COMP90038 Algorithms and Complexity

Dynamic Programming: Warshall and Floyd

Douglas Pires

Lecture 19

Semester 1, 2021

Dynamic Programming and Graphs

In the last lecture we looked at dynamic programming.

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 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 is important in all sorts of applications where we want to see if some "goal state" 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?

initial state goal state

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

$$\begin{aligned} R^0_{ij} &= A[i,j] \\ R^k_{ij} &= R^{k-1}_{ij} \text{ or } (R^{k-1}_{ik} \text{ and } R^{k-1}_{kj}) \end{aligned}$$

This gives us a dynamic-programming algorithm:

function WARSHALL($A[1..n, 1..n]$)
    $R^0 \leftarrow A$
    for $k \leftarrow 1$ to n **do**
        for $i \leftarrow 1$ to n **do**
            for $j \leftarrow 1$ to n **do**
                $R^k[i,j] \leftarrow R^{k-1}[i,j]$ **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

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

Example of Running Warshall's Algorithm

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

Example of Running Warshall's Algorithm

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

Example of Running Warshall's Algorithm

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

Example of Running Warshall's Algorithm

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

Example of Running Warshall's Algorithm

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

Example of Running Warshall's Algorithm

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

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"?

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

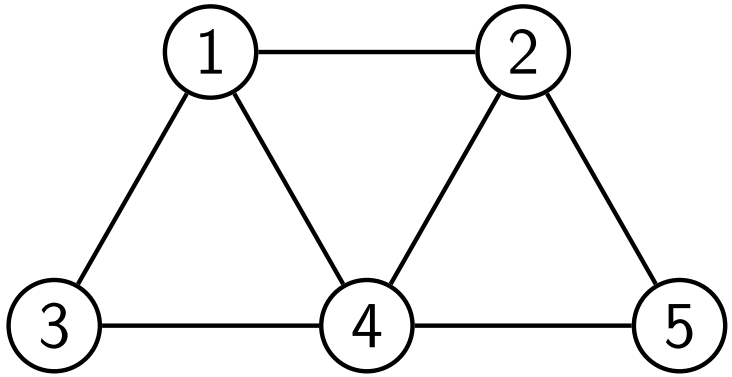
If G's weight matrix is W then we can express the recurrence relation for minimal distances as follows:

$$\begin{aligned} D^0_{ij} &= W[i,j] \\ D^k_{ij} &= min(D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}) \end{aligned}$$

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
```

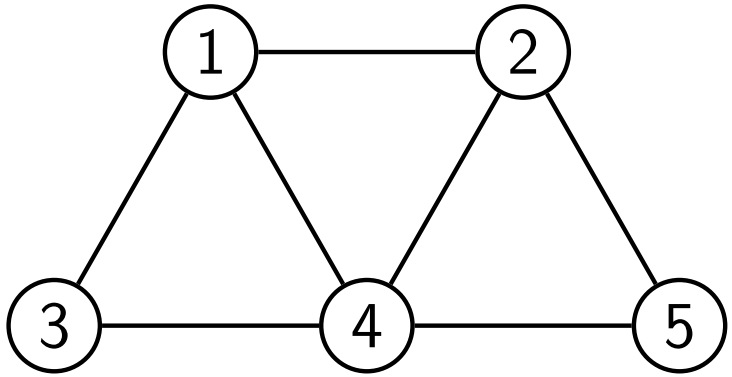
Example of Running Floyd's Algorithm



The initial distance matrix
(for the unweighted graph above)

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

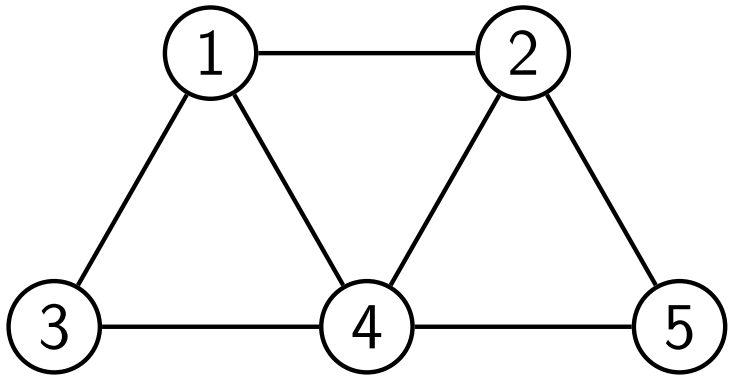
Example of Running Floyd's Algorithm



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

	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

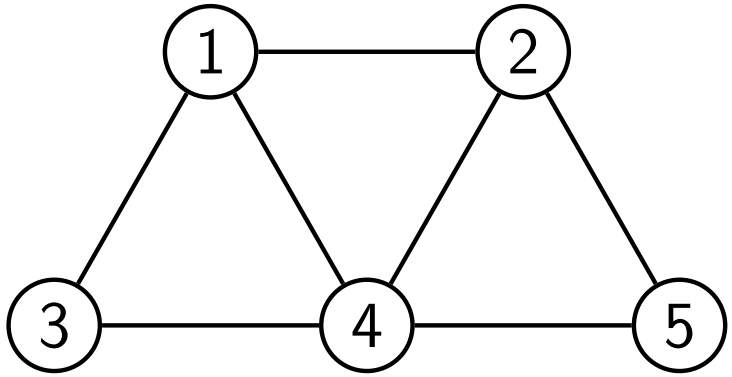


Distance matrix after second round ($k = 2$).

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 fourth round ($k = 4$).

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

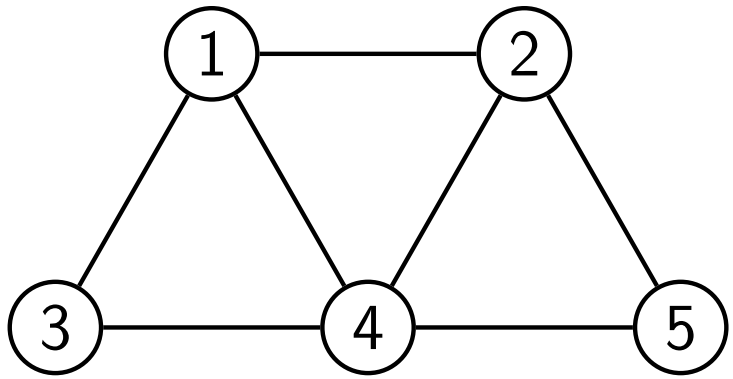
	1	2	3	4	5
1	0	1	1	1	2
2	1	0	2	1	1
3	1	2	0	1	2
4	1	1	1	0	1
5	2	1	2	1	0

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).



Is Greed Good?

Find out next week when we discuss greedy algorithms.