THE UNIVERSITY OF MELBOURNE

COMP90038
Algorithms and Complexity

Lecture 19: Warshall and Floyd

(with thanks to Harald Søndergaard & Michael Kirley)

Casey Myers
Casey.Myers@unimelb.edu.au
David Caro Building (Physics) 274

Review from Lecture 18: Dynamic Programming

- Dynamic programming is an algorithm design technique that is sometimes applicable when we want to solve a recurrence relation and the recursion involves overlapping instances.

- In Lecture 16 we achieved a spectacular performance improvement in the calculation of Fibonacci numbers by switching from a naïve top-down algorithm to one that solved, and tabulated, smaller sub-problems.

- The bottom-up approach used the tabulated results, rather than solving overlapping sub-problems repeatedly.

- That was a particularly simple example of dynamic programming.

Review from Lecture 18: Dynamic Programming

- Since all values S(1) to S(n) need to be found anyway, we may as well proceed from the bottom up, storing intermediate results in an array S as we go.

- Given an array C that holds the coin values, the recurrence relation tells us what to do:

function CoinRow($C[1..n]$)
    $S[0] \leftarrow 0$
    $S[1] \leftarrow C[1]$
    for $i \leftarrow 2$ to n **do**
        $S[i] \leftarrow max\{S[i-1], S[i-2] + C[i]\}$
    return $S[n]$

Review from Lecture 18: Dynamic Programming

- We can say the same thing formally, as a recurrence relation:

$$S(i) = max\{S(i-1), S(i-2) + v_i\}$$

- This holds for $i > 1$.

- We need two base cases: $S(0) = 0$ and $S(1) = v_1$.

Review from Lecture 18: Dynamic Programming

- In Lecture 5 we looked at the knapsack problem.
- Given n items with
  - weights: $w_1, w_2, \cdots, w_n$
  - values: $v_1, v_2, \cdots, v_n$
  - knapsack of weight capacity W.
- Find the most valuable selection of items that will fit in the knapsack.
- We assume that all entities involved are positive integers.

Review from Lecture 18: Dynamic Programming

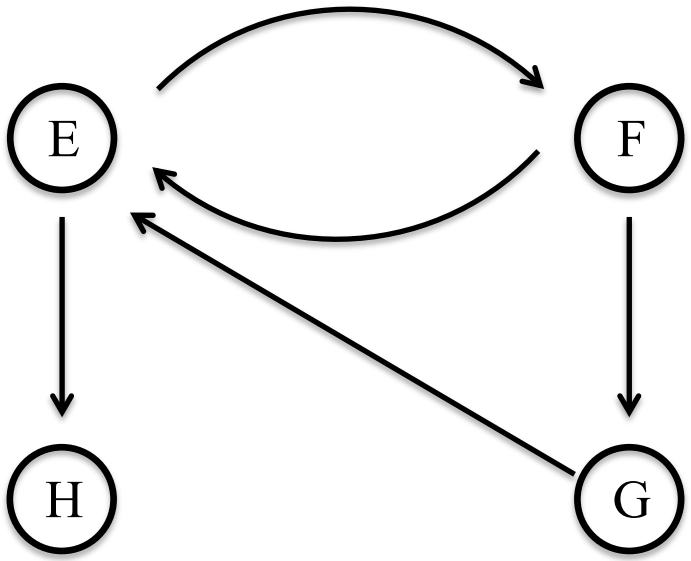
- First fill the leftmost column and top row, then proceed row by row:

```
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]
```

- The algorithm has time (and space) complexity $\Theta(nW)$.

Directed Graphs

- A directed graph:

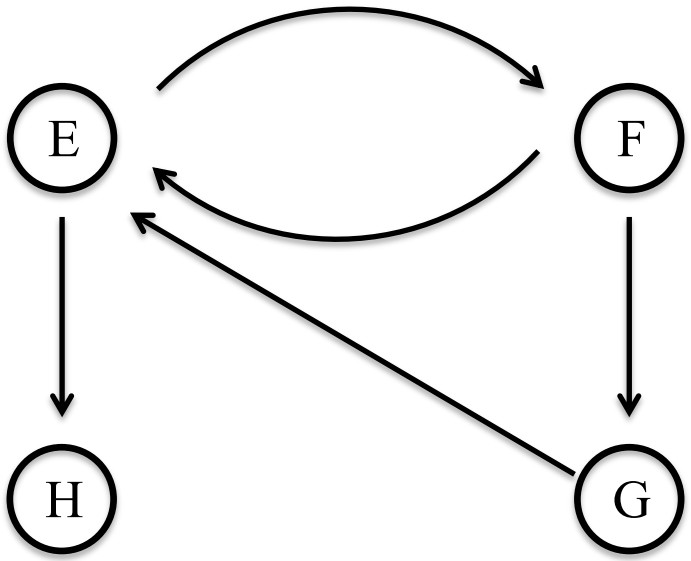


- And the adjacency matrix for this graph:

	E	F	G	H
E	0	1	0	1
F	1	0	1	0
G	1	0	0	0
H	0	0	0	0

Directed Graphs

- A directed graph:



- And the adjacency matrix for this graph:

	E	F	G	H
E	0	1	1	1
F	1	0	1	0
G	1	0	0	0
H	0	0	0	0

Dynamic Programming and Graphs

- In the last lecture we looked at dynamic programming.
- We now apply dynamic programming to two graph problems: For dynamic programming to be useful, the optimality principle must hold:
  - 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 binary relation (or a directed graph), presented as a matrix.
- An edge (a, z) is in the transitive closure of graph G iff there is a path in G from a to z.
- Warshall's algorithm was not originally thought of as an instance of dynamic programming but it fits the bill.

Transitive Closure over a State Space

- Transitive closure is important in all sorts of applications where we want to see of some "goal start" is reachable from some "initial state".

- For example is there a sequence of steps that will allow the containers of a ship to be reorganised in a certain way?

Reasoning about Transitive Closure

- Assume the nodes of graph G are numbered from 1 to n.
- Ask the question: Is there a path from node *i* to node *j* using only nodes that are no larger than some k as "stepping stones"?

Reasoning about Transitive Closure

- Assume the nodes of graph G are numbered from 1 to n.
- Ask the question: Is there a path from node i to node j using only nodes that are no larger than some k as "stepping stones"?
- Such a path either uses node k as a stepping stone, or it doesn't.
- So an answer is: There is such a path if and only if we can

  step from i to j using only nodes $\leq k-1$, or

  step from i to k using only nodes $\leq k-1$, and then
  step from k to j using only nodes $\leq k-1$.

Warshall's Algorithm

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

$$R_{ij}^{0} = A[i, j]$$
$$R_{ij}^{k} = R_{ij}^{k-1} \text{ or } \left(R_{ik}^{k-1} \text{ and } R_{kj}^{k-1}\right)$$

Warshall's Algorithm

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

$$R_{ij}^0 = A[i,j]$$

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

Use the existing path created in the previous step

Warshall's Algorithm

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

$$R_{ij}^0 = A[i,j]$$
$$R_{ij}^k = R_{ij}^{k-1} \text{ or } \left(R_{ik}^{k-1} \text{ and } R_{kj}^{k-1}\right)$$

Or create a new path using k as an intermediate step

Warshall's Algorithm

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

$$R_{ij}^0 = A[i,j]$$
$$R_{ij}^k = R_{ij}^{k-1} \text{ or } \left(R_{ik}^{k-1} \text{ and } R_{kj}^{k-1}\right)$$

- This gives us a dynamic programming algorithm:

```
function WARSHALL(A[1..n, 1..n])
    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
```

Warshall's Algorithm

- If we allow input A to be used for the output, we can simplify things.

- Namely, if $R^{k-1}[i,k]$ (that is, $A[i,k]$) is 0, then the assignment is doing nothing. And if it is 1, and if $A[k,j]$ is also 1, then $A[i,j]$ gets set to 1. So:

```
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            if A[i, k] then
                if A[k, j] then
                    A[i, j] ← 1
```

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

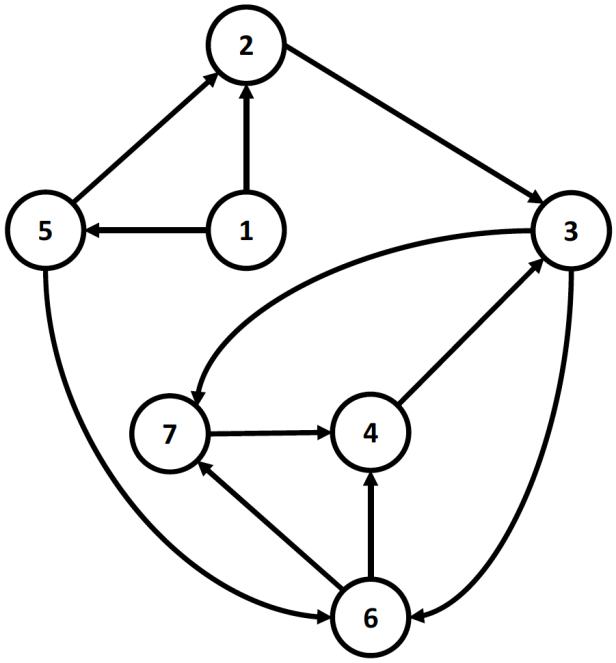
Warshall's Algorithm

- This leads to a better version of Warshall's algorithm:

```
for k ← 1 to n do
    for i ← 1 to n do
        if A[i, k] then
            for j ← 1 to n do
                if A[k, j] then
                    A[i, j] ← 1
```

- If each row in the matrix is represented as a bit-string, 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.

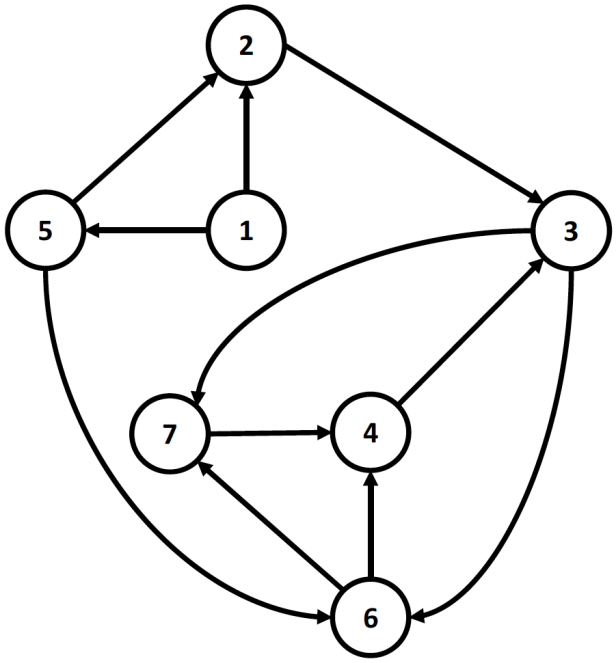
Example of Running Warshall's Algorithm



k = 1: nothing, not change

	1	2	3	4	5	6	7
1	0	1	0	0	1	0	0
2	0	0	1	0	0	0	0
3	0	0	0	0	0	1	1
4	0	0	1	0	0	0	0
5	0	1	0	0	0	1	0
6	0	0	0	1	0	0	1
7	0	0	0	1	0	0	0

Example of Running Warshall's Algorithm

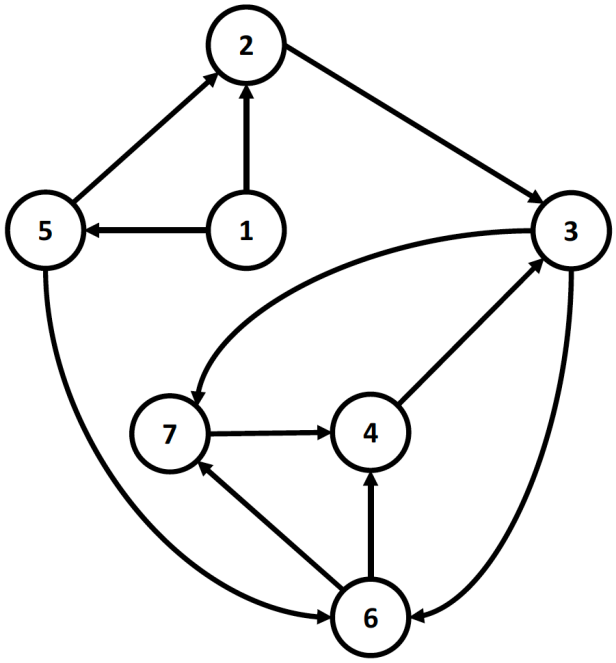


k = 2: i values A$[i,k]$: 1, 5
j values A$[k,j]$: 3

A[1,3] = 1
A[5,3] = 1

	1	2	3	4	5	6	7
1	0	1	0	0	1	0	0
2	0	0	1	0	0	0	0
3	0	0	0	0	0	1	1
4	0	0	1	0	0	0	0
5	0	1	0	0	0	1	0
6	0	0	0	1	0	0	1
7	0	0	0	1	0	0	0

Example of Running Warshall's Algorithm

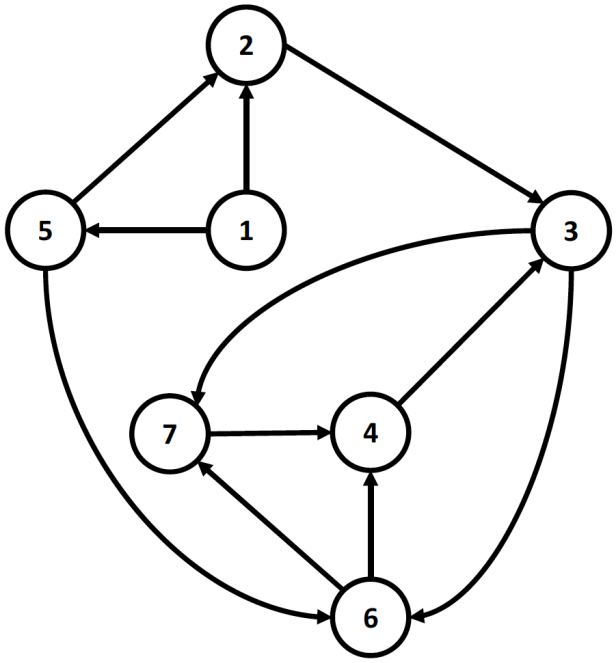


	1	2	3	4	5	6	7
1	0	1	1	0	1	0	0
2	0	0	1	0	0	0	0
3	0	0	0	0	0	1	1
4	0	0	1	0	0	0	0
5	0	1	1	0	0	1	0
6	0	0	0	1	0	0	1
7	0	0	0	1	0	0	0

k = 3: i values A$[i, k]$: 1, 2, 4, 5
j values A$[k, j]$: 6,7

$$A[1,6] = 1, A[1,7] = 1$$
$$A[2,6] = 1, A[2,7] = 1$$
$$A[4,6] = 1, A[4,7] = 1$$
$$A[5,6] = 1, A[5,7] = 1$$

Example of Running Warshall's Algorithm

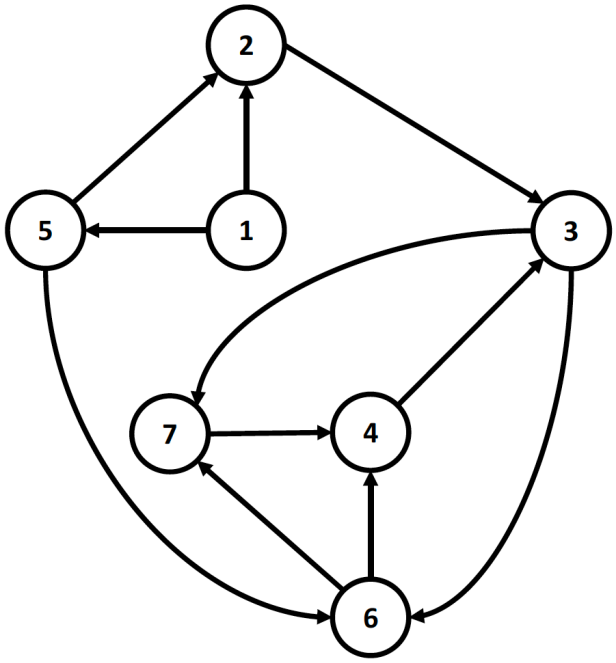


k = 4: i values A$[i,k]$: 6, 7
j values A$[k,j]$: 3,6,7

A[6,3] = 1, A[6,6] = 1, A[6,7] = 1
A[7,3] = 1, A[7,6] = 1, A[7,7] = 1

	1	2	3	4	5	6	7
1	0	1	1	0	1	1	1
2	0	0	1	0	0	1	1
3	0	0	0	0	0	1	1
4	0	0	1	0	0	1	1
5	0	1	1	0	0	1	1
6	0	0	0	1	0	0	1
7	0	0	0	1	0	0	0

Example of Running Warshall's Algorithm

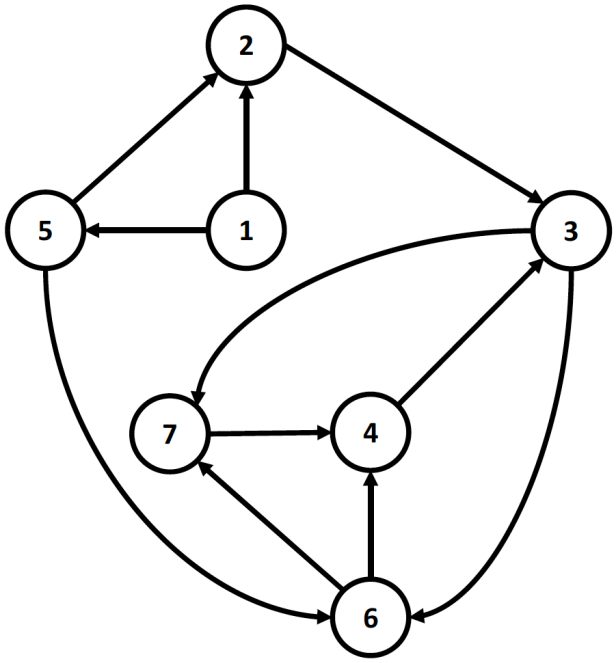


k = 5: i values A$[i,k]$: 1
j values A$[k,j]$: 2,3,6,7

A$[1,2] = 1$, A$[1,3] = 1$,
A$[1,6] = 1$, A$[1,7] = 1$

	1	2	3	4	5	6	7
1	0	1	1	0	1	1	1
2	0	0	1	0	0	1	1
3	0	0	0	0	0	1	1
4	0	0	1	0	0	1	1
5	0	1	1	0	0	1	1
6	0	0	1	1	0	1	1
7	0	0	1	1	0	1	1

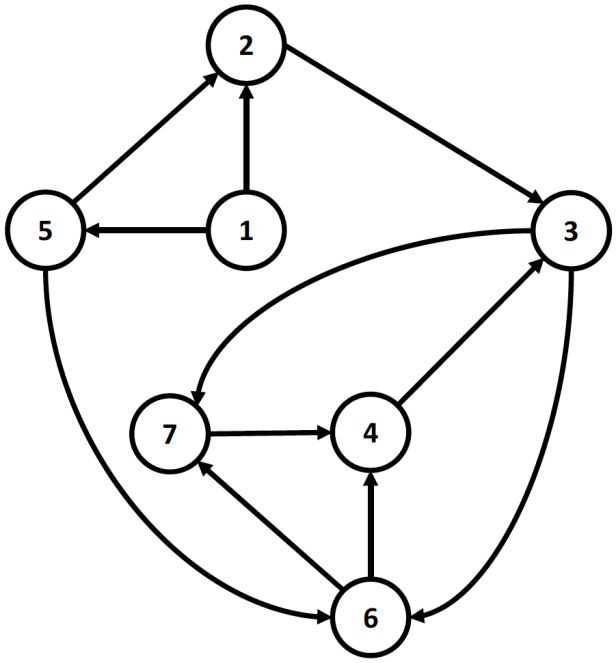
Example of Running Warshall's Algorithm



k = 6: i values A$[i,k]$: 1,2,3,4,5,6,7
j values A$[k,j]$: 3,4,6,7

	1	2	3	4	5	6	7
1	0	1	1	0	1	1	1
2	0	0	1	0	0	1	1
3	0	0	0	0	0	1	1
4	0	0	1	0	0	1	1
5	0	1	1	0	0	1	1
6	0	0	1	1	0	1	1
7	0	0	1	1	0	1	1

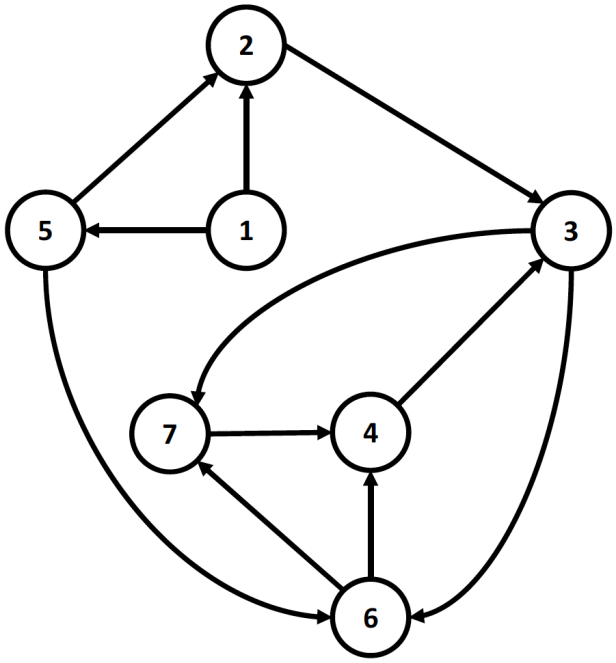
Example of Running Warshall's Algorithm



k = 7: i values A$[i,k]$: 1,2,3,4,5,6,7
j values A$[k,j]$: 3,4,6,7

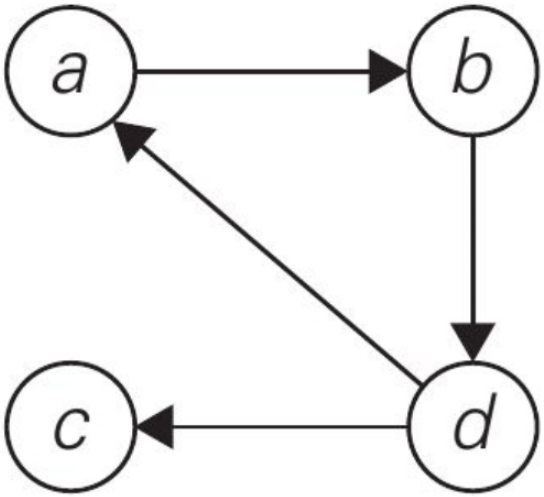
	1	2	3	4	5	6	7
1	0	1	1	1	1	1	1
2	0	0	1	1	0	1	1
3	0	0	1	1	0	1	1
4	0	0	1	1	0	1	1
5	0	1	1	1	0	1	1
6	0	0	1	1	0	1	1
7	0	0	1	1	0	1	1

Example of Running Warshall's Algorithm



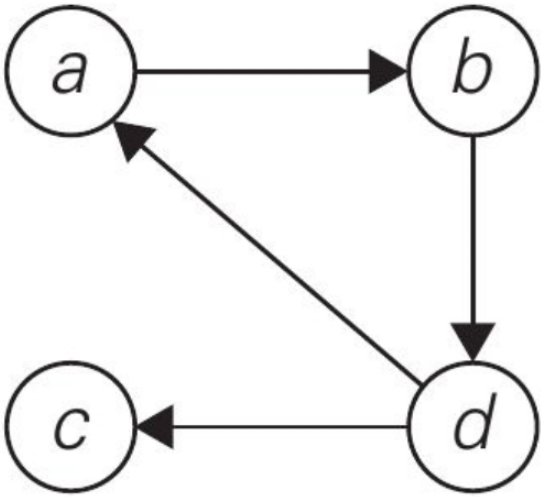
	1	2	3	4	5	6	7
1	0	1	1	1	1	1	1
2	0	0	1	1	0	1	1
3	0	0	1	1	0	1	1
4	0	0	1	1	0	1	1
5	0	1	1	1	0	1	1
6	0	0	1	1	0	1	1
7	0	0	1	1	0	1	1

Example of Running Warshall's Algorithm



$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Example of Running Warshall's Algorithm



$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \Rightarrow \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Analysis of Warshall's Algorithm

- The analysis is straightforward.
- Warshall's algorithm, as it is usually presented, is $\Theta(n^3)$, and 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.

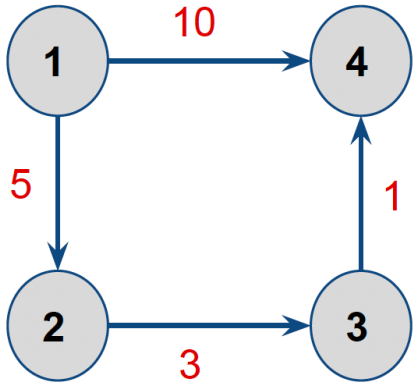
Floyd's Algorithm: All-Pairs Shortest-Paths

- 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.
- (It also works, in some circumstances, when there are non-positive weights in the graph, but not always.)
- We assume we are given a weight matrix W that holds all the edges' weights (for technical reasons, if there is no edge from node i to node j, we let $W[i,j] = \infty$).
- We will construct the distance matrix D, step by step.

Floyd's Algorithm

- We can use the same problem decomposition as we used to derive Warshall's algorithm. Again assume nodes are numbered 1 to n.

- This time ask the question: What is the shortest path from node i to node j using only nodes $\leq k$ as "stepping stones"?

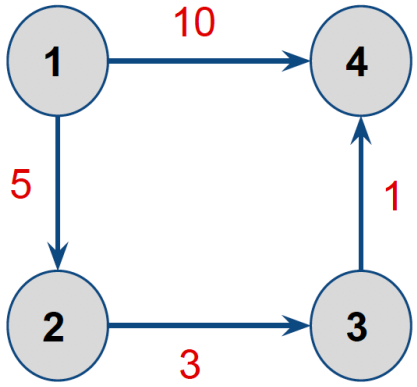


$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & \infty & 10 \\ \infty & 0 & 3 & \infty \\ \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Floyd's Algorithm

- We can use the same problem decomposition as we used to derive Warshall's algorithm. Again assume nodes are numbered 1 to n.

- This time ask the question: What is the shortest path from node i to node j using only nodes $\leq k$ as "stepping stones"?



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & \infty & 9 \\ \infty & 0 & 3 & \infty \\ \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Floyd's Algorithm

- We can use the same problem decomposition as we used to derive Warshall's algorithm. Again assume nodes are numbered 1 to n.

- This time ask the question: What is the shortest path from node i to node j using only nodes $\leq k$ as "stepping stones"?

- We either use node k as a stepping stone, or we avoid it. So again, we can

  step from i to j using only nodes $\leq k-1$, or

  step from i to k using only nodes $\leq k-1$, and then
  step from k to j using only nodes $\leq k-1$.

Floyd's Algorithm

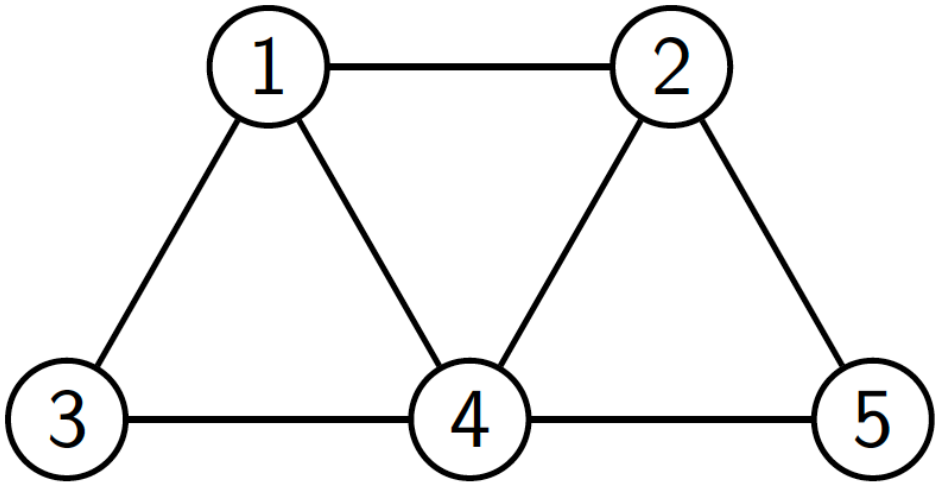
- If G's weight matrix is W then we can express the recurrence relation for minimal distances as follows:

$$D_{ij}^{0} = W[i,j]$$
$$D_{ij}^{k} = \min\left(D_{ij}^{k-1},\ D_{ik}^{k-1} + D_{kj}^{k-1}\right)$$

- And then the algorithm follows easily:

```
function FLOYD(W[1..n, 1..n])
    D ← W
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i, j] ← min(D[i, j], D[i, k] + D[k, j])
    return D
```

Example of Running Floyd's Algorithm



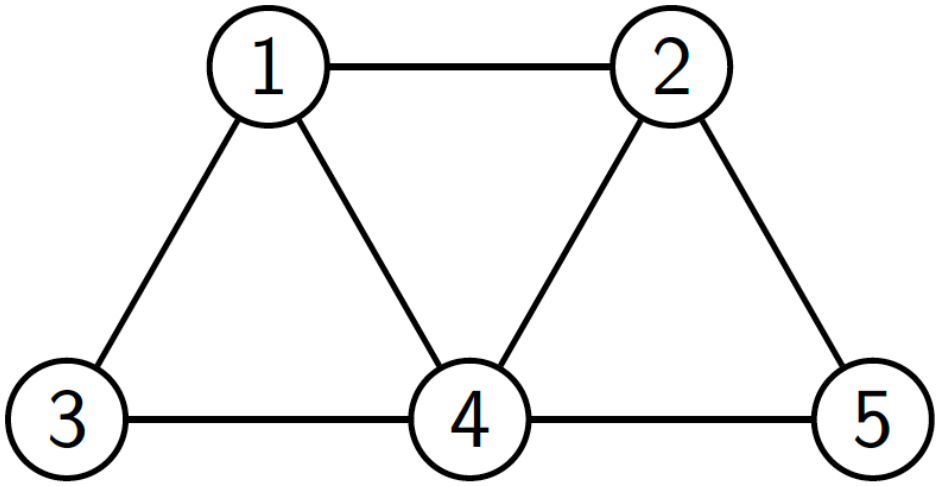
- The initial distance matrix (for the unweighted graph above).

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & \infty \\ 1 & 0 & \infty & 1 & 1 \\ 1 & \infty & 0 & 1 & \infty \\ 1 & 1 & 1 & 0 & 1 \\ \infty & 1 & \infty & 1 & 0 \end{bmatrix} \end{array}$$

k = 1: i values D$[i,k]$: 2,3,4
j values D$[k,j]$: 2,3,4

D[2,3] = 1 + 1 = 2
D[3,2] = 1 + 1 = 2

Example of Running Floyd's Algorithm

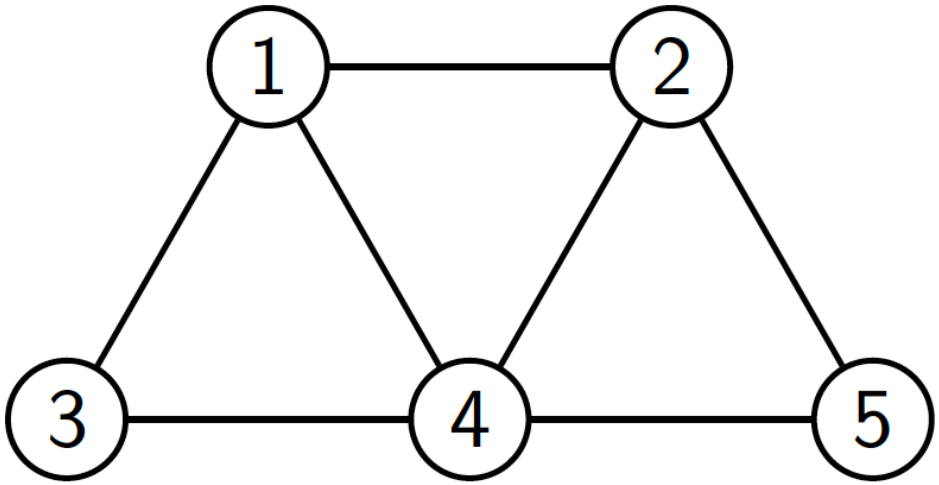


- Distance matrix after first round ($k = 1$).

k = 2: i values D$[i, k]$: 1,3,4,5
j values D$[k, j]$: 1,3,4,5
D[1,5] = 1 + 1 = 2
D[3,5] = 2 + 1 = 3
D[5,1] = 1 + 1 = 2

	1	2	3	4	5
1	0	1	1	1	∞
2	1	0	2	1	1
3	1	2	0	1	∞
4	1	1	1	0	1
5	∞	1	∞	1	0

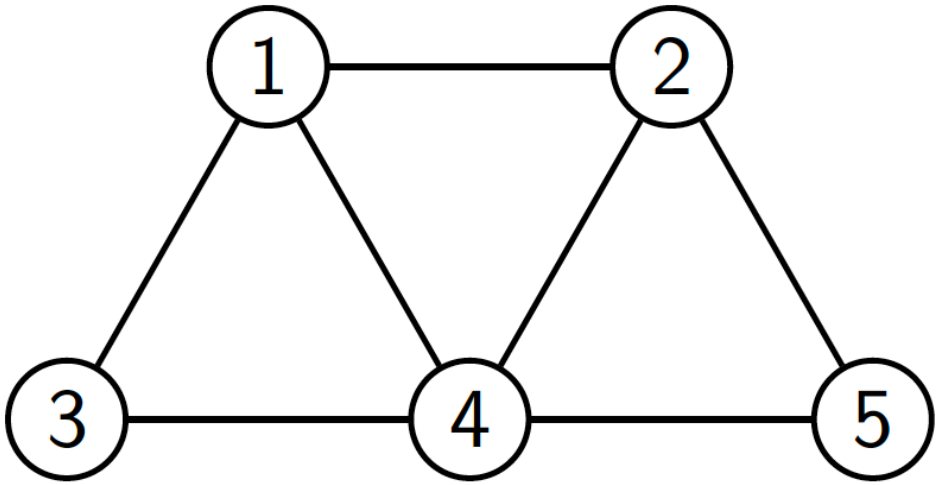
Example of Running Floyd's Algorithm



- Distance matrix after second round ($k = 2$).
- In this example, no change happens in the following round ($k = 3$).

	1	2	3	4	5
1	0	1	1	1	2
2	1	0	2	1	1
3	1	2	0	1	3
4	1	1	1	0	1
5	2	1	3	1	0

Example of Running Floyd's Algorithm

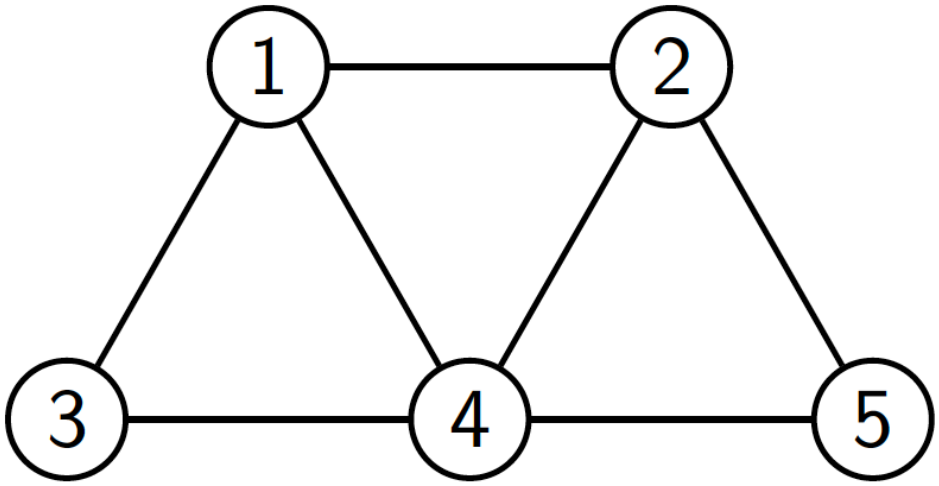


- Distance matrix after second round ($k = 2$).
- In this example, no change happens in the following round ($k = 3$).

k = 4: i values D$[i, k]$: 1,2,3,5
j values D$[k, j]$: 1,2,3,5
D[5,3] = 1 + 1 = 2

	1	2	3	4	5
1	0	1	1	1	2
2	1	0	2	1	1
3	1	2	0	1	3
4	1	1	1	0	1
5	2	1	3	1	0

Example of Running Floyd's Algorithm

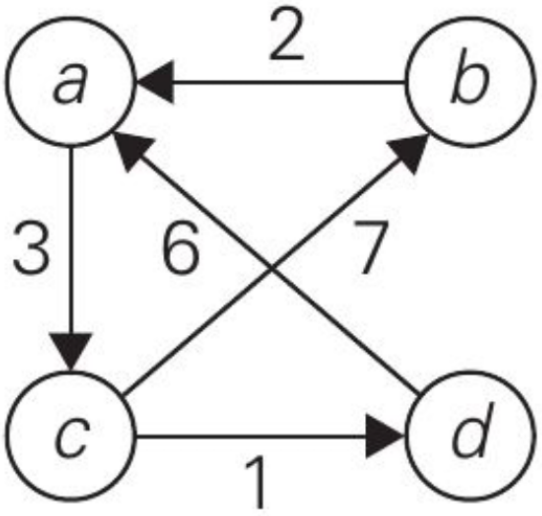


- Distance matrix after fourth round ($k = 4$).

	1	2	3	4	5
1	0	1	1	1	2
2	1	0	2	1	1
3	1	2	0	1	2
4	1	1	1	0	1
5	2	1	2	1	0

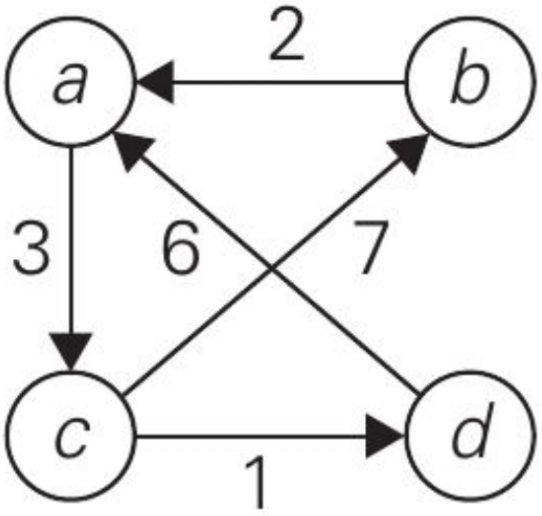
- In this example, no further change happens for $k = 5$, so this is the final result.

Example of Running Floyd's Algorithm



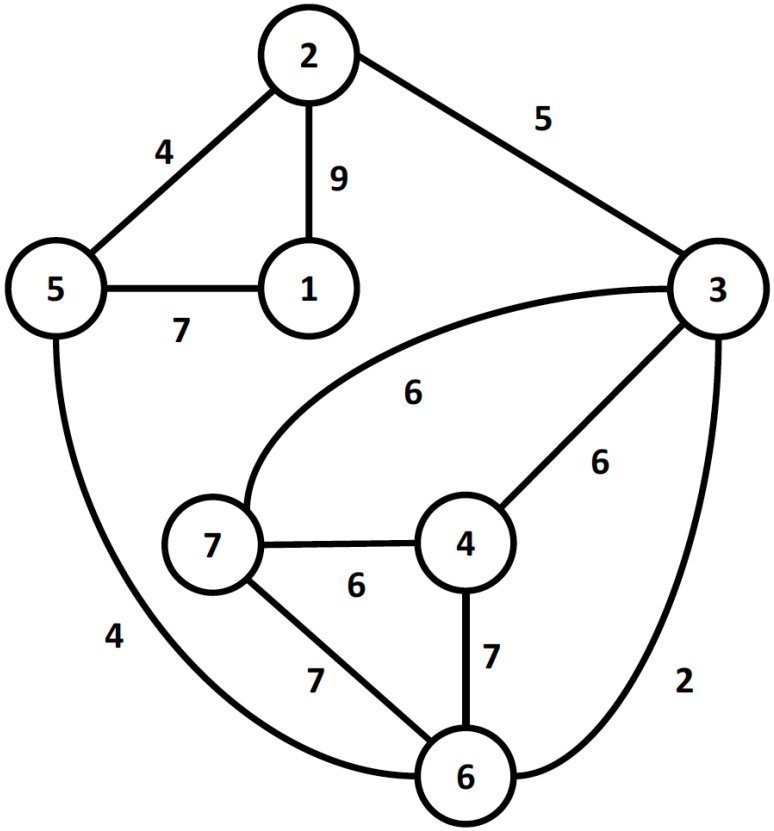
$$
\begin{array}{c} \\ a \\ b \\ c \\ d \end{array}
\begin{array}{c}
\begin{array}{cccc} a & b & c & d \end{array} \\
\begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix}
\end{array}
$$

Example of Running Floyd's Algorithm



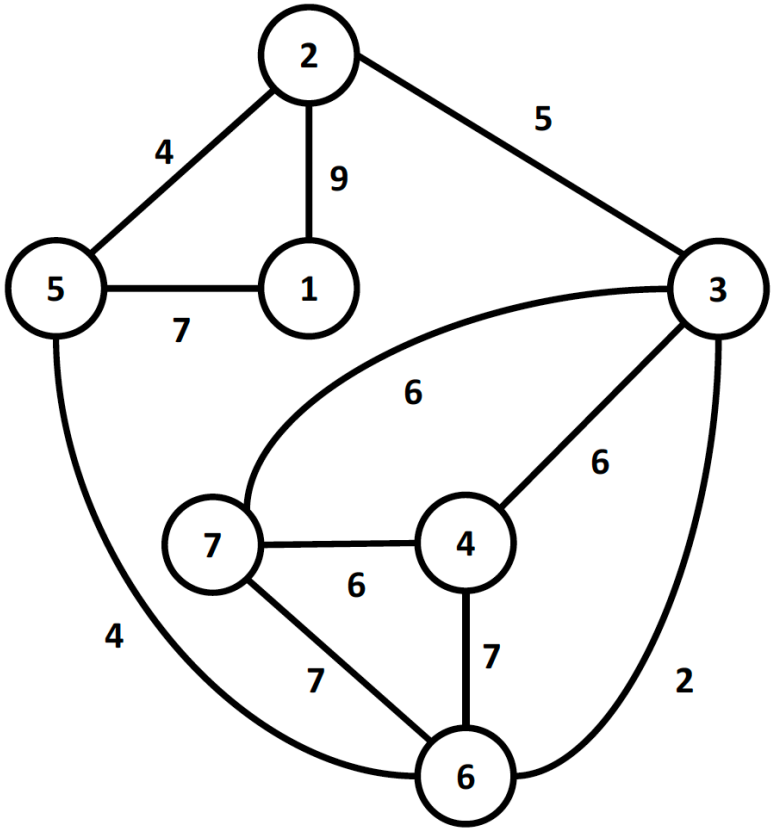
$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array} \quad \Rightarrow \quad \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

Example of Running Floyd's Algorithm



	1	2	3	4	5	6	7
1	0	9	∞	∞	7	∞	∞
2	9	0	5	∞	4	∞	∞
3	∞	5	0	6	∞	2	6
4	∞	∞	6	0	∞	7	6
5	7	4	∞	∞	0	4	∞
6	∞	∞	2	7	4	0	7
7	∞	∞	6	6	∞	7	0

Example of Running Floyd's Algorithm



	1	2	3	4	5	6	7
1	0	9	∞	∞	7	∞	∞
2	9	0	5	∞	4	∞	∞
3	∞	5	0	6	∞	2	6
4	∞	∞	6	0	∞	7	6
5	7	4	∞	∞	0	4	∞
6	∞	∞	2	7	4	0	7
7	∞	∞	6	6	∞	7	0

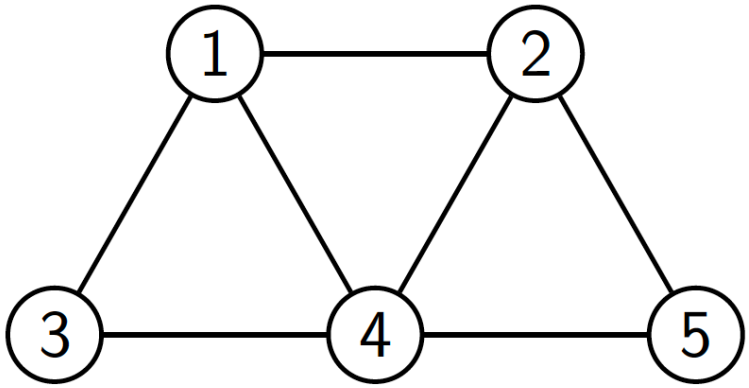
	1	2	3	4	5	6	7
1	0	9	13	18	7	11	18
2	9	0	5	11	4	7	11
3	13	5	0	6	6	2	6
4	18	11	6	0	11	7	6
5	7	4	6	11	0	4	11
6	11	7	2	7	4	0	7
7	18	11	6	6	11	7	0

A Sub-Structure Property

- For a dynamic programming approach to be applicable, the problem must have a certain "sub-structure" property that allows for a compositional solution.

- Shortest-path problems have the property: if $x_1 - x_2 - \cdots - x_i - \cdots - x_n$ is a shortest path from x_1 to x_n then $x_1 - x_2 - \cdots - x_i$ is a shortest path from x_1 to x_i.

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



Coming Up Next

- Greedy techniques
  - Prim's algorithm (Levitin Section 9.1)
  - Dijkstra's algorithm (Levitin Section 9.3).