Chapter 1

Basic Graph Theory and Shortest-Paths Algorithms

This chapter gives some basic information from graph theory and graph algorithms, followed by a discussion on the shortest paths problem. Since this chapter is based on well-known, standard results, we do not give the original references, instead we use standard texts in the area.

For graph theory, we refer to Bollobas [2] and Bondy & Murty [3], and readers interested in more informatiion are refered to those texts. For the design and analysis of algorithms, we refer to the text by Cormen, Leiserson and Rivest [4].

Our discussion on the shortest paths problem is brief but self-contained. First, Bellman's inequalities, node potentials, and shortest paths arborescences (directed trees) are discussed, followed by descriptions of Dijkstra's algorithm and the Floyd-Warshall algorithm. Our treatment is based on several texts, namely, Ahuja, Magnanti & Orlin [1, Chapter 4], Cormen et al [4, Chapters 25 & 26], Lawler [8], and Larson & Odoni [6, Chapter 6.2]. Readers interested in more informatiion are referred to one of the above texts. Readers interested in a compact and accessible introduction to network algorithms are referred to the chapter by Larson & Odoni.

1.1 Graph theory terminology

A graph $G = (V(G), E(G))$ consists of a finite set $V(G)$ of *nodes* and a finite set $E(G)$ of unordered pairs of (distinct) nodes called *edges*. (For the most part, we study simple graphs having neither loops nor multiple edges, so our definition does not allow these.) Usually, there is only one graph G under consideration, and since there is no ambiguity, we use V instead of $V(G)$ and E instead of $E(G)$. The number of nodes, $|V|$, is usually denoted by n, and the number of edges, $|E|$, is usually denoted by m. For an edge $e = \{x, y\}$, nodes x and y are said to be *incident* to e, and x and y are called the *ends* of e. We also denote the edge as xy. Nodes x and y are called *adjacent* and are said to be *joined* by the edge e. Figure 1.1 shows an example of a graph; here $n = 7$ and $m = 9$.

The *degree* of a node v, denoted $\deg(v)$, is the number of edges incident to v. Here are the degrees of the nodes for the example in Figure 1.1.

node v	s	a	b	c	d	e	t
$\deg(v)$	2	4	2	3	2	3	2

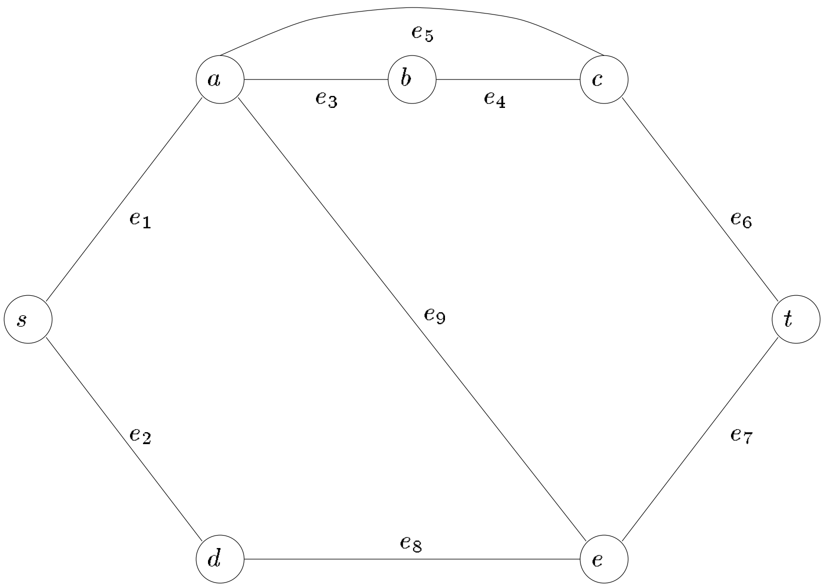

Figure 1.1: An example of a graph.

Graphs may be represented by matrices. The *adjacency* matrix $= (a_{ij})$ of a graph G has its rows and columns indexed by $V(G)$ and is defined

$$a_{ij} = \begin{cases} 1 & \text{if node } i \text{ and node } j \text{ are adjacent,} \\ 0 & \text{otherwise.} \end{cases}$$

Here is the adjacency matrix for the example in Figure 1.1.

	s	a	b	c	d	e	t
s	0	1	0	0	1	0	0
a	1	0	1	1	0	1	0
b	0	1	0	1	0	0	0
c	0	1	1	0	0	0	1
d	1	0	0	0	0	1	0
e	0	1	0	0	1	0	1
t	0	0	0	1	0	1	0

The *incidence* matrix $B = (b_{ij})$ of a graph G has its rows indexed by $V(G)$ and its columns indexed by $E(G)$ and is defined

$$b_{ij} = \begin{cases} 1 & \text{if node } i \text{ is incident to edge } j, \\ 0 & \text{otherwise.} \end{cases}$$

Here is the incidence matrix for the example in Figure 1.1.

$$\begin{array}{c|ccccccccc|}
 & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 \\
\hline
s & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
a & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
b & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
c & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
d & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
e & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
t & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
\hline
\end{array}$$

A *subgraph* $H = (V(H), E(H))$ of a graph G is a graph with node set $V(H) \subseteq V(G)$ and edge set $E(H) \subseteq E(G)$. Since H is a graph, every element of $E(H)$ is an unordered pair of elements of $V(H)$; in other words, both ends of every edge of H are nodes of H.

A subgraph H of a graph G is called *spanning* if $V(H) = V(G)$.

A subgraph H of a graph G is called an *induced* subgraph if $E(H)$ contains every edge of $E(G)$ such that both ends of the edge are in $V(H)$. For a node set $S \subseteq V(G)$, the subgraph of G *induced by* S means the induced subgraph H of G such that $V(H) = S$.

Given a graph $G = (V(G), E(G))$, a *walk* W is a sequence of nodes and edges

$$v_0 e_1 v_1 e_2 v_2 \cdots e_\ell v_\ell$$

where edge e_i, $1 \leq i \leq \ell$, is incident to nodes v_{i-1} and v_i. A walk is also denoted by its node sequence, e.g., $W = v_0 v_1 \ldots v_\ell$. The node v_0 is called the *start node* of the walk W, and v_ℓ is called the *end node* of W. Also, the walk W is said to *join* v_0 and v_ℓ, and W is called a v_0-v_ℓ walk.

A *trail* is a walk with no repeated edges.

A *path* is a walk with no repeated nodes.

The *length* of a walk (or trail or path) is defined to be the number of edges in it, e.g., the length of W is ℓ.

A *cycle* is a trail of positive length whose start node and end node are the same but all other nodes are distinct.

For example, in Figure 1.1, $s\ e_1\ a\ e_5\ c\ e_4\ b\ e_3\ a\ e_5\ c\ e_6\ t$ is a walk of length 6, $s\ e_1\ a\ e_5\ c\ e_4\ b\ e_3\ a\ e_9\ e\ e_7\ t$ is a trail of length 6, $s\ e_1\ a\ e_5\ c\ e_6\ t$ is a path of length 3, and $a\ e_5\ c\ e_4\ b\ e_3\ a$ is a cycle of length 3.

A graph is said to be *connected* if for every pair of nodes v, w there is a walk joining v and w.

A (connected) *component* of a graph is defined to be a subgraph induced by a maximal set S of nodes such that there is a walk joining every pair of nodes in S.

A graph that has no cycles is called a *forest.*

A connected graph that has no cycles is called a *tree.*

Given a node set $Q \subseteq V$, $\delta(Q)$ denotes the set of all edges with one end in Q and the other end in $V \backslash Q$. (Informally, $\delta(Q)$ is the "boundary" of the node set Q in G.) For example, in Figure 1.1, taking $Q = \{a, c, d, e\}$ we have $\delta(Q) = \{e_1, e_2, e_3, e_4, e_6, e_7\}$.

A cut consists of all edges that have one end in Q and the other end in $V \backslash Q$, where Q is a node set such that $Q \neq \emptyset$ and $Q \neq V$; this cut is denoted $(Q, V \backslash Q)$. Clearly, if $\emptyset \neq Q \neq V$, then $\delta(Q) = (Q, V \backslash Q)$. Another example for Figure 1.1 is $(\{s, a, b, c\},\ \{d, e, t\}) = \{e_2, e_6, e_9\}$.

A *network* is a graph (or directed graph), together with zero or more functions from the nodes to the real numbers, and zero or more functions from the edges to the real numbers. For the most part, we study networks consisting of a graph $G = (V(G), E(G))$ and a single real-valued function on the edges $c : E(G) \rightarrow \Re$ called the cost. The cost of an edge $e = xy$ is denoted by either $c(xy)$ or c_{xy}. For a subgraph H of G, the cost of H is defined to be the sum of the costs of all the edges in H and is denoted $c(H)$; so, $c(H) = \sum_{e \in E(H)} c(e)$.

1.2 The running time of algorithms

The number of steps executed by an algorithm is called the runing time. This is justified since each step takes roughly the same time to execute.
Big-Oh notation: Given two functions f and g from the natural numbers to the real numbers, f is said to be $O(g)$ if there exist positive numbers n_0 and k such that

$$\forall n \geq n_0 \ : \ f(n) \leq k \cdot g(n).$$

Informally, when we use this notation, we keep only the asymptotically largest terms, and then drop their coefficients.

$$\begin{aligned}
& Examples: \\
2n^3 - 5n^2 - 2n + 100 &= O(n^3). \\
10n^2 - 1000n + 10^6 n^{1/2} &= O(n^2). \\
\frac{\log n}{1000} + 10^6 &= O(\log n). \\
\frac{n^2 \log n}{1000} + 10^6 n^2 + 10^{12} n^{1.99} &= O(n^2 \lg n).
\end{aligned}$$

1.3 Shortest Paths Problems

Consider a network consisting of a graph $G = (V, E)$ and a nonnegative cost function c on the edges. That is, each edge e has a nonnegative real number $c(e)$ assigned to it. Assume that G is connected; otherwise, we focus on a connected component of G. The *cost* of a path (or walk) $P = v_0e_1v_1e_2v_2 \ldots v_{\ell-1}e_\ell v_\ell$ is defined to be

$$c(P) = \sum_{i=1}^{\ell} c(e_i).$$

The general problem we consider is to find a path P joining specified nodes such that $c(P)$ is minimum among all such paths. Such a path is called a *shortest path.*

Two specific problems of interest are:

SINGLE SOURCE:
Given a specified node s, find a shortest path from s to every other node of G.

ALL PAIRS:
Find shortest paths joining every pair of nodes in G.

1.3.1 Bellman's inequalities, node potentials and reduced costs

The basic idea behind all methods for solving shortest paths problems is this. Let s be a specified start node. Suppose we know that there is a path from s to v of cost $d(v)$ for each node v, and we find an edge vt satisfying $d(v) + c(vt) < d(t)$. Since appending vt to the path from s to v gives a path from s to t, we know that there is a shorter path to t, of cost $d(v) + c(vt)$. On the other hand, if each edge vw satisfies $d(v) + c(vw) \geq d(w)$, then the method of constructing a path to t by appending an edge vt to the path to v does not improve on the cost $d(t)$.

To study the properties of the values $d(v)$ $(v \in V)$, we focus on *Bellman's inequalities*:

$$d(s) = 0 \tag{1.1}$$

$$d(w) \leq d(v) + c(vw) \text{ for all } vw \in E. \tag{1.2}$$

The values $d(v)$ $(v \in V)$ satisfying Bellman's inequalities are called *node potentials.*

These inequalities and their solutions are fundamental for shortest path problems. We give some elementary but useful results about node potentials.

Proposition 1.1 *If there exists a solution to Bellman's inequalities, then G, c has no edges of negative cost.*

Lemma 1.2 *If $d : V \to \Re$ is a solution to Bellman's inequalities, then for each node v, $d(v)$ is less than or equal to the cost of any walk W from s to v.*

Proof: Let $s = v_0, v_1, \ldots, v_k = v$ be the sequence of nodes in W. We add the inequalities (1.2) for the edges of W:

$$\begin{array}{rcl} d(v_1) - d(v_0) & \le & c(v_0 v_1) \\ d(v_2) - d(v_1) & \le & c(v_1 v_2) \\ \cdots & . & . \\ \cdots & . & . \\ \cdots & . & . \\ d(v_k) - d(v_{k-1}) & \le & c(v_{k-1} v_k) \\ \hline d(v_k) - d(s) & \le & \sum_{i=1}^{k} c(v_{i-1} v_i) = c(W). \end{array}$$

We obtain $c(W) \ge d(v) - d(s)$, and hence by equation (1.1), we find that $d(v) \le c(W)$. ▌

Proposition 1.3 *Let $d : V \to \Re$ be a solution to Bellman's inequalities and let E' be the set of edges satisfying (1.2) with equality, i.e., $E' = \{uv \in E \ : \ d(v) - d(u) = c(uv)\}$. Let E^* be obtained by orienting each edge $e \in E'$ from the end with lower d-value to the end with higher d-value, $E^* = \{(u, v) \ : \ uv \in E'$ and $d(u) \le d(v)\}$. Let G^* be the directed graph (V, E^*). For each node v, every path from s to v in G^* is a shortest s-v path of G, c; if there is at least one such path, then $d(v)$ is the cost of a shortest s-v path.*

Proof: We add the inequalities (1.2) for the edges of a (directed) path P from s to v such that all of P's edges are in E^*, and obtain $d(v) = c(P)$. By Lemma 1.2, the cost of a shortest path must be at least $d(v)$, so P is a shortest path from s to v. ▌

Let $d : V \to \Re$ be a solution to Bellman's inequalities. Define the *reduced cost* of an edge vw, denoted $\bar{c}(vw)$, with respect to the node potentials d to be

$$\bar{c}(vw) = c(vw) + d(v) - d(w).$$

The reduced cost of a path $P = v_0 v_1 \ldots v_\ell$, $\bar{c}(P)$, and the reduced cost of a cycle $Q = v_0 v_1 \ldots v_\ell v_0$ ($v_{\ell+1} = v_0$), $\bar{c}(Q)$, are defined similarly to the cost of a path and of a cycle, namely, $\bar{c}(P) = \sum_{i=1}^{\ell} \bar{c}(v_{i-1} v_i)$ and $\bar{c}(Q) = \sum_{i=1}^{\ell+1} \bar{c}(v_{i-1} v_i)$.

Proposition 1.4 *Let $d : V \to \Re$ be a set of node potentials, and for each edge vw, let its reduced cost be $\bar{c}(vw) = c(vw) + d(v) - d(w)$. The reduced costs satisfy the following properties.*

(1) $\bar{c}(vw)$ is nonnegative for each edge vw.

(2) For any cycle Q, $\bar{c}(Q) = c(Q)$.

(3) Let P be a path with start node v_0 and end node v_ℓ. Then $\bar{c}(P) = c(P) + d(v_0) - d(v_\ell)$.

Proof: The proof is straightforward, using the fact that d satisfies Bellman's inequalities. For parts (2) and (3) of the lemma, add the reduced costs of the edges in the cycle Q or the path P as in Lemma 1.2. ▌

1.3.2 Shortest paths arborescence

Before presenting an algorithm for the single-source shortest paths problem, we state a few more results.

Proposition 1.5 *Suppose that $G,\ c$ has no edges of negative cost. Let W be a walk from v_0 to v_ℓ. Then there exists a path P from v_0 to v_ℓ that is a subsequence of W such that $c(P) \leq c(W)$.*

Proposition 1.6 *Suppose that $G,\ c$ has no edges of negative cost. Let $P = v_0v_1 \cdots v_\ell$ be a shortest path from v_0 to v_ℓ. Then for any i and j, $0 \leq i < j \leq \ell$, $P' = v_iv_{i+1} \ldots v_j$ is a shortest path from v_i to v_j.*

The optimal solutions to the single-source shortest paths problem have a particularly simple structure. An *s-arborescence* T of a graph $G = (V, E)$, where $s \in V(T)$, is a directed graph such that the node set $V(T)$ equals $V(G)$, the number of edges is $|E(T)| = |V(T)| - 1$, corresponding to every (directed) edge $(v, w) \in E(T)$ there is an edge $vw \in E(G)$, and for each node $v \in V(T)\backslash\{s\}$ there is exactly one directed path in T from s to v. Alternatively, T is a directed graph with node set $V(G)$ that has no directed cycles such that corresponding to every (directed) edge $(v, w) \in E(T)$ there is an edge $vw \in E(G)$, there are no edges of T entering node s, and for each node $v \in V\backslash\{s\}$, there is exactly one edge in T entering v. Roughly speaking, T is a spanning tree such that all its edges are directed "away" from s. See Figure 1.2. An s-arborescence T of a network $G,\ c$ is called a *shortest paths s-arborescence* of $G,\ c$ if for each node $v \in V(T)$, the path in T from s to v is a shortest s-v path of $G,\ c$. For each node v, if $v \neq s$, we take $p(v)$ to be the unique node such that $(p(v), v)$ is an edge of T, and we take $p(s) = \emptyset$. We call $p(v)$ the *predecessor* of v. If we have a shortest paths s-arborescence T of $G,\ c$, then a shortest path from s to any node v can be obtained by starting at v and repeatedly moving to the predecessor until we get to node s, i.e., the path $s\ \ldots\ p(p(p(v)))\ p(p(v))\ p(v)\ v$ is a shortest s-v path of $G,\ c$.

Proposition 1.7 *Suppose that $G,\ c$ has no edges of negative cost, and that G is connected. Then there exists a shortest paths s-arborescence for $G,\ c$.*

A shortest paths s-arborescence can be used to find a "tight" solution to Bellman's inequalities; conversely, Bellman's inequalities can be used to check whether a given s-arborescence is a shortest paths s-arborescence.

Theorem 1.8 *Suppose that G is connected and that $G,\ c$ has no edges of negative cost. Then there exists a solution $d : V \to \Re$ to Bellman's inequalities such that for each node v, $d(v)$ is the cost of a shortest path from s to v. Moreover, an s-arborescence T of G is a shortest paths s-arborescence if and only if each edge (v, w) of T satisfies (1.2) with equality, i.e., $d(w) = d(v) + c(vw)$.*

Using the theorem, it is easy to verify whether an s-arborescence T is a shortest paths s-arborescence. For each node v, we take $d'(v)$ to be the cost of the path in T from s to v (we take $d'(s) = 0$). We check whether $d' : V \to \Re$ satisfies Bellman's inequalities. If yes, then T is a shortest paths s-arborescence.

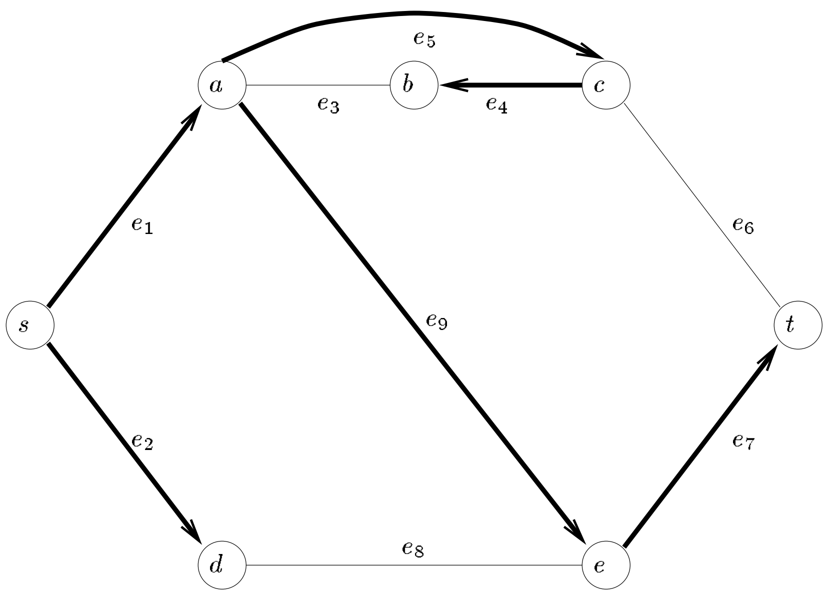

Figure 1.2: An s-arborescence for the graph in Figure 1.1 is indicated by thick lines.

1.3.3 Dijkstra's algorithm

Dijkstra's algorithm efficiently solves the single-source shortest paths problem, assuming that the edge costs are nonnegative. For each node u, let $d(u)$ denote the cost of a shortest path of $G,\ c$ from s to u. The main idea of the algorithm is this: Suppose that for some node set $S \subseteq V$ we know the value $d(u)$ for each $u \in S$. Of course, we have $s \in S$ and $d(s) = 0$. For each node $v \in V \backslash S$, we compute a "temporary label" $\ell(v)$, where

$$\ell(v) = \min\ \{d(u) + c(uv)\ :\ u \in S, \quad uv \in E\}.$$

(If no edge exists joining a node of S to v, then we define $\ell(v) = \infty$.)

Lemma 1.9 *If v^* is a node in $V \backslash S$ such that $\ell(v^*)$ is minimized over all nodes of $V \backslash S$, then $\ell(v^*) = d(v^*)$, i.e., $\ell(v^*)$ is the cost of a shortest s-v^* path of $G,\ c$.*

Proof: First, note that a shortest path from s to a node $u \in S$ such that $d(u) + c(uv^*) = \ell(v^*)$, plus the edge uv^* gives a walk from s to v^* of cost $\ell(v^*)$. Now let $P = v_0 v_1 v_2 \ldots v_q$ $(s = v_0$ and $v^* = v_q)$ be *any* path from s to v^*. To complete the proof, we show that its cost is at least $\ell(v^*)$. Let v_k be the first node of this path not belonging to S. Since the edge costs c are nonnegative, and $\ell(v^*) = \min_{v \in V \backslash S} \ell(v)$, we have

$$c(P)\ \geq\ c(v_0 v_1 v_2 \ldots v_k)\ \geq\ d(v_{k-1}) + c(v_{k-1} v_k)\ \geq\ \ell(v_k)\ \geq\ \ell(v^*).$$

In other words, the cost of every s-v^* path of $G,\ c$ is at least $\ell(v^*)$. ∎

Algorithm Dijkstra's Algorithm
input: Graph $G = (V, E)$, nonnegative edge costs c, and start node s; G is connected.
output: A shortest paths s-arborescence of G, c given by $p : V \backslash \{s\} \to V$, and
for each node v, the cost of a shortest path from s to v, $d(v)$.

STEP 0: $S := \{s\}$;
$d(s) := 0$; $p(s) := \emptyset$;
for each node $v \in V \backslash S$ **do**
$\ell(v) := \infty$; $p(v) := \emptyset$;
end; (for)
$v^* := s$; (v^* is last node added to S)
while $S \neq V$ **do**
STEP 1: **for** each edge v^*x **do**
if $x \in V \backslash S$ and $\ell(x) > d(v^*) + c(v^*x)$
then $\ell(x) := d(v^*) + c(v^*x)$;
$p(x) := v^*$;
end; (if)
end; (for)
STEP 2: (add node v in $V \backslash S$ with smallest $\ell(v)$ to S)
find a node $y \in V \backslash S$ with $\ell(y) = \min\{\ell(v) \ : \ v \in V \backslash S\}$;
$v^* := y$; $S := S \cup \{v^*\}$;
$d(v^*) := \ell(v^*)$;
end; (while)

Theorem 1.10 *Given a network $G = (V, E)$, c, where c is nonnegative, and a start node s, Dijkstra's algorithm correctly computes a shortest paths s-arborescence. The running time is $O(|V|^2)$.*

Remark: Using the Fibonacci heaps data structure, the running time can be improved to $O(|E| + |V| \log |V|)$.

1.3.4 The Floyd-Warshall algorithm

The Floyd-Warshall algorithm efficiently solves the all-pairs shortest paths problem, assuming that the edge costs are nonnegative. First we order the nodes (arbitrarily) as $v_1, v_2, \ldots, v_n$, where $n = |V|$. An *intermediate* node of a path is any node different from the start node and the end node. For $k = 0, 1, 2, \ldots, n$, we define $d^k(u, v)$ to be the cost of a shortest path from u to v, with the added condition that only nodes $\{v_1, v_2, \ldots, v_k\}$ can be used as intermediate nodes. (If $k = 0$, we say that no nodes can be intermediates, i.e., the paths consist of single edges.)

The idea of the algorithm is that the values $d^k(u, v)$ can be computed from the values of $d^{k-1}(u, v)$: A shortest path from u to v that uses only $\{v_1, v_2, \ldots v_k\}$ as intermediate nodes can either use or not use the node v_k. In the former case, its cost is $d^{k-1}(u, v)$. In the latter, its cost is $d^{k-1}(u, v_k) + d^{k-1}(v_k, v)$.

The algorithm also computes the values $d^k(u,u)$ for all u. All these values remain equal to zero always if and only if $G,\ c$ has no edges of negative cost.

Algorithm Floyd-Warshall Algorithm
input: Graph $G = (V, E)$, and nonnegative costs c on the edges.
output: For each pair of nodes $u,\ v$, finds the minimum cost of a path from u to v.

STEP 0: let $d^0(u,v) = c(uv)$ for all $uv \in E$;
let $d^0(u,u) = 0$ for all $u \in V$;
let $d^0(u,v) = \infty$ for all distinct $u,\ v \in V \times V$ such that $uv \notin E$;
STEP 1: **for** $k = 1, 2, \ldots, |V|$ **do**
  for all $u,\ v \in V \times V$ **do**
    $d^k(u,v) := \min\{d^{k-1}(u,v)\ ,\ d^{k-1}(u,v_k) + d^{k-1}(v_k,v)\}$;
  end; (for)
end; (for)

At termination, if $d^n(u,u) = 0$ for all $u \in V$, then the values $d^n(u,v)$ are the required shortest paths costs.

Theorem 1.11 *Given a network $G = (V, E)$, c, where c is nonnegative, the Floyd-Warshall algorithm correctly computes the cost of a shortest path between every pair of nodes. The running time is $O(|V|^3)$.*

If desired, we can modify the algorithm so that it constructs a shortest paths u-arborescence for each node $u \in V$. This requires each node v to have $|V|$ predecessor pointers $p_u(v)$, each representing the previous node in a shortest path from u to v, for some $u \in V$.

Bibliography

[1] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications.* Prentice-Hall, Englewood Cliffs, N. J., 1993.

[2] B. Bollobas, *Graph Theory: An Introductory Course.* Springer-Verlag, New York, 1979.

[3] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications.* Macmillan, London, 1976.

[4] N. Christofides, *Graph Theory: An Algorithmic Approach.* Academic Press, London, 1975.

[5] V. Chvátal, *Linear Programming*, (1983) W. H. Freeman and Company.

[6] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1992.

[7] R. C. Larson and A. R. Odoni, *Urban Operations Research.* Prentice-Hall, Englewood Cliffs, N. J., 1981.

[8] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, New York, 1976.