3]**, **A[4,3]**, **A[5,3]**, **A[3,6]** and **A[3,7]** equal to 1
  - Then, we can make **A[1,6]**, **A[2,6]**, **A[4,6]**, **A[1,7]**, **A[2,7]**, **A[4,7]**, and **A[5,7]** equal to 1.

$$
\begin{array}{l}
\textbf{for } k \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad \textbf{for } i \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad \textbf{if } A[i,k] \textbf{ then} \\
\quad\quad\quad \textbf{for } j \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad\quad\quad \textbf{if } A[k,j] \textbf{ then} \\
\quad\quad\quad\quad\quad A[i,j] \leftarrow 1
\end{array}
$$

$$
\begin{bmatrix}
0 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix}
$$

Warshall’s algorithm

- Let’s look at the final steps:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

k=3

Warshall’s algorithm

- Let’s look at the final steps:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}$$

k=3 k=4

Warshall's algorithm

- Let's look at the final steps:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

k=3

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}$$

k=4

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & \boxed{1} & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}$$

k=6

Warshall’s algorithm

- Let’s look at the final steps:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

k=3

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}$$

k=4

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & \boxed{1} & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}$$

k=6

- At k=5 and k=7, there is no changes to the matrix.

Warshall’s algorithm

- Warshall's algorithm’s complexity is $\Theta(n^3)$. There is **no difference** between the best, average, and worst cases.

- The algorithm has an incredibly tight inner loop, making it **ideal for dense graphs**.

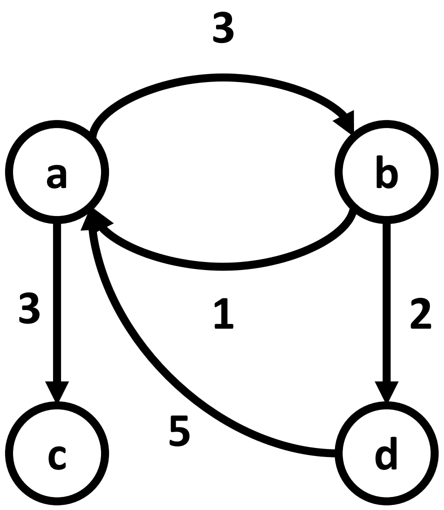
- However, it is **not the best** transitive-closure algorithm to use for **sparse graphs**.
  - For sparse graphs, you may be better off just doing DFS from each node v in turn, keeping track of which nodes are reached from v.

Floyd's Algorithm

- Floyd's algorithm solves the **all-pairs shortest-path** problem for weighted graphs with **positive weights.**
  - It works for **directed** as well as **undirected** graphs.

- We assume we are given a **weight matrix** W that holds all the edges' weights
  - If there is no edge from node i to node j, we set $W[i,j] = \infty$.

- We will construct the **distance matrix** D, step by step.



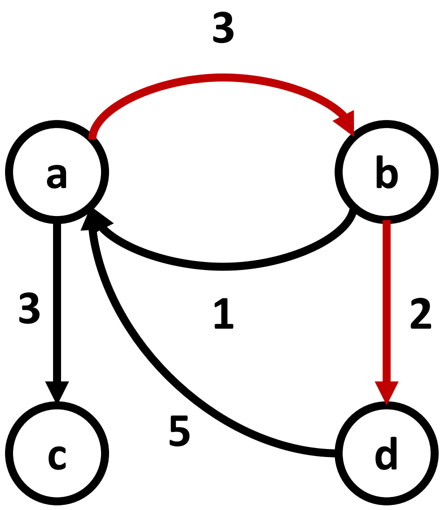
$$\begin{bmatrix} 0 & 3 & 3 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 \end{bmatrix}$$

Floyd's Algorithm

- Floyd's algorithm solves the **all-pairs shortest-path** problem for weighted graphs with **positive weights**.
  - It works for **directed** as well as **undirected** graphs.

- We assume we are given a **weight matrix** *W* that holds all the edges' weights
  - If there is no edge from node *i* to node *j*, we set $W[i,j] = \infty$.

- We will construct the **distance matrix** *D*, step by step.



$$\begin{bmatrix} 0 & 3 & 3 & \boxed{5} \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 \end{bmatrix}$$

Floyd's Algorithm

- As we did in the Warshall's algorithm, assume nodes are numbered 1 to *n*.
- **What is the shortest path** from node *i* to node *j* using nodes [1 … *k*] as “stepping stones”?
- Such path will exist if and only if we can:
  - step from *i* to *j* using only nodes [1 … *k*-1], or
  - step from *i* to *k* using only nodes [1 … *k*-1], and then step from *k* to *j* using only nodes [1 … *k*-1].

Floyd's Algorithm

- If *G*’s weight matrix is *W* then we can express the recurrence relation as:

$$D[i, j, 0] = W[i, j]$$

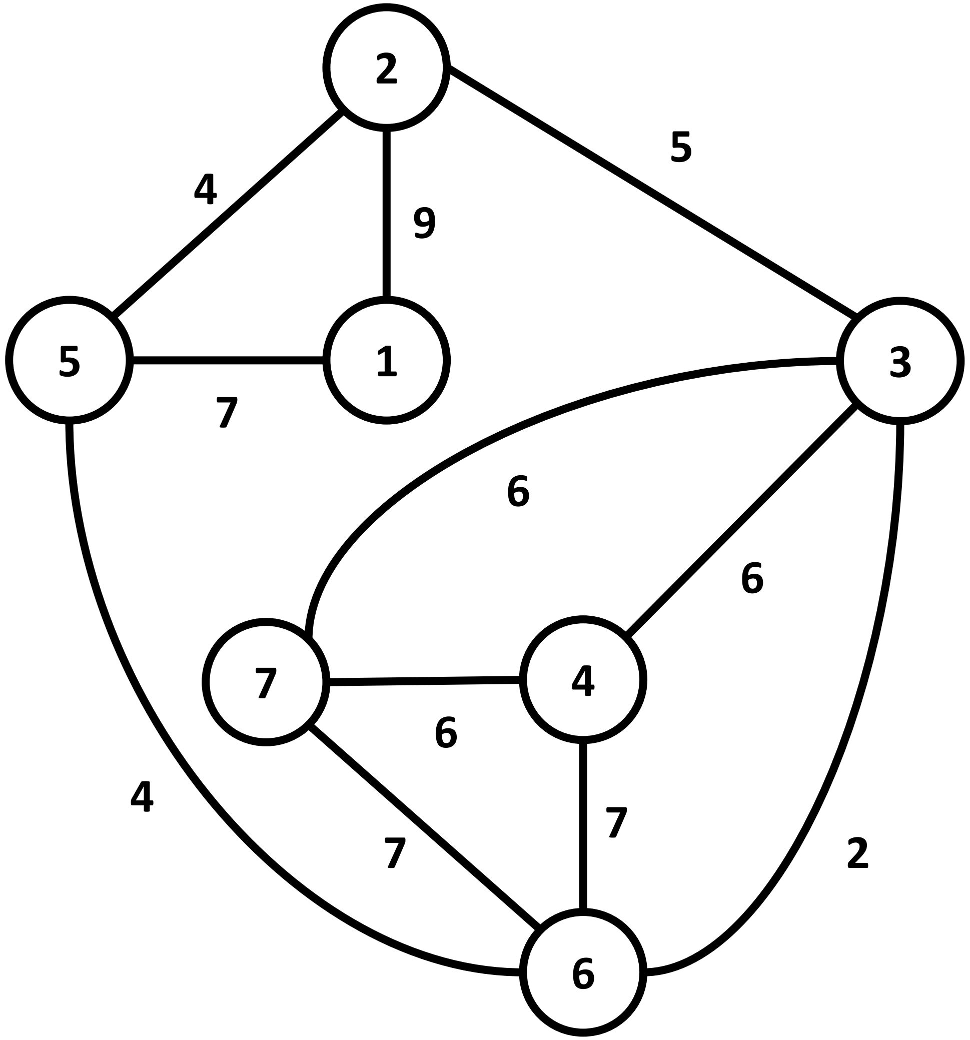
$$D[i, j, k] = \min\left(D[i, j, k-1], D[i, k, k-1] + D[k, j, k-1]\right)$$

- A simpler version updating D:

$$
\begin{array}{l}
\textbf{function } \textsc{Floyd}(W[\cdot,\cdot], n) \\
\quad D \leftarrow W \\
\quad \textbf{for } k \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad \textbf{for } i \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad\quad \textbf{for } j \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad\quad\quad D[i, j] \leftarrow \min\left(D[i, j], D[i, k] + D[k, j]\right) \\
\quad \textbf{return } D
\end{array}
$$

Floyd's Algorithm

- Let’s examine this algorithm. Let our graph be.



- Then, the weight matrix is:

$$\begin{bmatrix} 0 & 9 & \infty & \infty & 7 & \infty & \infty \\ 9 & 0 & 5 & \infty & 4 & \infty & \infty \\ \infty & 5 & 0 & 6 & \infty & 2 & 6 \\ \infty & \infty & 6 & 0 & \infty & 7 & 6 \\ 7 & 4 & \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 7 & 4 & 0 & 7 \\ \infty & \infty & 6 & 6 & \infty & 7 & 0 \end{bmatrix}$$

Floyd's Algorithm

- For k=1 there are no changes.

$$
\begin{array}{l}
\textbf{function } \textsc{Floyd}(W[\cdot,\cdot], n) \\
\quad D \leftarrow W \\
\quad \textbf{for } k \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad \textbf{for } i \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad\quad \textbf{for } j \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad\quad\quad\quad D[i,j] \leftarrow \min\left(D[i,j], D[i,k] + D[k,j]\right) \\
\quad \textbf{return } D
\end{array}
$$

$$
\begin{bmatrix}
0 & 9 & \infty & \infty & 7 & \infty & \infty \\
9 & 0 & 5 & \infty & 4 & \infty & \infty \\
\infty & 5 & 0 & 6 & \infty & 2 & 6 \\
\infty & \infty & 6 & 0 & \infty & 7 & 6 \\
7 & 4 & \infty & \infty & 0 & 4 & \infty \\
\infty & \infty & 2 & 7 & 4 & 0 & 7 \\
\infty & \infty & 6 & 6 & \infty & 7 & 0
\end{bmatrix}
$$

Floyd's Algorithm

- For k=2, **D[1,2] = 9** and **D[2,3]=5**; and **D[4,2] = 4** and **D[2,3]=5**.
  - Hence, we can make **D[1,3]=14** and **D[4,3]=9**
  - Note that the graph is undirected, which makes the matrix symmetric.

function $\textsc{Floyd}(W[\cdot,\cdot], n)$
  $D \leftarrow W$
  for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
      for $j \leftarrow 1$ to n **do**
        $D[i,j] \leftarrow \min\left(D[i,j], D[i,k] + D[k,j]\right)$
  return D

$$\begin{bmatrix} 0 & \boxed{9} & \infty & \infty & 7 & \infty & \infty \\ 9 & 0 & 5 & \infty & 4 & \infty & \infty \\ \infty & 5 & 0 & 6 & \infty & 2 & 6 \\ \infty & \infty & 6 & 0 & \infty & 7 & 6 \\ 7 & 4 & \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 7 & 4 & 0 & 7 \\ \infty & \infty & 6 & 6 & \infty & 7 & 0 \end{bmatrix}$$

Floyd's Algorithm

- For k=2, **D[1,2] = 9** and **D[2,3]=5**; and **D[4,2] = 4** and **D[2,3]=5**.
  - Hence, we can make **D[1,3]=14** and **D[4,3]=9**
  - Note that the graph is undirected, which makes the matrix symmetric.

$$\begin{bmatrix} 0 & 9 & \boxed{14} & \infty & 7 & \infty & \infty \\ 9 & 0 & 5 & \infty & 4 & \infty & \infty \\ \boxed{14} & 5 & 0 & 6 & \boxed{9} & 2 & 6 \\ \infty & \infty & 6 & 0 & \infty & 7 & 6 \\ 7 & 4 & \boxed{9} & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 7 & 4 & 0 & 7 \\ \infty & \infty & 6 & 6 & \infty & 7 & 0 \end{bmatrix}$$

function $\textsc{Floyd}(W[\cdot,\cdot], n)$
    $D \leftarrow W$
    for $k \leftarrow 1$ to n **do**
        for $i \leftarrow 1$ to n **do**
            for $j \leftarrow 1$ to n **do**
                $D[i,j] \leftarrow \min\left(D[i,j], D[i,k] + D[k,j]\right)$
    return D

Floyd's Algorithm

- For k=3, we can reach all other nodes in the graph.
  - However, these may not be the shortest paths.

$$\begin{bmatrix} 0 & 9 & 14 & \infty & 7 & \infty & \infty \\ 9 & 0 & 5 & \infty & 4 & \infty & \infty \\ 14 & 5 & 0 & 6 & 9 & 2 & 6 \\ \infty & \infty & 6 & 0 & \infty & 7 & 6 \\ 7 & 4 & 9 & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 7 & 4 & 0 & 7 \\ \infty & \infty & 6 & 6 & \infty & 7 & 0 \end{bmatrix}$$

function $\textsc{Floyd}(W[\cdot,\cdot], n)$
    $D \leftarrow W$
    for $k \leftarrow 1$ to n **do**
        for $i \leftarrow 1$ to n **do**
            for $j \leftarrow 1$ to n **do**
                $D[i,j] \leftarrow \min\left(D[i,j], D[i,k] + D[k,j]\right)$
    return D

Floyd's Algorithm

- For *k*=3, we can reach all other nodes in the graph.
  - However, these may not be the shortest paths.

$$\begin{bmatrix} 0 & 9 & 14 & \boxed{20} & 7 & \boxed{16} & \boxed{20} \\ 9 & 0 & 5 & \boxed{11} & 4 & \boxed{7} & \boxed{11} \\ 14 & 5 & 0 & 6 & 9 & 2 & 6 \\ \boxed{20} & \boxed{11} & 6 & 0 & \boxed{15} & 7 & 6 \\ 7 & 4 & 9 & \boxed{15} & 0 & 4 & \boxed{15} \\ \boxed{16} & \boxed{7} & 2 & 7 & 4 & 0 & 7 \\ \boxed{20} & \boxed{11} & 6 & 6 & \boxed{15} & 7 & 0 \end{bmatrix}$$

function $\textsc{Floyd}(W[\cdot,\cdot], n)$
    $D \leftarrow W$
    for $k \leftarrow 1$ to n **do**
        for $i \leftarrow 1$ to n **do**
            for $j \leftarrow 1$ to n **do**
                $D[i,j] \leftarrow \min\left(D[i,j], D[i,k] + D[k,j]\right)$
    return D

Floyd's Algorithm

- Let’s look at the final steps:

$$\begin{bmatrix} 0 & 9 & 14 & 20 & 7 & 16 & 20 \\ 9 & 0 & 5 & 11 & 4 & 7 & 11 \\ 14 & 5 & 0 & 6 & 9 & 2 & 6 \\ 20 & 11 & 6 & 0 & 15 & 7 & 6 \\ 7 & 4 & 9 & 15 & 0 & 4 & 15 \\ 16 & 7 & 2 & 7 & 4 & 0 & 7 \\ 20 & 11 & 6 & 6 & 15 & 7 & 0 \end{bmatrix}$$

k=4

Floyd's Algorithm

- Let's look at the final steps:

$$\begin{bmatrix} 0 & 9 & 14 & 20 & 7 & 16 & 20 \\ 9 & 0 & 5 & 11 & 4 & 7 & 11 \\ 14 & 5 & 0 & 6 & 9 & 2 & 6 \\ 20 & 11 & 6 & 0 & 15 & 7 & 6 \\ 7 & 4 & 9 & 15 & 0 & 4 & 15 \\ 16 & 7 & 2 & 7 & 4 & 0 & 7 \\ 20 & 11 & 6 & 6 & 15 & 7 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 9 & 14 & 20 & 7 & \boxed{11} & 20 \\ 9 & 0 & 5 & 11 & 4 & 7 & 11 \\ 14 & 5 & 0 & 6 & 9 & 2 & 6 \\ 20 & 11 & 6 & 0 & 15 & 7 & 6 \\ 7 & 4 & 9 & 15 & 0 & 4 & 15 \\ \boxed{11} & 7 & 2 & 7 & 4 & 0 & 7 \\ 20 & 11 & 6 & 6 & 15 & 7 & 0 \end{bmatrix}$$

k=4 k=5

Floyd's Algorithm

- Let’s look at the final steps:

$$\begin{bmatrix} 0 & 9 & 14 & 20 & 7 & 16 & 20 \\ 9 & 0 & 5 & 11 & 4 & 7 & 11 \\ 14 & 5 & 0 & 6 & 9 & 2 & 6 \\ 20 & 11 & 6 & 0 & 15 & 7 & 6 \\ 7 & 4 & 9 & 15 & 0 & 4 & 15 \\ 16 & 7 & 2 & 7 & 4 & 0 & 7 \\ 20 & 11 & 6 & 6 & 15 & 7 & 0 \end{bmatrix}$$

k=4

$$\begin{bmatrix} 0 & 9 & 14 & 20 & 7 & 11 & 20 \\ 9 & 0 & 5 & 11 & 4 & 7 & 11 \\ 14 & 5 & 0 & 6 & 9 & 2 & 6 \\ 20 & 11 & 6 & 0 & 15 & 7 & 6 \\ 7 & 4 & 9 & 15 & 0 & 4 & 15 \\ 11 & 7 & 2 & 7 & 4 & 0 & 7 \\ 20 & 11 & 6 & 6 & 15 & 7 & 0 \end{bmatrix}$$

k=5

$$\begin{bmatrix} 0 & 9 & \boxed{13} & \boxed{18} & 7 & 11 & \boxed{18} \\ 9 & 0 & 5 & 11 & 4 & 7 & 11 \\ \boxed{13} & 5 & 0 & 6 & \boxed{6} & 2 & 6 \\ \boxed{18} & 11 & 6 & 0 & \boxed{11} & 7 & 6 \\ 7 & 4 & \boxed{6} & \boxed{11} & 0 & 4 & \boxed{11} \\ 11 & 7 & 2 & 7 & 4 & 0 & 7 \\ \boxed{18} & 11 & 6 & 6 & \boxed{11} & 7 & 0 \end{bmatrix}$$

k=6

Floyd's Algorithm

- Let’s look at the final steps:

$$
\begin{bmatrix}
0 & 9 & 14 & 20 & 7 & 16 & 20 \\
9 & 0 & 5 & 11 & 4 & 7 & 11 \\
14 & 5 & 0 & 6 & 9 & 2 & 6 \\
20 & 11 & 6 & 0 & 15 & 7 & 6 \\
7 & 4 & 9 & 15 & 0 & 4 & 15 \\
16 & 7 & 2 & 7 & 4 & 0 & 7 \\
20 & 11 & 6 & 6 & 15 & 7 & 0
\end{bmatrix}
\quad
\begin{bmatrix}
0 & 9 & 14 & 20 & 7 & 11 & 20 \\
9 & 0 & 5 & 11 & 4 & 7 & 11 \\
14 & 5 & 0 & 6 & 9 & 2 & 6 \\
20 & 11 & 6 & 0 & 15 & 7 & 6 \\
7 & 4 & 9 & 15 & 0 & 4 & 15 \\
11 & 7 & 2 & 7 & 4 & 0 & 7 \\
20 & 11 & 6 & 6 & 15 & 7 & 0
\end{bmatrix}
\quad
\begin{bmatrix}
0 & 9 & \boxed{13} & \boxed{18} & 7 & 11 & \boxed{18} \\
9 & 0 & 5 & 11 & 4 & 7 & 11 \\
\boxed{13} & 5 & 0 & 6 & \boxed{6} & 2 & 6 \\
\boxed{18} & 11 & 6 & 0 & \boxed{11} & 7 & 6 \\
7 & 4 & \boxed{6} & \boxed{11} & 0 & 4 & \boxed{11} \\
11 & 7 & 2 & 7 & 4 & 0 & 7 \\
\boxed{18} & 11 & 6 & 6 & \boxed{11} & 7 & 0
\end{bmatrix}
$$

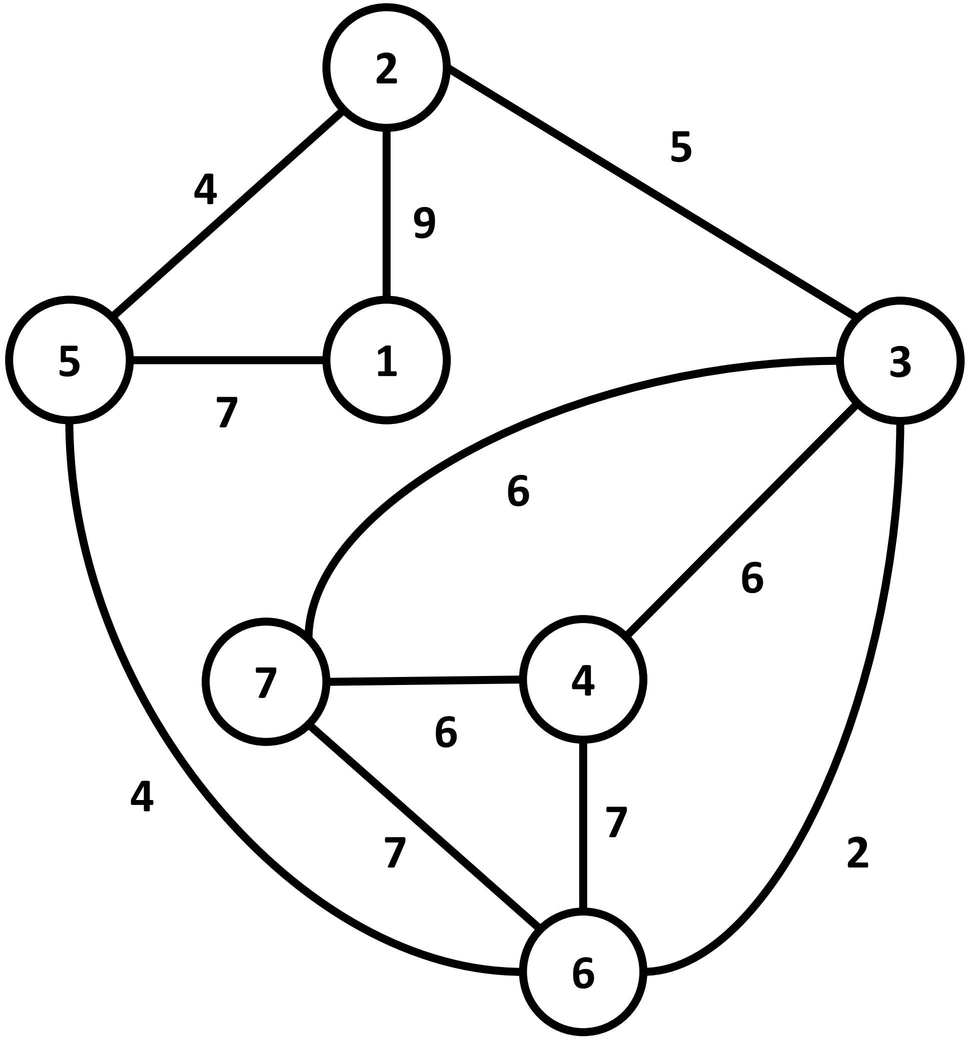
k=4 k=5 k=6

- For k=7, it is unchanged. So we have found the best paths.

A Sub-Structure Property

- For a DP approach to be applicable, the problem must have a "**sub-structure**" that allows for a compositional solution.
  - Shortest-path problems have this property. For example, if $\{x_1, x_2, \ldots, x_i, \ldots, x_n\}$ is a shortest path from x_1 to x_n then $\{x_1, x_2, \ldots, x_i\}$ is a shortest path from x_1 to $\cancel{x_n}$. Xi

- Longest-path problems don't have that property.
  - In our sample graph, {1,2,5,6,7,4,3} is a longest path from 1 to 3, but {1,2} is not a longest path from 1 to 2 (since {1,5,6,7,4,3,2} is longer).



Next lecture

- Greedy algorithms