Complex-Network Modelling and Inference

Lecture 19: Shortest paths (Floyd-Warshall algorithm)

Matthew Roughan
<matthew.roughan@adelaide.edu.au>
https://roughan.info/notes/Network_Modelling/

School of Mathematical Sciences,
University of Adelaide

March 7, 2024

Shortest-path problems

The shortest-path problem is a VERY common problem when we work with graphs and networks (and other problems too!)

- Used in metrics: *e.g.*,
  - distance
  - betweenness
- Its important in network *routing*
  - how do your packets find the best way to their destination in the Internet?
  - how does Google maps work out your best route?
  - how do illegal wildlife traffickers work out which way to ship their goods?
- Many other practical uses
  - image segmentation
  - AI
  - solving the Rubik's Cube
  - integrated circuit layout
- Shortest paths can also be part of another algorithm

Variants

- *single-source* shortest path problem
  - implicit that we find path to all destinations
  - no point solving in single source, single destination problem
- *all-pairs* shortest path problem

And there are other generalisations that we will talk about later.

Challenge

- Exponentially many possible paths
  - we can't even hope to list them all, let alone search through all of them
- Its an Integer Linear Program
  - but we can't write down all constraints for a large problem
- We could solve by taking matrix powers, but might need to compute A^n, which is a lot of computation

But it is *NOT* NP-hard

Algorithms

There are quite a few algorithms

- Dijkstra
- Bellman-Ford (dynamic programming)
- *Floyd-Warshall*
- ...

All use the idea that a shortest path is built of of shortest path (segments), but they use this idea in different ways.

Floyd-Warshall

Solves the **all-pairs** shortest path problem

- Can cope with negative weights, but assumes no negative cycles
- The approach is to add nodes in one by one, and re-compute shortest paths at each step
  - shortest path is either the same
  - or changes to include the new node

Input

- An undirected or directed graph (N, E)
  - WLOG label the nodes $\{1, 2, \ldots, n\}$
- Link **weights** α_e, define link distances

$$d_{ij} = \begin{cases} 0 & \text{if } i = j \\ \alpha_e & \text{where } (i,j) = e \in E \\ \infty & \text{where } (i,j) = e \notin E \end{cases}$$

Recursive description

Assume we have a function

```
shortestPath(i, j, k)
```

which finds the shortest path distance from i to j using only the nodes $\{1, 2, \ldots, k\}$, where `shortestPath(i, j, 0) = d(i,j)`, the distance of the direct link if it exists and ∞ otherwise. Then Floyd-Warshall computes

```
shortestPath(i, j, k+1) = min(
                               shortestPath(i, j, k),
                               shortestPath(i, k+1, k) +
                                  shortestPath(k+1, j, k)
                             )
```

Shortest Paths

As written, the algorithm is only finding the distance – its doesn't actually tell us the path itself

- Results of algorithm must be a *sink tree*
  - a "sink" is a destination
  - we get a tree leading to the destination
  - must be a tree: can't have loops
- We can represent a tree by listing each nodes "parent"
  - here we call it a *predecessor*
  - the node immediately before it in the path
- We get one such tree per destination, so we need to store a matrix of predecessor nodes we will call V, where

$$V_{ij} = \text{ the predecessor of node } i \text{ on the path to destination } j$$

A zero will indicate we haven't found a path.

Floyd-Warshall

Let $D_{ij}^{(k)}$ denote the shortest path length from node i to node j using intermediate nodes from 1 to k only.

Initialise: $D_{ij}^{(0)} = d_{ij} \quad \forall\, i, j \in N$

$V^{(0)} = [0]$, an $|N| \times |N|$ zero matrix.

Step: for $k = 1, 2, \ldots n$, compute new distance estimates

$$D_{ij}^{(k)} = \min\{D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\} \quad \forall\ i \neq j$$

Compute the predecessor nodes

If $D_{ij}^{(k)} < D_{ij}^{(k-1)}$ then

$V_{ij}^{(k)} = k;$

else

$V_{ij}^{(k)} = V_{ij}^{(k-1)}$

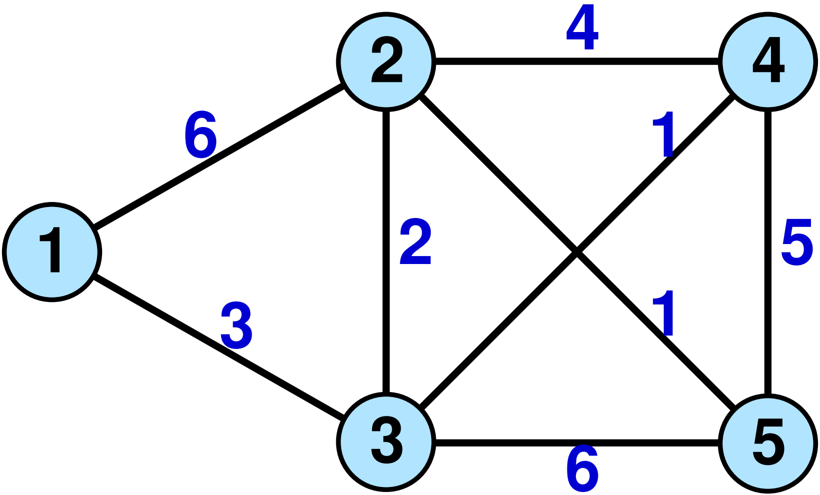
Floyd-Warshall

- The initialisation step gives the shortest path lengths subject to no intermediate nodes
- For a given k, $D_{ij}^{(k-1)}$ gives the shortest path from i to j using only nodes 1 through $k-1$ as possible intermediate nodes.
- On allowing node k as an intermediate node, either k IS on the shortest path, or it isn't.
  - **it isn't:** keep the same distance, and path
    - $D_{ij}^{(k)} = D_{ij}^{(k-1)}$ and $V_{ij}^{(k)} = V_{ij}^{(k-1)}$
  - **it is:** the new path must be made of two shortest paths, joined by node k, i.e. i–k and k–j
    - $D_{ij}^{(k)} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$
    - $V_{ij}^{(k)}$ shows where the join occurred

Floyd-Warshall

- The 0's in $V^{(n)}$ determine the adjacencies (links) in the final network.
  - $V_{ij}^{(n)}$ indicates that we never found a shorter path than d_{ij} along the direct path.
  - hence i and j are adjacent in the SPF tree
- The other terms in $V^{(n)}$ show the predecessor nodes for each end-to-end path.
  - construct paths, by concatenating predecessor nodes

Floyd-Warshall example



Floyd-Warshall example

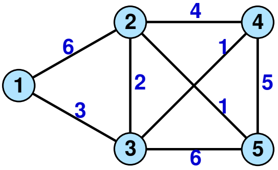
Initially, we put direct links into the matrix D

$D_{ij}^{(0)} =$

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

$V^{(0)} =$

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



Floyd-Warshall example

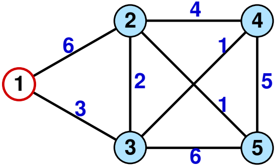
$k = 1$: include node 1 on existing direct paths (so any path already containing node 1 e.g. top line and first column of D, can be ignored). Here, nothing changes.

$D_{ij}^{(1)} =$

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

$V^{(1)} =$

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



Floyd-Warshall example

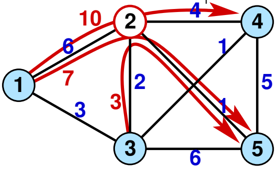
k = 2: try including node 2 on existing paths (so any path already containing node 2 e.g. line 2 and second column of D, can be ignored).

$D_{ij}^{(2)} =$

	1	2	3	4	5
1	0	6	3	10	7
2		0	2	4	1
3			0	1	3
4				0	5
5					0

$V^{(2)} =$

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



Floyd-Warshall example

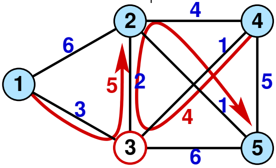
k = 3: try including node 3 on existing paths (so any path already containing node 3 e.g. line 3 and third column of D, can be ignored).

$D_{ij}^{(3)} =$

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

$V^{(3)} =$

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



E.G. The old path joining 4-5 was a direct link with distance $D_{45}^{(2)} = 5$. But when we are allowed to include node 3, we get an alternative $D_{43}^{(2)} + D_{35}^{(2)} = 4$, which is better, so we set $D_{45}^{(3)} = 4$, and $V_{45}^{(3)} = 3$.

Floyd-Warshall example

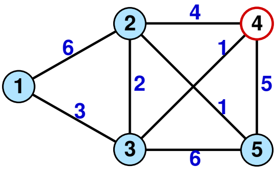
k = 4: try including node 4 on existing paths:
No changes.

$D_{ij}^{(4)} =$

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

$V^{(4)} =$

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



Floyd-Warshall example

k = 5: try including node 5 on existing paths. The entries $D_{ij}^{(5)}$ give the length of the shortest path from each node i to each other node j.

$D_{ij}^{(5)} =$

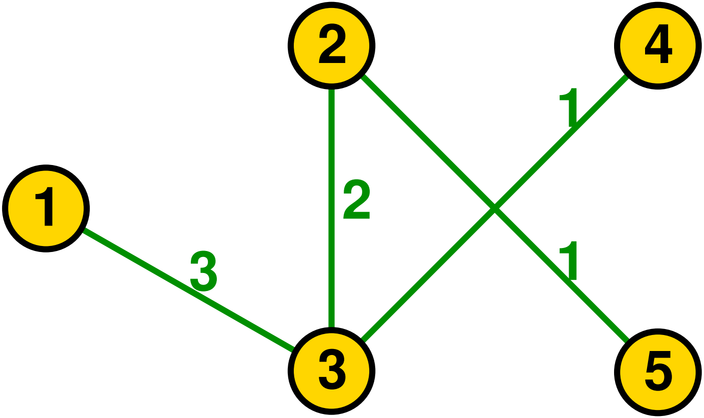
	1	2	3	4	5
1	0	5	3	4	6
2		0	2	3	1
3			0	1	3
4				0	4
5					0

$V^{(5)} =$

	1	2	3	4	5
1	0	3	[0]	3	3
2		0	[0]	3	[0]
3			0	[0]	2
4				0	3
5					0

Use the boxed zero entries in the final V to determine links: (1,3), (2,3), (2,5), (3,4).

Floyd-Warshall shortest paths



Floyd-Warshall complexity

- In calculating $D_{ij}^{(k)}$ at each step, we need to compare two possibilities for each of $\frac{|N|(|N|-1)}{2}$ pairs of nodes.
- The algorithm has $|N|$ steps
- Total computational complexity is $O(|N|^3)$.
- This is OK for a dense graph $E = O(N^2)$ but we can do much better for sparse graphs

Further reading I

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson, *Introduction to algorithms*, 2nd ed., McGraw-Hill Higher Education, 2001.