Shortest Path problems

A. Agnetis*

1 Basic properties

Consider a directed network $G = (N, A)$ having $|N| = n$ nodes and $|A| = m$ arcs, in which each arc $(i, j) \in A$ has weight c_{ij} (called *length*). Given two nodes $s \in N$ (source) and $t \in N$ (sink), let the weight of a path be defined as the sum of the arcs' lengths. Then, the *shortest path problem* consists in finding a path from s to t having minimum total length. Recalling that $\delta^-(i)$ and $\delta^+(i)$ denote the sets of arcs entering and outgoing node i respectively, the problem can be formulated as an ILP:

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{1}$$

$$\sum_{(s,j)\in\delta^+(s)} x_{sj} = 1 \tag{2}$$

$$-\sum_{(i,t)\in\delta^-(t)} x_{it} = -1 \tag{3}$$

$$\sum_{(h,j)\in\delta^+(h)} x_{hj} - \sum_{(i,h)\in\delta^-(h)} x_{ih} = 0, \quad h \in N \setminus \{s, t\} \tag{4}$$

$$x_{ij} \in \{0, 1\} \quad (i, j) \in A \tag{5}$$

where $x_{ij} = 1$ if arc (i, j) belongs to the path and $x_{ij} = 0$ otherwise.

Note that any vector $x \in I\!R^m$ corresponding to a simple path from s to t clearly that satisfies all constraints (2)–(5). However, the constraints may be satisfied also by vectors x that do not represent a simple path from s to t. In particular, (2)–(5) are satisfied also by vectors corresponding to arc sets including a simple path *plus* some cycles. However, *as long as in the network there are no cycles of negative length*, we are ensured that the optimal solution will only include a simple path from s to t. In fact, if this were not the case, one could still improve the solution by removing the cycles from it.

Observe that the coefficient matrix of (2)–(5) is the node-arc incidence matrix of G, and hence it is totally unimodular. As a consequence, any basic feasible solution (and

*Dipartimento di Ingegneria dell'Informazione e Scienze Matematiche - Università di Siena

hence also the optimal solution) is integer, so one can relax the integrality constraints and replace (5) by:

$$x_{ij} \geq 0 \quad (i,j) \in A.$$

This means that the problem can be solved as an LP. However, more efficient methods have been devised.

The above discussion shows that the shortest path problem is certainly easy when there are no negative cycles. One might therefore wonder what happens when such cycles do exist. In this case, the problem is not easy at all, but rather it belongs to the class of NP-hard combinatorial problems, and solution approaches are very different from those for ordinary shortest path problems.

2 Acyclic graphs

If graph $G(N, A)$ is acyclic, an optimal path can be found very efficiently. First of all, we introduce a simple but fundamental property of acyclic graphs. Given an acyclic graph G, a *topological numbering* of the nodes is an application $f : N \to \{1, 2, \ldots, n\}$ such that for each $(i, j) \in A$, $f(j) > f(i)$. In practice, f determines an ordering of the nodes so that each arc points from a lower-numbered to a higher-numbered node. The following property can be easily proved.

PROPOSITION 1 *A graph has a topological numbering if and only if it is acyclic.*

A topological numbering can be easily computed as follows. First of all, it is easy to prove that if a graph is acyclic, there exists at least one node i having no entering arcs, i.e., $\delta^-(i) = 0$. So, we pick one such node i and let $f(i) = 1$. Then we remove i and all incident arcs. We still have an acyclic graph, with at least one node having no entering arcs. We choose one such node j and let $f(j) = 2$, and so on.

The existence of the topological numbering allows solving the shortest path problem very efficiently. In fact, let us consider a topological numbering of the nodes (so we denote the nodes with their value $f()$), and let $F(k)$ denote the length of the shortest path from node 1 to node k. Considering the node that precedes k in such shortest path, it can only be one of the nodes in $\{1, \ldots, k-1\}$. Hence, $F(k)$ can be computed as:

$$F(k) = \min_{1 \leq i \leq k-1} \{F(i) + c_{ik}\}, \tag{6}$$

where we assume $c_{ik} = +\infty$ if $(i, k) \notin A$. Once $F(k)$ is computed, one can compute $F(k+1)$, and so on. So, in the acyclic case, we can establish the length of the shortest paths from 1 to n, following the topological ordering.

An approach using a recursive formula such as the (6) is called *dynamic programming*. Notice that the complexity of the algorithm is very low, since we consider the arcs of the network exactly once throughout the algorithm. Hence, if a graph is acyclic, the shortest paths from 1 to all the other nodes can be found in $O(m)$.

3 Dijkstra's algorithm

Dijkstra's algorithm allows one to efficiently find the shortest paths from a node s of a general graph (hence, it may contain cycles) to all other nodes. The only restriction is that the arc lengths are required to be nonnegative. This is a more restrictive assumption than the mere absence of negative cycles, but in several real-life applications such assumption is satisfied.

The key result on which Dijkstra's algorithm is based is the following.

THEOREM 1 *Let $S \subset N$ be a subset of nodes such that, for each $i \in S$, the shortest path from s to i is known, and denote as $F(i)$ its length. Let (v, h) be defined so that*

$$F(v) + c_{vh} = \min_{(i,j)\in\delta^+(S)} \{F(i) + c_{ij}\}. \tag{7}$$

Then, if $c_{ij} \geq 0$ for all $(i, j) \in A$, $F(v) + c_{vh}$ is the length of the shortest path from 1 to h.

Proof. $F(v) + c_{vh}$ represents the length of a path P from s to h which consists of two parts, namely a path from s to v, and the arc (v, h). Now consider any other path P' from s to h. Such path will sooner or later pass through the set $\delta^+(S)$, let (i, j) be the arc of $\delta^+(S)$ belonging to P'. Let us therefore consider P' as consisting of three parts, namely (i) a path P'_1 from s to i (completely inside S), (ii) arc (i, j), and (iii) a path P'_2 from j to h, which may include any node. Now, from (7) one has that the length of P'_1 plus arc (i, j) is already at least as large as $F(v) + c_{vh}$. Since $c_{ij} \geq 0$ for all $(i, j) \in A$, P'_2 can only add to path length, so $F(v) + c_{vh}$ is indeed the shortest distance to reach h from s. □

This results suggests a simple solution algorithm. The algorithm builds set S starting from node s, and adding at each step the node having shortest distance from s, i.e., the node h verifying (7).

Notice that vector $pred(i)$ allows reconstructing all shortest paths (in an arborescence structure).

The difference between the algorithm for acyclic graphs and Dijkstra's mainly lies in the fact that while at each step of the former, we know in advance which is the next node

Algorithm 1 Dijkstra's algorithm.

```
1: S := {s};
2: F(s) := 0;
3: for j ∈ N, j ≠ s do
4:    F(j) := +∞;
5: end for
6: for (s, j) ∈ δ+(s) do
7:    F(j) := c_sj
8: end for
9: while |S| ≠ n do
10:   i := arg min{F(j)|j ∈ N};
11:   S := S ∪ {i};
12:   for (i, j) ∈ δ+(i) do
13:      if F(j) > F(i) + c_ij then
14:         F(j) := F(i) + c_ij;
15:         pred(j) := i;
16:      end if
17:   end for
18: end while
```

for which $F()$ is going to be computed (it is node k if we are at step k), in Dijkstra's algorithm this is not known a priori, but rather is a result of (7).

To implement Dijkstra's algorithm, one can keep updated a *label* $F(i)$ associated with each node i, representing the length of the shortest path *so far* from s to i. Such temporary labels are initialized to $+\infty$, and, at each step (i.e., at each execution of the While cycle), they are updated as follows. If node h has just entered S, we update the temporary labels of all successors of h, i.e., all nodes $j \in \delta^+(h)$, as

$$F(j) := \min\{F(j), F(h) + c_{hj}\}$$

and the lowest label becomes *permanent*, i.e., if i is the node having minimum $F(j)$, node i enters S.

Figures 1–8 show the application of Dijkstra's algorithm in a sample problem.

Let us now consider the complexity of Dijkstra's algorithm. When we add a new node, we update the labels of all its successors, which requires $O(n)$ time. The selection of the minimum label is also $O(n)$. Since the While cycle has to be executed n times, the resulting complexity is $O(n^2)$.

Such complexity can be changed if we keep the list of temporary labels ordered. In this case, an effort of $\log n$ is required at each label update, but the lowest label can be found in constant time. Since each arc is considered exactly once by the algorithm, the complexity is then $O(m \log n)$.

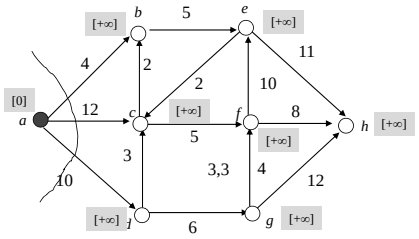

Figure 1: Example of Dijkstra's algorithm. $S = \{a\}$.

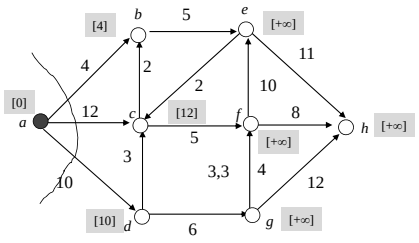

Figure 2: Label update after step 1. $S = \{a\}$.

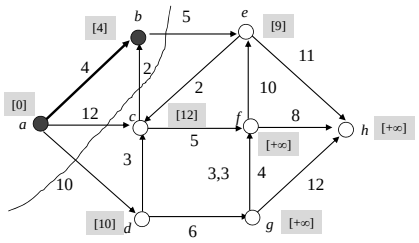

Figure 3: After step 2 and label update. $S = \{a, b\}$.

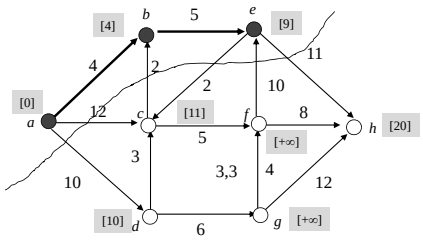

Figure 4: After step 3 and label update. $S = \{a, b, e\}$.

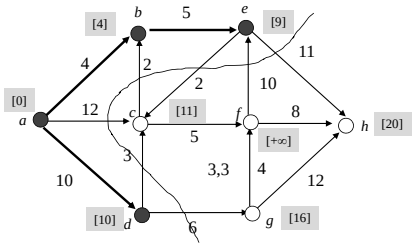

Figure 5: After step 4 and label update. $S = \{a, b, e, d\}$.

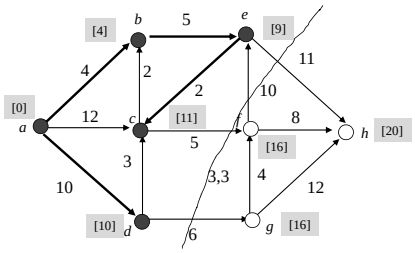

Figure 6: After step 5 and label update. $S = \{a, b, e, d, c\}$.

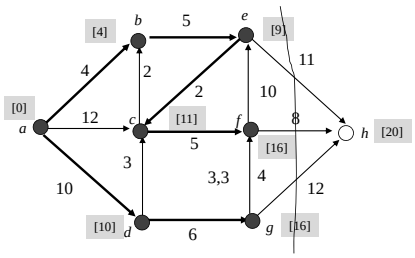

Figure 7: After steps 6 and 7 and label update. $S = \{a, b, e, d, c, f, g\}$.

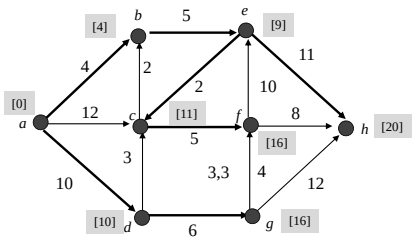

Figure 8: Dijkstra's algorithm after termination. Labels indicate the shortest distance from a to each other node.

4 Floyd-Warshall's algorithm

We already recalled that formulation (1)–(5) solves the problem as long as there are no negative cycles. In fact, if negative cycles exist, the concept of shortest path becomes controversial, since it would be convenient to pass through one such cycle an unlimited number of times. One can then formulate the shortest path problem as that of finding the shortest *simple* path (i.e., a path which does not pass twice through the same node) from s to t, but this is by no means an easy problem.

However, in general there *can* be negative cycles, and the problem arises of detecting their existence. Floyd-Warshall's algorithm is a simple, though effective algorithm that allows to:

- detect a negative cycle, if it exists
- compute the shortest path from i to j, for all node pairs i, j, if no negative cycles exist.

This algorithm computes in parallel the shortest path for all node pairs, and stops either when all shortest paths are computed or a negative cycle is found. In what follows, C denotes the $n \times n$ adjacency matrix, where we let $c_{ii} = 0$ for $i = 1, \ldots, n$ and $c_{ij} = +\infty$ if $(i, j) \notin A$. Also, recall that, given a path from i to j, we call *internal nodes* of the path all the nodes of the path except i and j. For the sake of clarity, we first illustrate the algorithm supposing that no negative cycle exists.

The central concept in Floyd-Warshall's algorithm is the following. Suppose that the nodes are arbitrarily numbered as 1, 2, ..., n. We let $\pi_{ij}^{(k)}$ denote the shortest path from i to j, *such that its internal nodes belong to the subset* $\{1, 2 \ldots, k\}$.

The algorithm has n iterations. At each iteration k, an $n \times n$ matrix $C^{(k)}$ is computed in which, for each pair i, j, entry $c_{ij}^{(k)}$ is the length of $\pi_{ij}^{(k)}$. At the beginning of the algorithm, we initialize $C^{(0)}$ as C. We next show that matrix $C^{(k+1)}$ can be easily computed from $C^{(k)}$.

Let us consider two nodes i and j, and suppose we have computed matrix $C^{(k)}$. We want to determine the value of $c_{ij}^{(k+1)}$, i.e., the length of the shortest path $\pi_{ij}^{(k+1)}$ from i to j such that its internal nodes belong to $\{1, \ldots, k+1\}$. Actually, two cases may hold.

- Path $\pi_{ij}^{(k+1)}$ does not pass through node $k+1$. In this case, having extended the set of candidate internal nodes to $k+1$ does not produce any benefit. Hence, $\pi_{ij}^{(k+1)} \equiv \pi_{ij}^{(k)}$ and therefore
$$c_{ij}^{(k+1)} = c_{ij}^{(k)}.$$

- Path $\pi_{ij}^{(k+1)}$ *does* pass through node $k+1$. Now consider the two subpaths of $\pi_{ij}^{(k+1)}$, from i to $k+1$ (say, π') and from $k+1$ to j (say, π''). The internal nodes of both π' and π'' are all contained in the subset $\{1, 2, \ldots, k\}$. Therefore, π' and π'' are indeed the shortest paths from i to $k+1$ and, respectively, from $k+1$ to j, such that their internal nodes belong to $\{1, 2, \ldots, k\}$, i.e., $\pi' \equiv \pi_{i,k+1}^{(k)}$ and $\pi'' \equiv \pi_{k+1,j}^{(k)}$. Therefore, in this case

$$c_{ij}^{(k+1)} = c_{i,k+1}^{(k)} + c_{k+1,j}^{(k)}.$$

In conclusion, the above considerations allow one to compute the generic element of $C^{(k+1)}$ as

$$c_{ij}^{(k+1)} = \min\{c_{ij}^{(k)}; c_{i,k+1}^{(k)} + c_{k+1,j}^{(k)}\}. \tag{8}$$

Hence, the algorithm proceeds by subsequently computing $C^{(1)}$,$C^{(2)}$,...,$C^{(n)}$. Clearly, the entries of $C^{(n)}$ are the lengths of the shortest paths between any two pairs of nodes.

Algorithm 2 Floyd-Warshall's algorithm.

```
1: for i = 1 to n do
2:    for j = 1 to n do
3:       p_ij := i; c_ij^(0) := c_ij (or +∞ if (i,j) does not exist)
4:    end for
5: end for
6: for h = 1 to n do
7:    for i = 1 to n do
8:       for j = 1 to n do
9:          if c_ih^(h-1) + c_hj^(h-1) < c_ij^(h-1) then
10:            c_ij^(h) := c_ih^(h-1) + c_hj^(h-1)
11:            p_ij^(h) := p_hj^(h-1)
12:         else
13:            c_ij^(h) := c_ij^(h-1)
14:            p_ij^(h) := p_ij^(h-1)
15:         end if
16:         for i = 1 to n do
17:            if c_ii^(h) < 0 then
18:               STOP (there are negative cycles)
19:            end if
20:         end for
21:      end for
22:   end for
23: end for
```

A few comments are in order.

- *Complexity.* The computational complexity of Floyd-Warshall's algorithm can be easily computed. In fact, for each value $c_{ij}^{(k)}$ can be computed in constant time, being the minimum between two quantities. Since i, j and k all span from 1 to n, the overall complexity is $O(n^3)$.

- *Path reconstruction.* Besides computing the values of the shortest paths, one must also be able to reconstruct such paths, for each node pair i, j. Floyd and Warshall proposed a very compact and clever way of storing all the information to reconstruct the path. In order to be able to backtrack each path, all we need to store, for each shortest path from i to j, is the last internal node, i.e., the predecessor of node j. In fact, if the last internal node of the shortest path from i to j is, say, node h, we can continue backtracking the path finding the last internal node of the shortest path from i to h... and so on. To this aim, while computing the matrix $C^{(k)}$, we also construct the *predecessor matrix* $P^{(k)}$, in which the entry $p_{ij}^{(k)}$ is defined as the last internal node of the path $\pi_{ij}^{(k)}$, i.e., the predecessor of j. (We suppose that $p_{ij}^{(k)}$ is undefined if $c_{ij}^{(k)} = +\infty$.) Actually, $p_{ij}^{(k)}$ can be computed very easily. In fact, with reference to the two cases considered in (8), one has that if $\pi_{ij}^{(k+1)} \equiv \pi_{ij}^{(k)}$, then of course $p_{ij}^{(k+1)} = p_{ij}^{(k)}$, while in the other case, being $\pi_{ij}^{(k+1)}$ the concatenation of $\pi_{i,k+1}^{(k)}$ and $\pi_{k+1,j}^{(k)}$, one has $p_{ij}^{(k+1)} = p_{k+1,j}^{(k)}$. The entries of $P^{(n)}$ then allow one to reconstruct all shortest paths.

  Notice that $P^{(n)}$ gives a compact representation of all shortest paths. In fact, since there are $O(n^2)$ shortest paths, listing them explicitly would require $O(n^3)$ memory space. Instead, $P^{(n)}$ contains all relevant information in $O(n^2)$.

- *Negative cycles.* From the viewpoint of computational complexity, one may observe that all shortest paths can also be computed applying n times Dijkstra's algorithm, every time from a different source node. Since the complexity of Dijkstra's algorithm is $O(n^2)$, also in this way one would get an overall complexity of $O(n^3)$. However, unlike Dijkstra's algorithm, Floyd-Warshall's allows dealing with negative cycles too. Actually, we only need apply (8) even when $i = j$. In fact, suppose that a negative cycle exists, and let i the highest-numbered node belonging to the cycle. Then, by construction, one has that $c_{ii}^{(i-1)} < 0$, indicating that there is a way to go from i to i passing only through lower-numbered nodes, such that the total length of the arcs is negative. When this occurs, of course the algorithm stops.

As an example, consider a graph having the following adjacency matrix:

$$C = \begin{pmatrix} 0 & 5 & 1 & +\infty \\ +\infty & 0 & +\infty & 3 \\ 10 & 2 & 0 & 6 \\ 4 & +\infty & +\infty & 0 \end{pmatrix}$$

applying the algorithm, one gets the following matrices, in which at each step the entries corresponding to a new path (i.e., whenever $c_{ij}^{(k+1)} = c_{i,k+1}^{(k)} + c_{k+1,j}^{(k)}$) are circled.

$$C^{(0)} = \begin{pmatrix} 0 & 5 & 1 & +\infty \\ +\infty & 0 & +\infty & 3 \\ 10 & 2 & 0 & 6 \\ 4 & +\infty & +\infty & 0 \end{pmatrix} \qquad P^{(0)} = \begin{pmatrix} 1 & 1 & 1 & - \\ - & 2 & - & 2 \\ 3 & 3 & 3 & 3 \\ 4 & - & - & 4 \end{pmatrix}$$

$$C^{(1)} = \begin{pmatrix} 0 & 5 & 1 & +\infty \\ +\infty & 0 & +\infty & 3 \\ 10 & 2 & 0 & 6 \\ 4 & \textcircled{9} & \textcircled{5} & 0 \end{pmatrix} \qquad P^{(1)} = \begin{pmatrix} 1 & 1 & 1 & - \\ - & 2 & - & 2 \\ 3 & 3 & 3 & 3 \\ 4 & \textcircled{1} & \textcircled{1} & 4 \end{pmatrix}$$

$$C^{(2)} = \begin{pmatrix} 0 & 5 & 1 & \textcircled{8} \\ +\infty & 0 & +\infty & 3 \\ 10 & 2 & 0 & \textcircled{5} \\ 4 & 9 & 5 & 0 \end{pmatrix} \qquad P^{(2)} = \begin{pmatrix} 1 & 1 & 1 & \textcircled{2} \\ - & 2 & - & 2 \\ 3 & 3 & 3 & \textcircled{2} \\ 4 & 1 & 1 & 4 \end{pmatrix}$$

$$C^{(3)} = \begin{pmatrix} 0 & \textcircled{3} & 1 & \textcircled{6} \\ +\infty & 0 & +\infty & 3 \\ 10 & 2 & 0 & 5 \\ 4 & \textcircled{7} & 5 & 0 \end{pmatrix} \qquad P^{(3)} = \begin{pmatrix} 1 & \textcircled{3} & 1 & \textcircled{2} \\ - & 2 & - & 2 \\ 3 & 3 & 3 & 2 \\ 4 & \textcircled{3} & 1 & 4 \end{pmatrix}$$

$$C^{(4)} = \begin{pmatrix} 0 & 3 & 1 & 6 \\ \textcircled{7} & 0 & \textcircled{8} & 3 \\ 10 & 2 & 0 & 5 \\ 4 & \textcircled{7} & 5 & 0 \end{pmatrix} \qquad P^{(4)} = \begin{pmatrix} 1 & 3 & 1 & 2 \\ \textcircled{4} & 2 & \textcircled{1} & 2 \\ 3 & 3 & 3 & 2 \\ 4 & \textcircled{3} & 1 & 4 \end{pmatrix}.$$

These two final matrices contain the length of all shortest paths and all the information allowing their reconstruction.