2. Shortest paths and minimum spanning trees

Let G be a (di)graph and let $w : E(G) \to \mathbb{R}$.

The number $w(e)$ is the **weight** of an edge $e \in E(G)$.

The pair (G, w) is a **network** (or a **weighted (di)graph**).

The weight of an edge in a network may be interpreted as length, delay, probability, cost/profit, exchange rate, ...

Edge weights may also be negative.

In this lecture we study the following two problems together with a number of related problems.

SHORTEST PATH: Given digraph G, a weight function w, and two vertices $a, b \in V(G)$ as input, determine a shortest path (i.e. a path of the minimum possible total weight) from a to b, or conclude that none exists.

MINIMUM SPANNING TREE: Given a connected graph G and a weight function w as input, determine a spanning tree for G that has the minimum total weight among all spanning trees for G.

A generalization of these two problems is the Steiner tree problem, which we will discuss briefly at the end of this lecture.

Sources for this lecture

The material for this lecture has been prepared with the help of [Jun, Chapters 3–4], [Cor, Chapters 24–26], [Wes, Section 2.3], and the following references:

[Che] B. V. Cherkassky and A. V. Goldberg, Negative-cycle detection algorithms, *Math. Program.* **85** (1999) 277–311. (Online in Springer LINK.)

[Prö] H. J. Prömel and A. Steger, *The Steiner Tree Problem*, Friedr. Vieweg & Sohn, Braunschweig, 2002.

[Tar] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia PA, 1983.

Distance in networks

Throughout this section (i.e. until but not including the part on minimum spanning trees) we assume that G is a simple digraph without loops.

Let (G, w) be a network and let $W = (e_1, \dots, e_n)$ be a walk in G.

The **weight** (or **length**) of W is

$$w(W) := w(e_1) + \ldots + w(e_n).$$

The **distance** $d(a, b)$ between two vertices a and b in a network is the minimum length of a **path** from a to b taken over all such paths. If no such path exists, we set $d(a, b) = \infty$. The empty path always has weight zero.

Negative-weight edges

In general, the shortest path problem for arbitrary networks (G, w) with negative edge weights is **NP**-hard.

This is because a longest path in (G, w) is a shortest path in $(G, -w)$. The problem of determining whether a graph contains a long path is **NP**-complete. More specifically, the problem

> HAMILTONIAN PATH (DECISION): Given a graph G as input, decide whether G contains a spanning path.

is **NP**-complete, and remains so when restricted to graphs with only two vertices of degree one. Thus, an algorithm for SHORTEST PATH on arbitrary networks enables us to solve HAMILTONIAN PATH (DECISION).

Good algorithms for SHORTEST PATH are known only in cases where the network does not contain cycles of negative weight (**negative cycles**).

Detecting (and finding) negative cycles is relevant in several applications. For an example of an application in locating opportunities for arbitrage, see [Cor, Exercise 25-3].

Properties of shortest paths

Theorem A.15 *Let (G, w) be a network without negative cycles and let W be a walk from a to b. Then, there exists a path P from a to b that satisfies $w(P) \le w(W)$.*

Proof: If W does not contain repeated vertices, we are done. Otherwise, let $(e_j, e_{j+1}, \ldots, e_k)$, where $e_i = v_i v_{i+1} \in E(G)$ for all $j \le i \le k$, be a subwalk of W for which all the vertices v_i are pairwise distinct except that $v_j = v_{k+1}$. Such a subwalk is clearly a cycle, and hence has nonnegative weight. By removing this cycle from W we obtain a walk from a to b whose weight is at most the weight of W and with one less repeated vertex pair. Since walks are finite, iterating this construction eventually gives us a path P from a to b with $w(P) \le w(W)$. $\square$

Theorem A.16 *Let (G, w) be a network without negative cycles and let P be a shortest path from a to b. Then, any subpath P_{cd} of P from c to d is a shortest path from c to d.*

Proof: To reach a contradiction, suppose that P_{cd} is not a shortest path from c to d. Then, there exists a path P'_{cd} from c to d that satisfies $w(P'_{cd}) < w(P_{cd})$. Denote by P_{ac} and P_{db} the subpaths of P from a to c and from d to b, respectively. Now, the walk $W' = P_{ac} + P'_{cd} + P_{db}$ satisfies

$$w(W') = w(P_{ac}) + w(P'_{cd}) + w(P_{db}) < w(P_{ac}) + w(P_{cd}) + w(P_{db}) = w(P).$$

By Theorem A.15 there exists a path P' from a to b that satisfies $w(P') \le w(W')$, which is a contradiction since P is a shortest path from a to b. Thus, P_{cd} is a shortest path from c to d. $\square$

Corollary A.3 *Let (G, w) be a network without negative cycles. Then, for any two vertices s, b ($s \neq b$),*

$$d(s,b) = \min\{d(s,a) + w(ab) \ : \ a \in N^-(b)\}.$$

Proof: The claim clearly holds if b is not accessible from s. Otherwise, a shortest path P from s to b has to contain a last edge ab. By Theorem A.16, the subpath P_{sa} from s to a is a shortest path. Consequently, $d(s,b) = w(P) = w(P_{sa}) + w(ab) = d(s,a) + w(ab)$. □

The labeling method

The **labeling method** was discovered by Ford (1956, 1962). The exposition here follows [Tar, Chapter 7]. The labeling method takes as input a network (G, w) and a **source** vertex $s \in V(G)$.

The method maintains a tentative distance $d[v]$ and a tentative parent vertex $p[v]$ for each vertex $v \in V(G)$. Initially $p[v] = \text{UNDEF}$ and $d[v] = \infty$ for all $v \in V(G)$, with the exception of $d[s] = 0$.

The method performs a series of **labeling steps**:

```
Select an edge $ab \in E(G)$ such that $d[a] + w(ab) < d[b]$.
Put $d[b] \leftarrow d[a] + w(ab)$ and $p[b] \leftarrow a$.
```

When no labeling step can be performed, the method halts.

Properties of the labeling method

Lemma A.1 *The labeling method maintains the invariant that there exists a walk W from s to v with $w(W) = d[v]$.*

Proof: By induction on the number of labeling steps. □

Lemma A.2 *When the labeling method halts, $w(W) \geq d[v]$ for any walk W from s to v.*

Proof: By induction on the number of edges in W. □

Theorem A.17 *When the labeling method halts, $d[v]$ is the length of a shortest path from s to v if v is reachable from s; otherwise, $d[v] = \infty$. If there is a negative cycle reachable from s, the method never halts.*

Proof: If G contains a negative cycle reachable from s, there are arbitrarily short walks from s to any vertex in the cycle. Thus, the labeling method cannot halt in the presence of negative cycles because this would contradict Lemma A.2.

Hence, there are no negative cycles reachable from s if the labeling method halts. Consequently, to any shortest walk there is a path of the same length by Theorem A.15. So, by Lemmata A.1 and A.2, $d[v] < \infty$ is the length of a shortest path from s to v; moreover, $d[v] = \infty$ if and only if v is not reachable from s. □

Lemma A.3 *The labeling method maintains the invariant that whenever $p[v] \neq$ UNDEF, we have*

$$d[p[v]] + w(p[v]v) \leq d[v] \tag{1}$$

with equality when the labeling method halts.

Proof: By induction on the number of labeling steps. □

Lemma A.4 *Let G_p be the digraph defined by*

$$\begin{aligned} V(G_p) &:= \{s\} \cup \{v \ : \ p[v] \neq \text{UNDEF}\} \\ E(G_p) &:= \{p[v]v \ : p[v] \neq \text{UNDEF}\}. \end{aligned}$$

The labeling method maintains the invariant that either G_p is an arborescence with root s or G_p contains a cycle.

Proof: Exercise. □

Lemma A.5 *If G_p contains a cycle at any point during the execution of the labeling method, the corresponding cycle in G is negative.*

Proof: Without loss of generality let $v_1, \ldots, v_n$ be vertices of a cycle in G_p such that $v_i = p[v_{i+1}]$ for all $1 \leq i < n$ and $v_n v_1$ is the last edge added to the cycle during a labeling step. By (1) we have $d[v_i] + w(v_i v_{i+1}) \leq d[v_{i+1}]$ for all $1 \leq i < n$ before the labeling step that adds $v_n v_1$ to G_p. Since the labeling step is executed for $v_n v_1$, we have $d[v_n] + w(v_n v_1) < d[v_1]$. Adding the inequalities, we obtain $w(v_1 v_2) + \cdots + w(v_{n-1} v_n) + w(v_n v_1) < 0$. □

Theorem A.18 *When the labeling method halts, G_p is an arborescence with root s that spans the vertices reachable from s. Moreover, paths in G_p are shortest paths in G.*

Proof: Immediate from Theorem A.17 and Lemmata A.3, A.4, and A.5. □

Summary

We have now proven that

- The labeling method never halts if there exists a negative cycle reachable from s.
- The labeling method correctly computes distances and shortest paths from s, assuming that it halts. (In the exercises we will prove that the method always halts in the absence of negative cycles after at most $2^{e(G)} - 1$ labeling steps.)
- A cycle in G_p corresponds to a negative cycle in G. (This enables us to find a negative cycle; we will prove that a cycle will eventually appear in G_p if G contains a negative cycle.)

In what follows, we will enhance the labeling method so that it

- in the absence of negative cycles reachable from s, correctly computes distances and shortest paths; and
- in the presence of negative cycles reachable from s, outputs one such cycle; and
- always halts in time $O(n(G)e(G))$, assuming that the arithmetic on edge weights is constant-time.

The scanning method

The **scanning method** is a special case of the labeling method. Each vertex $v \in V(G)$ has one of the three states $S[v] \in \{\text{scanned}, \text{labeled}, \text{unreached}\}$.

Initially all vertices are unreached, with the exception of s, which is labeled. The content of the arrays $d[\cdot]$ and $p[\cdot]$ is the same as in the labeling method.

The scanning method works by repeatedly **scanning** labeled vertices until either there are no labeled vertices or a negative cycle is found.

The scanning of a vertex a consists of performing the labeling operation (if applicable) to each edge ab. If the operation applies, then b becomes labeled. The scan terminates by making a scanned.

Pseudocode for the scanning step is as follows.

```
Procedure SCAN($a$)
(1)  for each $b \in N^+(a)$ do
(2)    if $d[a] + w(ab) < d[b]$ then
(3)        CYCLE-DETECT($a, b$);
(4)        $d[b] \leftarrow d[a] + w(ab)$;
(5)        $p[b] \leftarrow a$;
(6)        $S[b] \leftarrow$ labeled
(7)    end if
(6)  end for
(7)  $S[a] \leftarrow$ scanned
```

The procedure CYCLE-DETECT checks whether the edge ab completes a cycle in G_p. If so, the scanning method returns the corresponding negative cycle in G and halts.

Three scanning orders

We follow [Tar] and consider three scanning orders for vertices that will result in efficient shortest path algorithms for different types of networks.

1. Topological order (the acyclic case).
2. Shortest-first order (Dijkstra's algorithm for networks with nonnegative edge weights).
3. Breadth-first order (Bellman-Ford-Moore algorithm for the general case).

Topological order

If the graph induced by the vertices reachable from s is acyclic, we can scan the labeled vertices in topological order.

Since a scanned vertex can never become labeled again, this algorithm has running time $O(n(G) + e(G))$ (assuming the arithmetic on edge weights is constant time), which includes the time required to run DFS for topological sorting.

Shortest (and longest; replace w by $-w$) paths in acyclic digraphs have important applications in e.g. project scheduling [Jun, Section 3.5].

Shortest-first order (Dijkstra)

In shortest-first order, we scan next a labeled vertex v that has the minimum tentative distance $d[v]$ among all labeled vertices.

In the presence of negative-weight edges, this is in general a bad scanning strategy which results in an inefficient algorithm. However, if there are no negative-weight edges, the resulting algorithm is highly efficient since a scanned vertex never becomes labeled again (exercise). This algorithm was invented by Dijkstra (1959).

With the use of appropriate data structures (e.g. Fibonacci heaps; see [Cor, p. 505–510]), the algorithm of Dijkstra can be implemented to run in time $O(e(G) + n(G) \log n(G))$ (assuming the arithmetic on edge weights is constant time).

Breadth-first order

The breadth-first scanning order is to scan next the vertex least recently labeled. This strategy is due to Bellman (1958), Ford (1956), and Moore (1959).

Typically this strategy is implemented with the help of a queue Q that contains the labeled vertices. The vertex to be scanned is always selected from the head of Q, and the vertices that become labeled (and do not already appear on Q) are appended to the tail of Q.

Analysis of breadth-first order

We next analyze the scanning method with breadth-first scanning order both in the absence and presence of negative cycles reachable from s.

Note that in the absence of negative cycles the scanning method is correct since it is a special case of the labeling method. Thus, we need only to analyze the running time in this case.

In the presence of negative cycles we will prove that a cycle will eventually appear in G_p, which establishes correctness since the cycle is detected by CYCLE-DETECT.

For purposes of analysis we divide the execution of the scanning method into **passes**.

Pass 0 ends when the source vertex s has been scanned for the first time. Pass $k \geq 1$ ends when all the vertices that were in the queue at the end of pass $k - 1$ have been scanned.

We say that the algorithm terminates in k passes if Q is empty at the end of pass k.

Theorem A.19 *Suppose there is no negative cycle reachable from s. Then, the scanning method with breadth-first scanning order terminates in at most $n(G) - 1$ passes.*

Proof: We prove by induction on k that if there is a shortest path from s to v with k edges, then in the beginning of pass k we have that $d[v]$ is the length of a shortest path from s to v. Consequently, v cannot become labeled during or after pass k since this would decrease $d[v]$ and lead to a contradiction by Lemma A.1 and Theorem A.15. The claim follows since a shortest path has at most $n(G) - 1$ edges.

The base case $k = 0$ holds because $d[s] < 0$ implies by Lemma A.1 the existence of a negative cycle (which does not necessarily contain s).

Let P be a shortest path from s to v with $k + 1$ edges and let uv be the last edge on P. The subpath P_{su} is a shortest path by Theorem A.16. In the beginning of pass k, we have $d[u] = w(P_{su})$ by the IH. Moreover, u is either scanned or labeled. In both cases we must have $d[v] = d[u] + w(uv) = w(P)$ by the end of pass k. □

Theorem A.20 *Suppose there exists a negative cycle reachable from s. Then,* CYCLE-DETECT *will detect a cycle in G_p in at most $n(G) - 1$ passes of the scanning method with breadth-first scanning order.*

Proof: Suppose first that $d[s]$ becomes negative after a labeling step. Then, after this step G_p must contain a cycle because all vertices in G_p have indegree one. (Exercise.) Note that s need not occur in this cycle (in which case the cycle is detected at an earlier labeling step).

Next, suppose that $d[s] = 0$ and $p[s] = \text{UNDEF}$ at all times. Let u be the first vertex that is scanned during pass $n(G)$. Let pass(v) be the largest number of a pass during which v appears on the queue. By our assumptions, pass$(s) = 0$ and pass$(u) = n(G)$. Now, pass$(p[v]) \geq$ pass$(v) - 1$ for all $v \in V(G)$ such that $p[v] \neq \text{UNDEF}$. But this implies that there cannot be a path from s to u in G_p. Thus, a cycle must appear in G_p before the end of pass $n(G) - 1$. □

Remarks

- Negative cycle detection is a subtle business. For example, cycles in G_p can appear and disappear [Che]. (Exercise.)
- If there exists a negative cycle reachable from s, then G_p always contains a cycle after the first labeling operation in pass $n(G)$ [Che, Lemma 7].
- For more on negative cycle detection algorithms, see [Che]. To achieve the $O(n(G)e(G))$ time bound for the scanning method we can use e.g. Tarjan's subtree disassembly technique in CYCLE-DETECT.
- For further reference on single-source shortest path algorithms in general, see [Che], [Tar, Chapter 7], [Jun, p. 83], [Cor, Chapter 25] and the references therein.

All-pairs shortest paths

Sometimes it is necessary to compute the distances between all pairs of vertices in a network.

> ALL-PAIRS SHORTEST PATH: Given a digraph G and a weight function w as input, determine the distance $d(a, b)$ (and a shortest path from a to b) for all pairs of vertices a, b; or, if a negative cycle exists in the network, report one such cycle.

Clearly, this problem can be solved by repeated application of a single-source shortest path algorithm, but also more efficient algorithms exist; see [Jun, Section 3.8], [Cor, Chapter 26], [Tar, Section 7.3], and the references therein.

Minimum-weight spanning trees

Let G be a simple connected undirected graph and let $w : E(G) \to \mathbb{R}$ be a weight function.

Let T be a subgraph of G. The **weight** of T is

$$w(T) := \sum_{e \in E(T)} w(e).$$

The minimum-weight spanning tree problem asks for a spanning tree T of G that has the minimum weight among all spanning trees of G.

Let T be a spanning tree of G and let $e \notin E(T)$. Then, $T + e$ contains a unique cycle which we denote by $C_T(e)$.

Theorem A.21 *A spanning tree T has the minimum weight if and only if for every edge $e \notin E(T)$ we have*

$$w(e) \geq w(f) \quad \textit{for all edges } f \textit{ in } C_T(e). \tag{2}$$

Proof: ($\Rightarrow$) Suppose T is a minimum-weight spanning tree. To reach a contradiction, suppose that (2) does not hold for some $e \notin E(T)$ and $f \in E(C_T(e))$. Then, $T' := T - f + e$ is a spanning tree of G and $w(T') = w(T) - w(f) + w(e) < w(T)$, a contradiction.

Proof: ($\Leftarrow$) Let T' be any minimum-weight spanning tree. We prove that if (2) holds for T, then $w(T) = w(T')$. If $T = T'$, the claim holds. Otherwise, we construct a minimum-weight tree T'' that has one more edge in common with T than T'. Repeating this construction a finite number of steps allows us to conclude that T is minimum-weight.

Let $e' \in E(T') \setminus E(T)$. Since e' is a cut-edge in T', removing e' partitions T' into two components, T_1' and T_2'. Let $e \in C_T(e')$ so that $e \neq e'$ and $e = uv$, where $u \in E(T_1')$ and $v \in E(T_2')$. Put $T'' := T' - e' + e$. Clearly, T'' is a spanning tree. Since T' is minimum-weight, we have $w(e') \leq w(e)$ (otherwise $w(T'') < w(T')$, a contradiction). Since (2) holds for T, we have $w(e') \geq w(e)$ because $e \in C_T(e')$. Thus $w(e') = w(e)$, so T'' is minimum-weight. Since T'' has one more edge in common with T than T', we are done. $\square$

Cut, cocycle

A **cut** is a partition of the vertex set of a graph into two nonempty subsets.

Let $S = (W_1, W_2)$ be a cut. We denote by $E(W_1, W_2)$ (or $E(S)$) the set of all edges in G with one endvertex in W_1 and the other in W_2. Such an edge set is sometimes called a **cocycle**.

We denote by $\overline{W}$ the complement of $W \subseteq V(G)$ relative to $V(G)$, that is $\overline{W} := V(G) \setminus W$. Thus, $E(W, \overline{W})$ denotes the set of edges with exactly one endvertex in W.

An observation of Prim

Let G be a connected graph and let $w : E(G) \to \mathbb{R}$ be a weight function. Then, the following algorithm constructs a minimum-weight spanning tree T for G. (We assume that $V(G) = \{1, \ldots, n(G)\}$.)

```
Procedure MINTREE($G$, $w$; $T$)
(1)  for $k = 1$ to $n(G)$ do $V_i \leftarrow \{i\}$; $T_i \leftarrow \emptyset$; end for
(4)  for $k = 1$ to $n(G) - 1$ do
(5)     choose any nonempty $V_i$;
(6)     choose any edge $e \in E(V_i, \overline{V}_i)$ with $w(e) \le w(e')$ for all $e' \in E(V_i, \overline{V}_i)$;
(7)     determine the index $j$ for which $e = uv$, $u \in V_i$, $v \in V_j$;
(8)     $V_i \leftarrow V_i \cup V_j$; $V_j \leftarrow \emptyset$;
(9)     $T_i \leftarrow T_i \cup T_j \cup \{e\}$; $T_j \leftarrow \emptyset$
(10)    if $k = n(G) - 1$ then $V(T) \leftarrow V(G)$; $E(T) \leftarrow T_i$ end if
(11) end for
```

Theorem A.22 *Procedure* MINTREE *constructs a minimum-weight spanning tree.*

Proof: Denote by t the number of edges in $T_1 \cup \cdots \cup T_n$ during the execution of Procedure MINTREE. It suffices to prove that for all $0 \le t \le n - 1$, there exists a minimum-weight spanning tree that contains the edges in $T_1 \cup \cdots \cup T_n$.

We proceed by induction on t. Suppose the claim holds for $T_1 \cup \cdots \cup T_n$ before the **for**-loop on lines 4–11 is executed for the kth time, where $k = t - 1$. The base case $k = 1$ (i.e. $t = 0$) is clear. For the inductive step, let T' be a minimum-weight spanning tree that contains the edges $T_1 \cup \cdots \cup T_n$. Suppose the edge e is selected during the kth iteration. If $e \in E(T')$, we are done. Otherwise, we must show the existence of a minimum-weight spanning tree T'' that contains the edges $T_1 \cup \cdots \cup T_n \cup \{e\}$.

Proof: (cont.) Let $e' \in C_{T'}(e)$ be the unique edge that satisfies $e' \neq e$ and $e' \in E(V_i, \overline{V}_i)$. Put $T'' := T' - e' + e$. We must have $e' \notin T_1 \cup \cdots \cup T_n$ since each T_ℓ consists of edges with both endvertices in V_ℓ, so T'' contains all the necessary edges. Moreover, T'' is a spanning tree because e bridges the components induced by the removal of e' from T'.

We must still show that T'' has the minimum weight. By the selection of e on line 6, $w(e) \le w(e')$. On the other hand, since T' is minimal, $w(e') \le w(e)$ (otherwise T'' would have lesser weight than T', a contradiction). Thus, $w(e') = w(e)$, so T'' has the minimum weight. $\square$

The algorithms of Prim, Kruskal, and Boruvka

By varying the strategy with which the choices on lines 5 and 6 of Procedure MINTREE are made, we obtain several algorithms for the minimum-weight spanning tree problem.

- Prim (1957); Jarník (1930): Always choose V_1 on line 5.
- Kruskal (1956): Always choose V_i and e so that e has the minimum weight among all possible choices of V_i and e.
- Boruvka (1926): For each V_i simultaneously, choose a minimum-weight $e \in E(V_i, \overline{V}_i)$. (This requires that all edges have distinct weights.)

Remarks

- For pseudocode and implementation details of these three algorithms, see [Jun, p. 108–115] and [Cor, Chapter 24].
- It is possible to implement each of the algorithms of Prim, Kruskal, and Boruvka so that they have running time $O(e(G) \log n(G))$. The algorithm of Prim can be made to run in time $O(e(G) + n(G) \log n(G))$ with the use of Fibonacci heaps; see [Cor, p. 505–510]. (We again assume that arithmetic on edge weights is constant-time.)
- A maximum-weight spanning tree can be constructed by replacing w with $-w$ and applying a minimum-weight spanning tree algorithm.

The bottleneck problem

Let P be a path in a connected graph G and let $w : E(G) \to \mathbb{R}$ be a weight function. The **capacity** (or **inf-section**) of P is

$$c(P) = \min\{w(e) \ : \ e \in E(P)\}.$$

If P is empty, we define $c(P) = \infty$.

Consider the following problem.

BOTTLENECK PROBLEM: Given a connected graph G, a weight function $w : E(G) \to \mathbb{R}$, and two vertices u, v as input, determine an u, v-path in G that has the maximum capacity.

Perhaps somewhat surprisingly, paths in a maximum spanning tree of G are paths of maximum capacity.

Cuts and minimum-weight spanning trees

Let T be a spanning tree of G. Since every edge of T is a cut-edge, removing an edge $e \in E(T)$ partitions T into two connected components.

The vertex sets of these components define a cut of G, which we denote by $S_T(e)$.

Clearly, e is the only edge of T that occurs in $E(S_T(e))$. Furthermore, $S_T(e)$ is the only cut of G that contains exactly the edge e of T.

Theorem A.23 *Let T be a spanning tree of G. Then, T has the minimum weight if and only if for every $e \in E(T)$ we have*

$$w(e) \leq w(f) \qquad \text{for each edge } f \in E(S_T(e)). \tag{3}$$

Proof: ($\Rightarrow$) Suppose that T has the minimum weight and that (3) does not hold for some e and $f \in E(S_T(e))$. Then, $T' := T - e + f$ is a tree with $w(T') < w(T)$, a contradiction.

($\Leftarrow$) Suppose (3) holds for a spanning tree T. We show that T satisfies (2). Select any $e \notin E(T)$ and $f \in C_T(e)$, $f \neq e$. But then also $e \in E(S_T(f))$, so $w(e) \geq w(f)$ by (3). This proves that (2) holds for T and hence T has the minimum weight by Theorem A.21. □

Note that a similar claim holds for maximum-weight spanning trees with $w(e) \leq w(f)$ replaced by $w(e) \geq w(f)$.

A solution to the bottleneck problem

Theorem A.24 *Let T be a maximum-weight spanning tree of G. Then, for each pair u, v of vertices, the unique u, v-path in T is a maximum-capacity u, v-path in G.*

Proof: Let $u, v \in V(G)$ and let P be the u, v-path in T. Unless $u = v$ (in which case the claim is trivial), there exists an edge $e \in E(P)$ such that $c(P) = w(e)$. Suppose there exists an u, v-path P' with $c(P') > c(P)$. Then clearly $e \notin E(P')$. Since u and v are in different components of $T - e$, there exists an edge $f \in E(P')$ such that $f \in E(S_T(e))$. Since $c(P') > c(P)$, we must have $w(f) > w(e)$, which is impossible by Theorem A.23 since T has the maximum weight. $\square$

There is a converse to this theorem (exercise).

Additional remarks

- The analogous problem to minimum spanning tree in the context of digraphs is the problem of determining a minimum-weight arborescence in a weighted digraph. According to [Jun], this problem is considerably more difficult than the minimum spanning tree problem; see the references in [Jun, p. 127] for an $O(e(G) + n(G) \log n(G))$ algorithm for this problem.
- Typically in applications determining a minimum spanning tree is not enough. Usually there are additional restrictions to the spanning tree which may or may not make the problem **NP**-hard. See [Jun, Section 4.7].

The Euclidean Steiner problem

Often in applications the real problem is not in computing the minimum spanning tree for a given network, but in *constructing* the network subject to some requirements.

A famous example is the following problem.

> EUCLIDEAN STEINER PROBLEM: Given n points in the plane, find a shortest network (in the Euclidean metric) that interconnects them.

The EUCLIDEAN STEINER PROBLEM is **NP**-hard; see [Prö, Chapter 10] for a discussion of geometric Steiner problems.

The Steiner problem in networks

Let G be a connected graph and let $K \subseteq V(G)$ be a nonempty set of vertices (called **terminals**). A **Steiner tree** for K is a tree in G that contains every terminal.

> STEINER TREE PROBLEM IN NETWORKS: Given a connected graph G, a weight function $w : E(G) \to \mathbb{R}$, and a set of terminals $K \subseteq V(G)$, find a minimum-weight Steiner tree for K.

This problem is **NP**-hard even in the case when all edge weights are equal and positive.

Note that for $|K| = 2$ the problem reduces to the shortest path problem and for $K = V(G)$ to the minimum spanning tree problem.

Remarks

- An excellent recent reference to the Steiner tree problem is [Prö]; also [Jun, Section 4.6] contains a brief discussion with references.
- Steiner trees are relevant for many practical applications, including for example VLSI layout and multicast routing.
- In the Euclidean Steiner problem, a minimum spanning tree for the n points has length at most $2\alpha/\sqrt{3}$, where α is the length of the shortest interconnecting network.