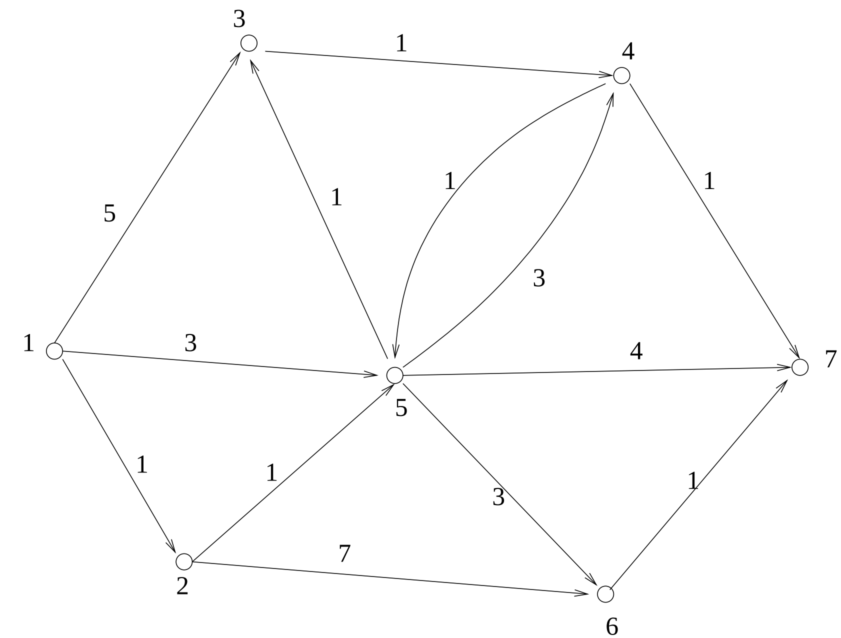
The Shortest Path Problem

Jesper Larsen & Jens Clausen

jla,jc@imm.dtu.dk

Informatics and Mathematical Modelling
Technical University of Denmark

The Shortest Path Problem



Given a directed network $\mathcal{G} = (V, E, c)$ for which the underlying undirected graph is connected. Furthermore, a source vertex r is given. Find for each $v \in V$ a dipath from r to v (if such one exists).

Mathematical Programming Formulation

- Suppose r is vertex 1. For r $n-1$ paths have to leave r. For any other vertex, the number of paths entering the vertex must be exactly 1 larger than the number of paths leaving the vertex.
- Let x_e denote the number of paths using each edge $e \in E$.

This gives the following mathematical model:

$$
\begin{aligned}
\min \quad & \sum_{e \in E} c_e x_e \\
s.t. \quad & \sum_{v \in V} x_{v1} - \sum_{v \in V} x_{1v} = -(n-1) \\
& \sum_{v \in V} x_{vu} - \sum_{v \in V} x_{uv} = 1 && u \in \{2, \ldots, n\} \\
& x_e \in \mathcal{Z}_+ && e \in E
\end{aligned}
$$

Feasible potentials

Consider an n-vector $y = y[1], \ldots, y[n]$. If y satisfies that $y[r] = 0$ and

$$\forall (v,w) \in E : y[v] + c[v,w] \geq y[w]$$

then y is called a feasible potential. If P is a path from r to $v \in V$, then if y is a feasible potential, $c(P) \geq y[v]$:

$$c(P) = \sum_{i=1}^{k} c_{e_i} \geq \sum_{i=1}^{k} (y[v_i] - y[v_{i-1}]) = y[v_k] - y[v_0] = y[v]$$

Basic algorithmic idea

- Start with the potential with $y[r] = 0$
- Check for each edge if the potential is feasible
- If YES - Stop - the potentials identify shortest paths
- If an edge (v, w) violates the feasibility condition, update $y[w]$ - this is sometimes called “correct (v, w)” or “relax (v, w)”

Ford's Shortest Path Algorithm

- Start with $p[1] = 0; y[1] = 0; y[v] = \infty; p[v] = -1$ for all other v.

  The *predecessor* vector $p[1], \ldots, p[n]$ is used for path identification.

- while an edge exists $(v, w) \in E$ such that $y[w] > y[v] + c[v, w]$: set $y[w] := y[v] + c[v, w]; p[w] := v$

Ford's Shortest Path Algorithm

Input: A distance matrix C for a digraph $G = (V, E)$ with n vertices. If the edge (i, j) belongs to E the $c(i, j)$ equals the distance from i to j, otherwise $c(i, j)$ equals ∞.

Output: Two n-vectors, y[.] og p[.], containing the length of the shortest path from 1 to i resp. the predecessor vertex for i on the path for each vertex in $\{1, ..., n\}$.

Ford's algorithm

1. Start with p[1] = 0; y[1] = 0; y[v] = ∞ ; p[v] = -1 for all other v;
2. Choose an edge (v,w) $\in$ E with y[w] $>$ y[v] + c[v,w]
   - note that no particular sequence is required ...
3. Set y[w] := y[v] + c[v,w]; p[w] := v; ##"correct(v,w)"
4. Stop when no edge (v,w) $\in$ E exists with y[w] $>$ y[v] + c[v,w].

Problem with Ford's Algorithm

Complexity ! Beware of *negative length circuits* - these may lead to an non-finite computation. Solution: Use the same sequence for the edges in each iteration.

Ford-Bellman's Shortest Path Algorithm

Input: A distance matrix C for a digraph $G = (V, E)$ with n vertices. If the edge (i, j) belongs to E the $c(i, j)$ equals the distance from i to j, otherwise $c(i, j)$ equals ∞.

Output: Two n-vectors, y[.] og p[.], containing the length of the shortest path from 1 to i resp. the predecessor vertex for i on the path for each vertex in $\{1, ..., n\}$.
The vector y[.] is called **feasible** if for any $(i, j) \in E$ it holds that y[j] $\leq$ y[i] + c[i,j].

Ford-Bellman's Algorithm

1. Start with p[1] = 0; y[1] = 0; y[v] = ∞ ; p[v] = -1 for all other v;
2. Set i := 0;
3. **while** $i < n$ **and** $\neg$(y feasible):

   i := i + 1 ;

   For (v,w) $\in$ E with y[w] $>$ y[v] + c[v,w]:

      Set y[w] := y[v] + c[v,w]; p[w] := v;
      ##"correct(v,w)"

Ford-Bellman's Algorithm with "scan"

1. Start with p[1] = 0; y[1] = 0; y[v] = ∞ ; p[v] = -1 for all other v;
2. Set i := 0;
3. **while** $i < n$ **and** $\neg$(y feasible):

   i := i + 1 ;

   for $v \in V$

   for w $\in V^+(v)$: if y[w] $>$ y[v] + c[v,w]:

   Set y[w] := y[v] + c[v,w]; p[w] := v;

Complexity of Ford-Bellman's Algorithm

Initialization: $O(n)$. Outer loop: $(n-1)$ times. In the loop: each edge is considered one time - $O(m)$. All in all: $O(nm)$.

Correctness of Ford-Bellman's Algorithm

Induction: After iteration k of the main loop, y[v] contains the length of a shortest path with *at most* k edges from 1 to v for any $v \in V$. If all distances are non-negative, a shortest path containing at most $(n-1)$ edges exists for each $v \in V$. If negative edge lengths are present, the algorithm still works. If a *negative length circuit* exists, this can be discovered by an extra iteration in the main loop. If any y[.] changes, there is such a cycle.

Shortest Path in an acyclic graph

Input: A distance matrix C for a digraph $G = (V, E)$ with n vertices. If the edge (i, j) belongs to E the $c(i, j)$ equals the distance from i to j, otherwise $c(i, j)$ equals ∞.

Output: Two n-vectors, y[.] og p[.], containing the length of the shortest path from 1 to i resp. the predecessor vertex for i on the path for each vertex in $\{1, ..., n\}$.
$V^+(v)$ denotes the edges out of v, i.e. $\{(v, w) \in E | w \in V\}$.

Shortest Path Algorithm for acyclic graphs

A **topological sorting** of the vertices is a numbering $number : V \mapsto \{1, ..., n\}$ such that for any $(v, w) \in V \quad : number(v) < number(w)$.

1. Find a **topological sorting** of $v_1, ..., v_n$.
2. Start with p[1] = 0; y[1] = 0; y[v] = ∞ ; p[v] = -1 for all other v;
3. **for** i = 1 to n-1:

   For each $w \in V^+(v_i)$ with $y[w] > y[v_i] + c[v_i, w]$:

   Set: $y[w] := y[v_i] + c[v_i, w]; p[w] := v_i$;

   ##“scan v_i”

Time Complexity of SP for acyclic graphs

Each edge is considered **only once** in the main loop due to the topological sorting. Hence, the complexity is $O(m)$.

Topological sorting

Input: An acyclic digraph $G = (V, E)$ with n vertices.

Output: A numbering number[.] of the vertices in V so for each edge $(v, w) \in E$ it holds that number[v] $<$ number[w].

Algorithm for topological sorting

1. Start with all edges unmarked; number[v] = 0 for all $v \in V$;
   i = 1;
2. **while** $i \leq n$ **do**
   - find v with all *incoming* edges marked;
   - if no such v exists : STOP;

   number[v] := i ; i := i + 1;
   - mark all $(v, w) \in E$;

   endwhile

Dijkstra's Shortest Path Algorithm

Input: A distance matrix C for a digraph $G = (V, E)$ with n vertices. If the edge (i, j) belongs to E the $c(i, j)$ equals the distance from i to j, otherwise $c(i, j)$ equals ∞.

Output: Two n-vectors, y[.] og p[.], containing the length of the shortest path from 1 to i resp. the predecessor vertex for i on the path for each vertex in $\{1, ..., n\}$.
P is the set, for which the shortest path is **already** found.

1. Start with S = {r}; p[r] = 0, y[r] = 0;
   p[v] = -1, y[v] = ∞ for all other v;
   $P = \emptyset$;
2. Select a v $\in$ S such that y[v] is minimal;
   For $\{w|(v,w) \in E\}$ – P with y[w] $>$ y[v] + c[v,w]
   set: y[w] := y[v] + c[v,w]; p[w] := v; S := S $\cup$ {w};
   When all vertices in $\{w|(v,w) \in E\}$ – P has been examined:
   S := S – {v}; P:= P $\cup$ {v};
3. Stop when S is empty.

Complexity of Dijkstras Algorithm

The only difference to Prim-Dijkstra's algorithm for Minimum Spanning Trees is the update step in the inner loop, and this step takes - like in the MST algorithm - $O(1)$. Hence the complexity of the algorithm is $O(n^2)$ if a list representation of the y[.]'s is used, and a complexity of $O(mlogn)$ can be obtained if the heap data structure is used for the representation of y[.]'s.

Correctness of Dijkstras Algorithm

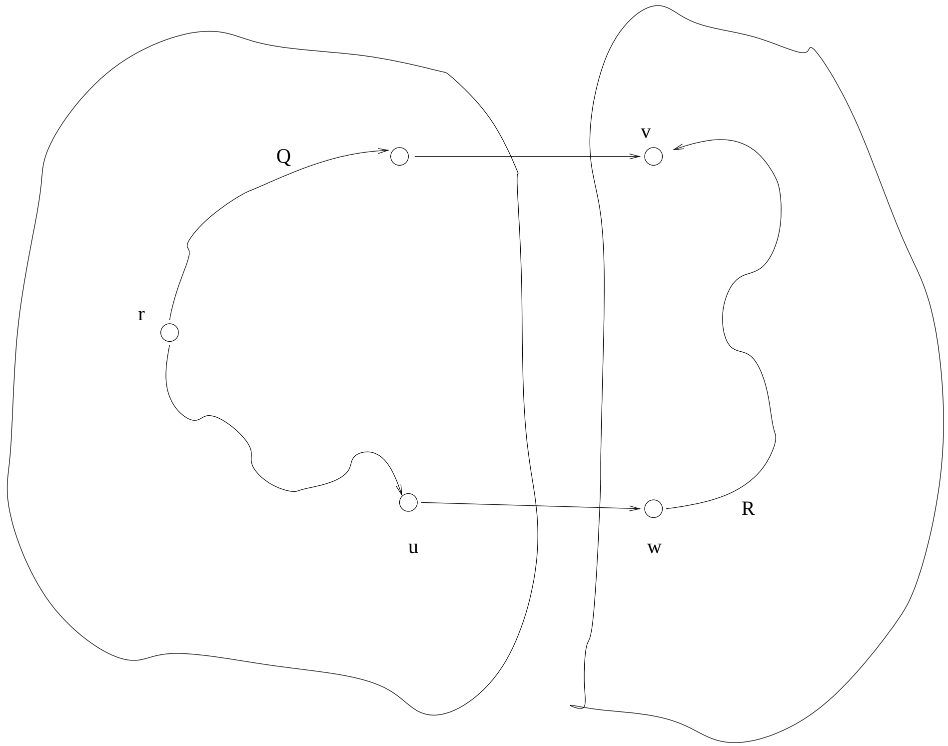
This proof is made by induction:

Suppose that before an operation it holds that 1) for each vertex u in P, the shortest path from r has been found and is of length y[u], and 2) for each vertex u not in P, y[u] is the shortest path from from r to u with all vertices except u belonging to P. This is obviously true initially.

Correctness of Dijkstras Algorithm II

Let v be the element with least y[.] value picked initially in the inner loop of iteration k. y[v] is the length of a path Q from r to v passing only through vertices in P. Suppose that this is not the shortest path from r to v - then another path R from r to v is shorter.

Correctness of Dijkstras Algorithm III



$$y[w] >= y[v](= \text{length}(Q)) \Rightarrow \text{length}(R) \geq \text{length}(Q)$$

Correctness of Dijkstras Algorithm II

R starts in r, which is in P. Since v is not in P, R has an edge from a vertex in P to a vertex not in P. Let (u, w) be the first edge of this type. w is a candidate for vertex choice is the current iteration, where v is picked. Hence y[w] $\geq$ y[v]. If all edge lengths are non-negative, the length of the part of R from w to v is non-negative, and hence the total length of R is at least the length of Q.

Correctness of Dijkstras Algorithm III

This is a contradiction – hence Q is a shortest path from r to v. Furthermore, the update step in the inner loop ensures that after the current iteration it again holds for u not in P (which is now the “old” P augmented with v) that y[u] is the shortest path from from r to u with all vertices except u belonging to P.

Floyd-Warshall's all-to-all Algorithm

Input: A distance matrix C for a digraph $G = (V, E)$ with n vertices. If the edge (i, j) belongs to E the $c(i, j)$ equals the distance from i to j, otherwise $c(i, j)$ equals ∞. $c(i, i)$ equals 0 for all i.

Output: Two $n \times n$-vectors, y[.,.] and p[.,.], containing the length of the shortest path from i to j resp. the predecessor vertex for j on the shortest path for all pairs of vertices in $\{1, ..., n\} \times \{1, ..., n\}$.

Floyd-Warshalls Algorithm

1. Start with y[i,j] = $c(i,j)$, p[i,j] = i for all (i,j) with $c(i,j) \neq \infty$, p[i,j] = 0 otherwise.
2. **for** k = 1 **to** n **do**

   for i = 1 **to** n **do**

       for j = 1 **to** n **do**

         if i $\neq$ k $\wedge$ j $\neq$ k $\wedge$ y[i,j] $>$ y[i,k] + y[k,j] **then**

         y[i,j] = y[i,k] + y[k,j]; p[i,j] := p[k,j];

       enddo

   enddo

   enddo

Complexity of Floyd-Warshall's Algorithm

In addition to the initialisation, which takes O(n^2), the algorithm has three nested loops each of which is performed n times. The overall complexity is hence O(n^3).

Correctness of Floyd-Warshall's Algorithm

This proof is made by induction:

Suppose that prior to iteration k it holds that for $i, j \in v$ y[i,j] contains length of the shortest path Q from i to j in G containing only vertices in the vertex set $\{1, ..., k-1\}$, and that p[i,j] contains the immediate predecesor of j on Q. This is obviously true after the initialisation.

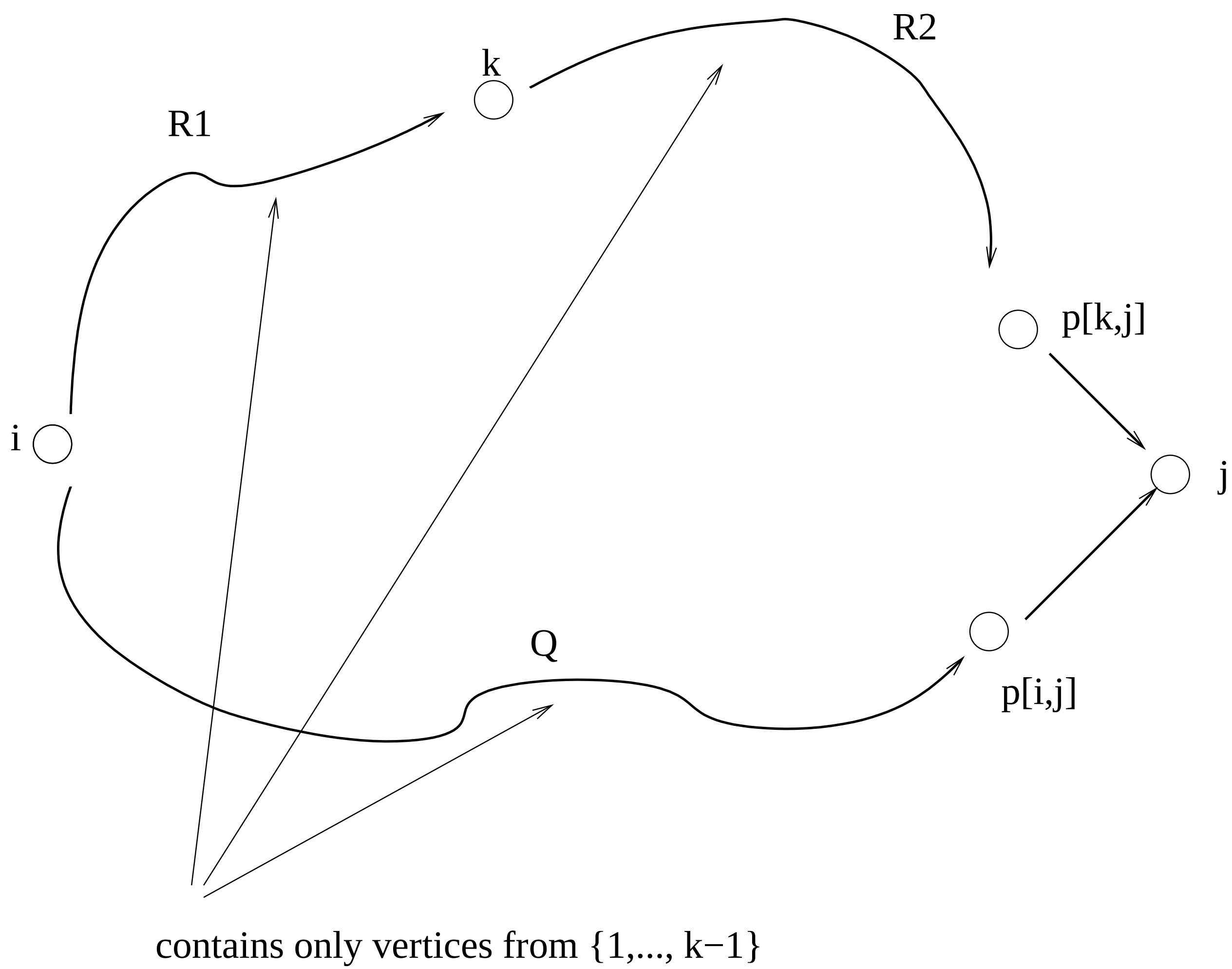
Correctness of Floyd-Warshall's Algorithm II

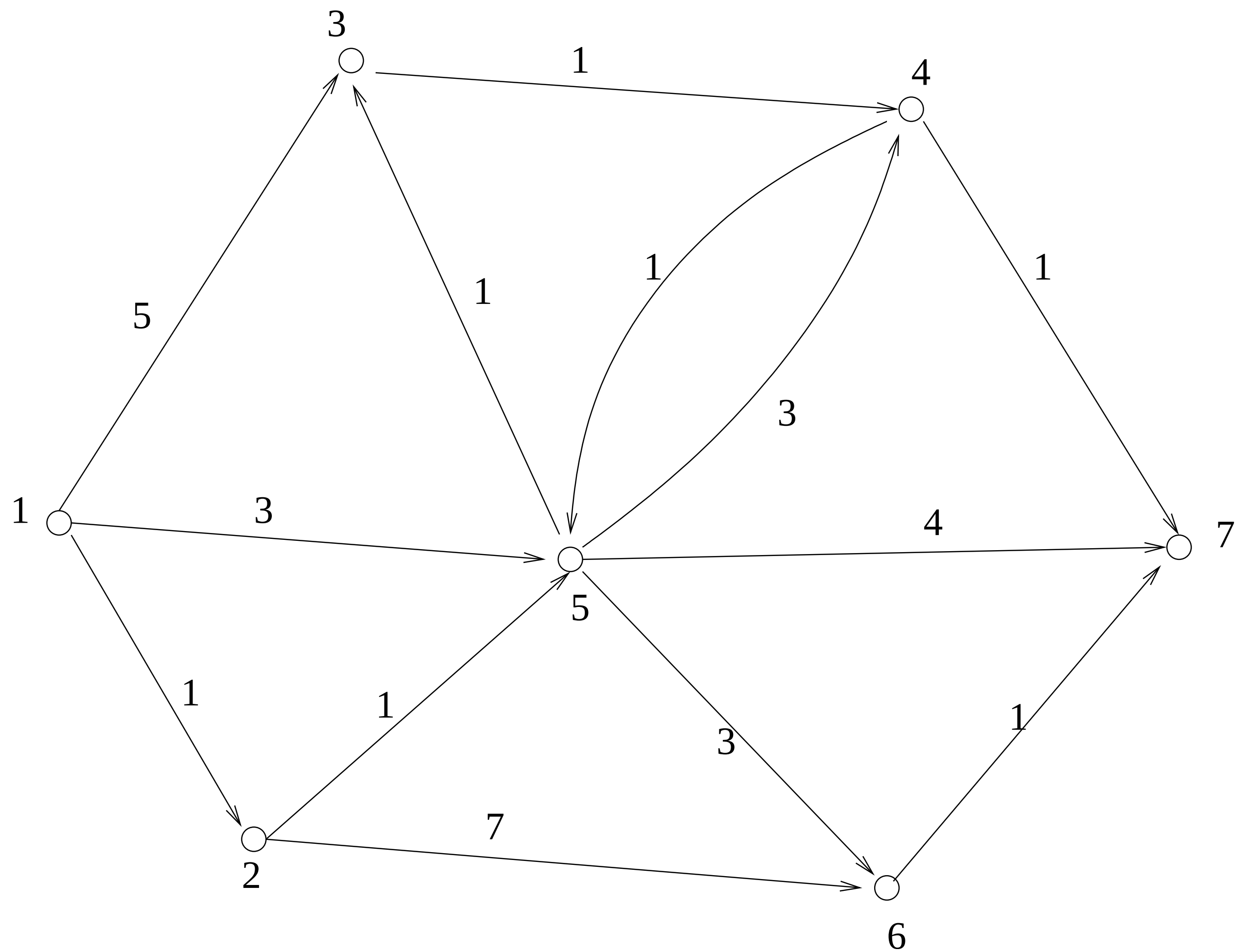
In iteration k, the length of Q is compared to the length of a path R composed of two subpaths, $R1$ and $R2$. $R1$ is an i,k path with "intermediate vertices" only in $\{1,...,k-1\}$, and $R1$ is a k,j path with "intermediate vertices" only in $\{1,...,k-1\}$. The shorter of these two is chosen.

Correctness of Floyd-Warshall's Algorithm II

The shortest path from i to j in G containing only vertices in the vertex set $\{1, ..., k\}$ either a) does not contain k - and hence is the one found in iteration $k-1$ - or b) contains k - and then can be decomposed into an i, k followed by a k, j path, each of which has been found in iteration $k-1$. Hence the update ensures the correctness of the induction hypothesis after iteration k.

Correctness of Floyd-Warshall's Algorithm IV




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