By Corollary 4.2.6, $\det B|S \neq 0$ if and only if the edges of G corresponding to S form a tree; moreover, in this case, $(\det B|S)^2 = 1$. This proves the theorem. □

Theorem 4.2.7 is contained implicitly in [Kirh47]. Not surprisingly, this result may also be used to determine the number of spanning trees of a graph G by considering the incidence matrix of any orientation of G. We need the following simple lemma; then the desired result is an immediate consequence of this lemma and Theorem 4.2.7.

Lemma 4.2.8. *Let A be the adjacency matrix of a graph G and M the incidence matrix of an arbitrary orientation H of G, where both matrices use the same ordering of the vertices for numbering the rows and columns. Then $MM^T = \operatorname{diag}(\deg 1, \dots, \deg n) - A$.*

Proof. The (i,j)-entry of MM^T is the inner product of the i-th and the j-th row of M. For $i \neq j$, this entry is -1 if ij or ji is an edge of H and 0 otherwise. For $i = j$, we get the degree $\deg i$. □

Theorem 4.2.9. *Let A be the adjacency matrix of a graph G and A' the matrix $-A + \operatorname{diag}(\deg 1, \dots, \deg n)$. Then the number of spanning trees of G is the common value of all minors of A' which arise by deleting a row and the corresponding column from A'.* □

In Section 4.8, we will give a different proof for Theorem 4.2.9 which avoids using the theorem of Cauchy and Binet. The matrix A' is called the *degree matrix* or the *Laplacian matrix* of G. As an example, let us consider the case of complete graphs and thus give a third proof for Corollary 1.2.11.

Example 4.2.10. Theorem 4.2.9 contains a formula for the number of all trees on n vertices; note that this formula counts the different trees, not the isomorphism classes of trees. Obviously, the degree matrix of K_n is $A' = nI - J$, where J is the matrix having all entries $= 1$. By Theorem 4.2.9, the number of trees on n vertices is the determinant of a minor of A', that is

$$\begin{vmatrix} n-1 & -1 & \dots & -1 \\ -1 & n-1 & \dots & -1 \\ \dots & \dots & \dots & \dots \\ -1 & -1 & \dots & n-1 \end{vmatrix} = \begin{vmatrix} n-1 & -n & -n & \dots & -n \\ -1 & n & 0 & \dots & 0 \\ -1 & 0 & n & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ -1 & 0 & 0 & \dots & n \end{vmatrix}$$

$$= \begin{vmatrix} 1 & 0 & 0 & \dots & 0 \\ -1 & n & 0 & \dots & 0 \\ -1 & 0 & n & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ -1 & 0 & 0 & \dots & n \end{vmatrix}$$

$$= n^{n-2}.$$

The following exercise concerns a similar application of the matrix tree theorem; see [FiSe58]. A simple direct proof can be found in [Abu90] where this result is also used to give yet another proof for Corollary 1.2.11.

Exercise 4.2.11. Use Theorem 4.2.9 to show that the number of spanning trees of the complete bipartite graph $K_{m,n}$ is $m^{n-1}n^{m-1}$.

Note that we can also define incidence matrices for graphs: the matrix M has entry $m_{ij} = 1$ if vertex i is incident with edge e_j, and $m_{ij} = 0$ otherwise. But the statements analogous to Lemma 4.2.2 and Theorem 4.2.3 do not hold; for example, the three columns of a cycle of length 3 are linearly independent over $\mathbb{Z}$. However, the situation changes if we consider the incidence matrix M as a matrix over $\mathbb{Z}_2$.

Exercise 4.2.12. Prove the analogues of 4.2.2 through 4.2.4 for graphs, where M is considered as a binary matrix.

The incidence matrix M of a graph – considered as a matrix over the integers – is not unimodular in general, as the following exercise shows. Moreover, it provides a further important characterization of bipartite graphs.

Exercise 4.2.13. Let G be a graph with incidence matrix M. Show that G is bipartite if and only if M is totally unimodular as a matrix over $\mathbb{Z}$. Hint: The proof that unimodularity of M is necessary is similar to the proof of Theorem 4.2.5. The converse can be proved indirectly.

Exercise 4.2.14. Let e be an edge of K_n. Determine the number of spanning trees of $K_n \setminus e$.

Exercise 4.2.15. Let G be a forest with n vertices and m edges. How many connected components does G have?

Sometimes, a list of all spanning trees of a given graph is needed, or an arbitrary choice of some spanning tree of G (a *random* spanning tree). These tasks are treated in [CoDN89]; in particular, it is shown that the latter problem can be solved with complexity $O(|V|^3)$.

4.3 Minimal spanning trees

In this section, we consider spanning forests in networks. Thus let (G, w) be a network. For any subset T of the edge set of G, we define the *weight* of T by

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

A spanning forest of G is called a *minimal spanning forest* if its weight is minimal among all the weights of spanning forests; similarly, a *minimal spanning tree* has minimal weight among spanning trees. We restrict ourselves to

spanning trees; the general case can be treated by considering a minimal spanning tree for each connected component of G. Thus, we now assume G to be connected.

Minimal spanning trees were first considered by Boruvka [Bor26a, Bor26b]. Shortly after 1920, electricity was to be supplied to the rural area of Southern Moravia; the problem of finding as economical a solution as possible for the proposed network was presented to Boruvka. He found an algorithm for constructing a minimal spanning tree and published it in the two papers cited above. We will present his algorithm in the next section. Boruvka's papers were overlooked for a long time; often the solution of the minimal spanning tree problem is attributed to Kruskal and Prim [Kru56, Pri57], although both of them quote Boruvka; see the interesting article [GrHe85] for a history of this problem. There one also finds references to various applications reaching from the obvious examples of constructing traffic or communication networks to more remote ones in classification problems, automatic speech recognition, image processing, etc.

As the orientation of edges is insignificant when looking at spanning trees, we may assume that G is a graph. If the weight function w should be constant, every spanning tree is minimal; then such a tree can be found with complexity $O(|E|)$ using a BFS, as described in Section 3.3. For the general case, we shall give three efficient algorithms in the next section. Corollary 1.2.11 and Exercise 4.2.11 show that the examination of all spanning trees would be a method having non-polynomial complexity.

But first we characterize the minimal spanning trees. Let us introduce the following notation. Consider a spanning tree T and an edge e not contained in T. By Lemma 4.1.1, the graph arising from T by adding e contains a unique cycle; we denote this cycle by $C_T(e)$. The following result is of fundamental importance.

Theorem 4.3.1. *Let (G, w) be a network, where G is a connected graph. A spanning tree T of G is minimal if and only if the following condition holds for each edge e in $G \setminus T$:*

$$w(e) \geq w(f) \qquad \text{for every edge } f \text{ in } C_T(e). \qquad (4.1)$$

Proof. First suppose that T is minimal. If (4.1) is not satisfied, there is an edge e in $G \setminus T$ and an edge f in $C_T(e)$ with $w(e) < w(f)$. Removing f from T splits T into two connected components, since f is a bridge. Adding e to $T \setminus f$ gives a new spanning tree T'; as $w(e) < w(f)$, T' has smaller weight than T. This contradicts the minimality of T.

Conversely, suppose that (4.1) is satisfied. We choose some minimal spanning tree T' and show $w(T) = w(T')$, so that T is minimal as well. We use induction on the number k of edges in $T' \setminus T$. The case $k = 0$ (that is, $T = T'$) is trivial. Thus let e' be an edge in $T' \setminus T$. Again, we remove e' from T', so that T' splits into two connected components V_1 and V_2. If we add the path $C_T(e') \setminus \{e'\}$ to $T' \setminus \{e'\}$, V_1 and V_2 are connected again. Hence $C_T(e')$ has to contain an edge e connecting a vertex in V_1 to a vertex in V_2. Note that e cannot be an edge of T', because otherwise $T' \setminus \{e'\}$ would still be connected. The minimality of T' implies $w(e) \geq w(e')$: replacing e' by e in T', we obtain another spanning tree T''; and if $w(e) < w(e')$, this tree would have smaller weight than T', a contradiction. On the other hand, by condition (4.1), $w(e') \geq w(e)$; hence $w(e') = w(e)$ and $w(T'') = w(T')$. Thus T'' is a minimal spanning tree as well. Note that T'' has one more edge in common with T than T'; using induction, we conclude $w(T) = w(T'') = w(T')$. □

Next we give another characterization of minimal spanning trees. To do so, we need two definitions. Let G be a graph with vertex set V. A *cut* is a partition $S = \{X, X'\}$ of V into two nonempty subsets. We denote the set of all edges incident with one vertex in X and one vertex in X' by $E(S)$ or $E(X, X')$; any such edge set is called a *cocycle*. We will require cocycles constructed from trees:

Lemma 4.3.2. *Let G be a connected graph and T a spanning tree of G. For each edge e of T, there is exactly one cut $S_T(e)$ of G such that e is the only edge which T has in common with the corresponding cocycle $E(S_T(e))$.*

Proof. If we remove e from T, the tree is divided into two connected components and we get a cut $S_T(e)$. Obviously, the corresponding cocycle contains e, but no other edge of T. It is easy to see that this is the unique cut is the with the desired property. □

Theorem 4.3.3. *Let (G, w) be a network, where G is a connected graph. A spanning tree T of G is minimal if and only if the following condition holds for each edge $e \in T$:*

$$w(e) \leq w(f) \qquad \text{for every edge } f \text{ in } E(S_T(e)). \qquad (4.2)$$

Proof. First let T be minimal. Suppose that there is an edge e in T and an edge f in $E(S_T(e))$ with $w(e) > w(f)$. Then, by removing e from T and adding f instead, we could construct a spanning tree of smaller weight than T, a contradiction.

Conversely, suppose that (4.2) is satisfied. We want to reduce the statement to Theorem 4.3.1; thus we have to show that condition (4.1) is satisfied. Let e be an edge in $G \setminus T$ and $f \neq e$ an edge in $C_T(e)$. Consider the cocycle $E(S_T(f))$ defined by f. Obviously, e is contained in $E(S_T(f))$; hence (4.2) yields $w(f) \leq w(e)$. □

Exercise 4.3.4. Let (G, w) be a network, and let v be any vertex. Prove that every minimal spanning tree has to contain an edge incident with v which has smallest weight among all such edges.

Exercise 4.3.5. Let (G, w) be a network, and assume that all edges have distinct weights. Show that (G, w) has a *unique* minimal spanning tree [Bor26a].

4.4 The algorithms of Prim, Kruskal and Boruvka

In this section, we will treat three popular algorithms for determining minimal spanning trees, all of which are based on the characterizations given in the previous section. Let us first deal with a generic algorithm which has the advantage of allowing a rather simple proof. The three subsequent algorithms are special cases of this general method which is due to Prim [Pri57].

Algorithm 4.4.1. Let $G = (V, E)$ be a connected graph with vertex set $V = \{1, \dots, n\}$ and $w: E \to \mathbb{R}$ a weight function for G. The algorithm constructs a minimal spanning tree T for (G, w).

Procedure MINTREE($G, w; T$)

(1) **for** $i = 1$ **to** n **do** $V_i \leftarrow \{i\}$; $T_i \leftarrow \emptyset$ **od**;
(2) **for** $k = 1$ **to** $n - 1$ **do**
(3) choose V_i with $V_i \neq \emptyset$;
(4) choose an edge $e = uv$ with $u \in V_i$, $v \notin V_i$, and $w(e) \leq w(e')$ for all edges $e' = u'v'$ with $u' \in V_i$, $v' \notin V_i$;
(5) determine the index j for which $v \in V_j$;
(6) $V_i \leftarrow V_i \cup V_j$; $V_j \leftarrow \emptyset$;
(7) $T_i \leftarrow T_i \cup T_j \cup \{e\}$; $T_j \leftarrow \emptyset$;
(8) **if** $k = n - 1$ **then** $T \leftarrow T_i$ **fi**;
(9) **od**

Theorem 4.4.2. *Algorithm 4.4.1 determines a minimal spanning tree for the network* (G, w).

Proof. We use induction on $t := |T_1| + \dots + |T_n|$ to prove the following claim:

$$\text{For } t = 0, \dots, n-1, \text{ there exists a minimal spanning tree } T \text{ of } G \text{ containing } T_1, \dots, T_n. \quad (4.3)$$

For $t = n - 1$, this claim shows that the algorithm is correct. Clearly, (4.3) holds at the beginning of the algorithm – before the loop (2) to (9) is executed for the first time – since $t = 0$ at that point. Now suppose that (4.3) holds for $t = k - 1$, that is, before the loop is executed for the k-th time. Let $e = uv$ with $u \in V_i$ be the edge which is constructed in the k-th iteration. If e is contained in the tree T satisfying (4.3) for $t = k - 1$, there is nothing to show. Thus we may assume $e \notin T$. Then $T \cup \{e\}$ contains the unique cycle $C = C_T(e)$; obviously, C has to contain another edge $f = rs$ with $r \in V_i$ and $s \notin V_i$. By Theorem 4.3.1, $w(e) \geq w(f)$. On the other hand, by the choice of e in step (4), $w(e) \leq w(f)$. Hence $w(e) = w(f)$, and $T' = (T \cup \{e\}) \setminus \{f\}$ is a minimal spanning tree of G satisfying (4.3) for $t = k$. □

Of course, we cannot give the precise complexity of Algorithm 4.4.1: this depends both on the choice of the index i in step (3) and on the details of the implementation. We now turn to the three special cases of Algorithm 4.4.1 mentioned above. All of them are derived by making steps (3) and (4) in MINTREE precise. The first algorithm was favored by Prim and is generally known as the *algorithm of Prim*, although it was already given by Jarník [Jar30].

Algorithm 4.4.3. Let G be a connected graph with vertex set $V = \{1, \dots, n\}$ given by adjacency lists A_v, and let $w: E \to \mathbb{R}$ be a weight function for G.

Procedure PRIM($G, w; T$)

(1) $g(1) \leftarrow 0$, $S \leftarrow \emptyset$, $T \leftarrow \emptyset$;
(2) **for** $i = 2$ **to** n **do** $g(i) \leftarrow \infty$ **od**;
(3) **while** $S \neq V$ **do**
(4) choose $i \in V \setminus S$ such that $g(i)$ is minimal, and set $S \leftarrow S \cup \{i\}$;
(5) **if** $i \neq 1$ **then** $T \leftarrow T \cup \{e(i)\}$ **fi**;
(6) **for** $j \in A_i \cap (V \setminus S)$ **do**
(7) **if** $g(v) > w(ij)$ **then** $g(v) \leftarrow w(ij)$ and $e(v) \leftarrow ij$ **fi**
(8) **od**
(9) **od**

Theorem 4.4.4. *Algorithm 4.4.3 determines with complexity* $O(|V|^2)$ *a minimal spanning tree* T *for the network* (G, w).

Proof. It is easy to see that Algorithm 4.4.3 is a special case of Algorithm 4.4.1 (written a bit differently): if we always choose V_1 in step (3) of MINTREE, we get the algorithm of Prim. The function $g(i)$ introduced here is just used to simplify finding the shortest edge leaving $V_1 = S$. Hence the algorithm is correct by Theorem 4.4.2; it remains to discuss its complexity. The **while**-loop is executed $|V|$ times. During each of these iterations, the comparisons in step (4) can be done in at most $|V| - |S|$ steps, so that we get a complexity of $O(|V|^2)$. As G is simple, this is also the overall complexity: in step (6), each edge of G is examined exactly twice. □

Example 4.4.5. Let us apply Algorithm 4.4.3 to the undirected version of the network of Figure 3.5, where we label the edges as follows: $e_1 = \{1, 5\}$, $e_2 = \{6, 8\}$, $e_3 = \{1, 3\}$, $e_4 = \{4, 5\}$, $e_5 = \{4, 8\}$, $e_6 = \{7, 8\}$, $e_7 = \{6, 7\}$, $e_8 = \{4, 7\}$, $e_9 = \{2, 5\}$, $e_{10} = \{2, 4\}$, $e_{11} = \{2, 6\}$, $e_{12} = \{3, 6\}$, $e_{13} = \{5, 6\}$, $e_{14} = \{3, 8\}$, $e_{15} = \{1, 2\}$. Thus the edges are ordered according to their weight. We do not need really this ordering for the algorithm of Prim, but will use it later for the algorithm of Kruskal. The algorithm of Prim then proceeds as follows; the resulting minimal spanning tree is indicated by the bold edges in Figure 4.1.

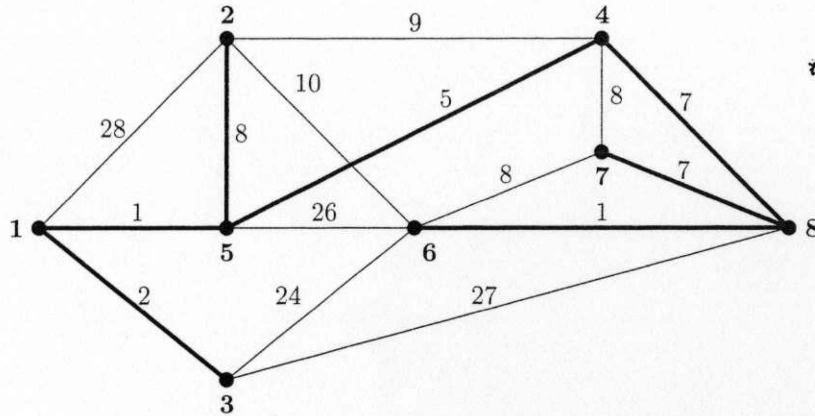

Fig. 4.1. A network

Iteration 1: $i = 1$, $S = \{1\}$, $T = \emptyset$, $g(2) = 28$, $e(2) = e_{15}$, $g(5) = 1$, $e(5) = e_1$, $g(3) = 2$, $e(3) = e_3$
Iteration 2: $i = 5$, $S = \{1, 5\}$, $T = \{e_1\}$, $g(2) = 8$, $e(2) = e_9$, $g(4) = 5$, $e(4) = e_4$, $g(6) = 26$, $e(6) = e_{13}$
Iteration 3: $i = 3$, $S = \{1, 5, 3\}$, $T = \{e_1, e_3\}$, $g(6) = 24$, $e(6) = e_{12}$, $g(8) = 27$, $e(8) = e_{14}$
Iteration 4: $i = 4$, $S = \{1, 5, 3, 4\}$, $T = \{e_1, e_3, e_4\}$, $g(7) = 8$, $e(7) = e_8$, $g(8) = 7$, $e(8) = e_5$
Iteration 5: $i = 8$, $S = \{1, 5, 3, 4, 8\}$, $T = \{e_1, e_3, e_4, e_5\}$, $g(6) = 1$, $e(6) = e_2$, $g(7) = 7$, $e(7) = e_6$
Iteration 6: $i = 6$, $S = \{1, 5, 3, 4, 8, 6\}$, $T = \{e_1, e_3, e_4, e_5, e_2\}$
Iteration 7: $i = 7$, $S = \{1, 5, 3, 4, 8, 6, 7\}$, $T = \{e_1, e_3, e_4, e_5, e_2, e_6\}$
Iteration 8: $i = 2$, $S = \{1, 5, 3, 4, 8, 6, 7, 2\}$, $T = \{e_1, e_3, e_4, e_5, e_2, e_6, e_9\}$

Now we turn to the second special case of Algorithm 4.4.1; this is due to Kruskal [Kru56]. We first give a somewhat vague version.

Algorithm 4.4.6. Let $G = (V, E)$ be a connected graph with $V = \{1, \ldots, n\}$, and let $w\colon E \to \mathbb{R}$ be a weight function. The edges of G are ordered according to their weight, that is, $E = \{e_1, \ldots, e_m\}$ with $w(e_1) \leq \ldots \leq w(e_m)$.

Procedure KRUSKAL$(G, w; T)$

(1) $T \leftarrow \emptyset$;
(2) **for** $k = 1$ **to** m **do**
(3) **if** e_k does not form a cycle together with some edges of T **then** append e_k to T **fi**
(4) **od**

Note that the algorithm of Kruskal is the special case of MINTREE where V_i and e are chosen in such a way that $w(e)$ is minimal among all edges which are still available: that is, among all those edges which do not have both end vertices in one of the sets V_j and would therefore create a cycle. Again, Theorem 4.4.2 shows that the algorithm is correct. Alternatively, we could also appeal to Theorem 4.3.1 here: in step (3), we choose the edge which does not create a cycle with the edges already in the forest and which has minimal weight among all edges with this property. Thus the set T of edges constructed satisfies (4.1), proving again that T is a minimal spanning tree.

Let us consider the complexity of Algorithm 4.4.6. In order to arrange the edges according to their weight and to remove the edge of smallest weight, we use the data structure *priority queue* already described in Section 3.6. Then these operations can be performed in $O(|E| \log |E|)$ steps. It is more difficult to estimate the complexity of step (3) of the algorithm: how do we check whether an edge creates a cycle, and how many steps does this take? Here it helps to view the algorithm as a special case of Algorithm 4.4.1. In step (1), we begin with a (totally) disconnected forest T on $n = |V|$ vertices which consists of n trees without any edges. During each iteration, an edge is added to the forest T if and only if its two end vertices are contained in different connected components of the forest constructed so far; these two connected components are then joined by adding the edge to the forest T. Therefore we may check for possible cycles by keeping a list of the connected components; for this task, we need a data structure appropriate for treating partitions. In particular, operations like disjoint unions (MERGE) and finding the component containing a given element should be easy to perform. Using such a data structure, we can write down the following more precise version of Algorithm 4.4.6.

Algorithm 4.4.7. Let $G = (V, E)$ be a connected graph with $V = \{1, \ldots, n\}$, and let $w\colon E \to \mathbb{R}$ be a weight function on G. We assume that E is given as a list of edges.

Procedure KRUSKAL $(G, w; T)$

(1) $T \leftarrow \emptyset$;
(2) **for** $i = 1$ **to** n **do** $V_i \leftarrow \{i\}$ **od**;
(3) put E into a priority queue Q with priority function w;
(4) **while** $Q \neq \emptyset$ **do**
(5) $e :=$ DELETEMIN(Q);
(6) find the end vertices u and v of e;
(7) find the components V_u and V_v containing u and v, respectively;
(8) **if** $V_u \neq V_v$ **then** MERGE(V_u, V_v); $T \leftarrow T \cup \{e\}$ **fi**
(9) **od**

Now it is easy to determine the complexity of the iteration. Finding and removing the minimal edge e in the priority queue takes $O(\log |E|)$ steps.

Successively merging the original n trivial components and finding the components in step (7) can be done with a total effort of $O(n \log n)$ steps; see [AhHU83] or [CoLR90]. As G is connected, G has at least $n-1$ edges, so that the overall complexity is $O(|E| \log |E|)$. We have established the following result.

Theorem 4.4.8. *The algorithm of Kruskal (as given in 4.4.7) determines with complexity $O(|E| \log |E|)$ a minimal spanning tree for (G, w).* □

For sparse graphs, this complexity is much better than the complexity of the algorithm of Prim. In practice, the algorithm of Kruskal often contains one further step: after each merging of components, it is checked whether there is only one component left; in this case, T is already a tree and we may stop the algorithm.

Example 4.4.9. Let us apply the algorithm of Kruskal to the network of Figure 4.1. The edges $e_1, e_2, e_3, e_4, e_5, e_6$ and e_9 are chosen successively, so that we obtain the same spanning tree as with the algorithm of Prim (although there the edges were chosen in a different order). This has to happen here, since our small example has only one minimal spanning tree. In general, however, the algorithms of Prim and Kruskal will yield different minimal spanning trees.

Now we turn to our third and final special case of Algorithm 4.4.1; this is due to Boruvka [Bor26a] and requires that all edge weights are distinct. Then we may combine several iterations of MINTREE into one larger step: we always treat each nonempty V_i and add the shortest edge leaving V_i. We shall give a comparatively brief description of the resulting algorithm.

Algorithm 4.4.10. Let $G = (V, E)$ be a connected graph with $V = \{1, \dots, n\}$, and let $w: E \to \mathbb{R}$ be a weight function for which two distinct edges always have distinct weights.

Procedure BORUVKA$(G, w; T)$

(1) **for** $i = 1$ **to** n **do** $V_i \leftarrow \{i\}$ **od**;
(2) $T \leftarrow \emptyset$; $M \leftarrow \{V_1, \dots, V_n\}$;
(3) **while** $|T| < n - 1$ **do**
(4)   **for** $U \in M$ **do**
(5)   find an edge $e = uv$ with $u \in U$, $v \notin U$ and $w(e) < w(e')$
            for all edges $e' = u'v'$ with $u' \in U$, $v' \notin U$;
(6)   find the component U' containing v;
(7) $T \leftarrow T \cup \{e\}$;
(8)   **od**
(9)   **for** $U \in M$ **do** MERGE(U, U') **od**
(10) **od**

Theorem 4.4.11. *The algorithm of Boruvka determines a minimal spanning tree for (G, w) in $O(|E| \log |V|)$ steps.*

Proof. It follows from Theorem 4.4.2 that the algorithm is correct. The condition that all edge weights are distinct guarantees that no cycles are created during an execution of the **while**-loop. As the number of connected components is at least halved in each iteration, the **while**-loop is executed at most $\log |V|$ times. We leave it to the reader to give a precise formulation of steps (5) and (6) leading to the complexity of $O(|E| \log |V|)$. (Hint: For each vertex v, we should originally have a list E_v of the edges incident with v.) □

Example 4.4.12. Let us apply the algorithm of Boruvka to the network shown in Figure 4.2. When the **while**-loop is executed for the first time, the edges $\{1,2\}$, $\{3,6\}$, $\{4,5\}$, $\{4,7\}$ and $\{7,8\}$ (drawn bold in Figure 4.2) are chosen and inserted into T. That leaves only three connected components, which are merged during the second execution of the **while**-loop by adding the edges $\{2,5\}$ and $\{1,3\}$ (drawn bold broken in Figure 4.2).

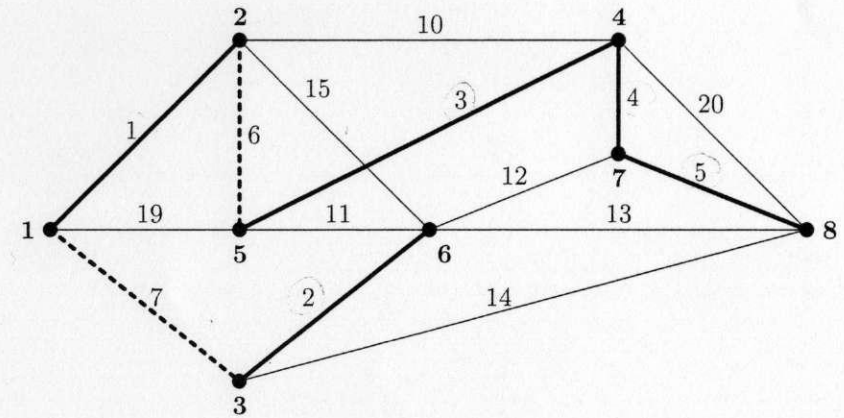

Fig. 4.2. A network

Exercise 4.4.13. Show that the condition that all edge weights are distinct is necessary for the correctness of the algorithm of Boruvka.

Exercise 4.4.14. The following table taken from [BoMu76] gives the distances (in units of 100 miles) between the airports of the cities London, Mexico City, New York, Paris, Peking and Tokyo:

	L	MC	NY	Pa	Pe	To
L	–	56	35	2	51	60
MC	56	–	21	57	78	70
NY	35	21	–	36	68	68
Pa	2	57	36	–	51	61
Pe	51	78	68	51	–	13
To	60	70	68	61	13	–

Find a minimal spanning tree for the corresponding graph.

Exercise 4.4.15. The *tree graph* $T(G)$ of a connected graph G has the spanning trees for G as vertices; two of these trees are adjacent if they have $|V|-2$ edges in common. Prove that $T(G)$ is connected. What can be said about the subgraph of minimal spanning trees (for a given weight function w)?

The complexity of the algorithms discussed in this section can often be improved by using appropriate data structures. Implementations for the algorithms of Prim and Kruskal with complexity $O(|E|\log|V|)$ are given in [Joh75] and [ChTa76]. Using Fibonacci heaps, the algorithm of Prim can be implemented with complexity $O(|E|+|V|\log|V|)$; see [AhMO93]. Boruvka's algorithm (or appropriate variations) can likewise be implemented with complexity $O(|E|\log|V|)$: see [Yao75] and [ChTa76]. Almost linear bounds are in [FrTa87] and [GaGST86]; finally, an algorithm with linear complexity was discovered by Fredman and Willard [FrWi94]; of course, this supposes that the edges are already sorted according to their weights. Unfortunately, the best theoretical algorithms tend to be of no practical interest because of the large size of the implicit constants. There is a simple algorithm with complexity $O(|V|)$ for planar graphs; see [Mat95].

The problem of finding a new minimal spanning tree if we change the weight of an edge and know a minimal spanning tree for the original graph already is discussed in [Fre85] and [Epp94]. On the average, an update may be done in $O(\log|V|)$ steps (under suitable assumptions). Finally, it can be verified in linear time (that is, with complexity $O(|E|)$) whether a given spanning tree is minimal. A similar result holds for the *sensitivity analysis* of minimal spanning trees; this is the problem how much the weight of a given edge e can be increased without changing the minimal spanning tree already known. For the latter two problems, see [DiRT92].

4.5 Maximal spanning trees

For some practical problems, it is necessary to consider *maximal spanning trees*: we want to determine a spanning tree whose weight is maximal among all spanning trees for a given network (G,w). Obviously, a spanning tree T for (G,w) is maximal if and only if T is minimal for $(G,-w)$. Hence we can find a maximal spanning tree by replacing w by $-w$ and using one of the algorithms of Section 4.4. Alternatively, we could also stay with w and just replace *minimum* by *maximum* in the algorithms of Prim, Kruskal and Boruvka: of course, in Kruskal's Algorithm, we then need to order the edges according to decreasing weight.

Let us give some examples where one requires a maximal spanning tree; the first of these is taken from [Chr75].

Example 4.5.1. Consider the problem of sending confidential information to n persons. We define a graph G with n vertices corresponding to the n persons; two vertices i and j are adjacent if it is possible to send information directly from i to j. For each edge ij, let p_{ij} denote the probability that the information sent is overheard; we suppose that these probabilities are independent of each other. Now we replace p_{ij} by $q_{ij}=1-p_{ij}$, that is, by the probability that the information is sent without being overheard. In order to send the information to all n persons, we are looking for a spanning subgraph of G for which the product of the q_{ij} (over the edges occurring in the subgraph) is maximal. Replacing q_{ij} by $w(ij)=\log q_{ij}$, we have reduced our problem to finding a spanning tree of maximal weight.

Problem 4.5.2 (network reliability problem). Let us consider the vertices in Example 4.5.1 as the nodes of a communication network, and let us interpret p_{ij} as the probability that the connection between i and j fails. Then a maximal spanning tree is a tree which maximizes the probability for undisturbed communication between all nodes of the network. This interpretation – and its algorithmic solution – is already contained in [Pri57].

Problem 4.5.3 (bottleneck problem). Let (G,w) be a network, where G is a connected graph, and let

$$W \quad = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} v_2 \ \ldots \ \xrightarrow{e_n} v_n,$$

be any path. Then $c(W)=\min\{w(e_i): i=1,\ldots,n\}$ is called the *capacity* or the *inf-section* of W. (We may think of the cross-section of a tube in a supply network or the capacity of a road.) For each pair (u,v) of vertices of G, we want to determine a path from u to v with maximal capacity.

The following theorem due to Hu [Hu61] reduces Problem 4.5.3 to finding a maximal spanning tree. Thus the algorithms of Prim, Kruskal, and Boruvka – modified for determining maximal spanning trees – can be used to solve the bottleneck problem.

Theorem 4.5.4. *Let (G,w) be a network on a connected graph G, and let T be a maximal spanning tree for G. Then, for each pair (u,v) of vertices, the unique path from u to v in T is a path of maximal capacity in G.*

Proof. Let W be the path from u to v in T, and e some edge of W with $c(W)=c(e)$. Suppose there exists a path W' in G having start vertex u and end vertex v such that $c(W')>c(W)$. Let $S_T(e)$ be the cut of G defined in Lemma 4.3.2 and $E(S_T(e))$ the corresponding cocycle. As u and v are in different connected components of $T\setminus e$, the path W' has to contain some edge f of $E(S_T(e))$. As $c(W')>c(W)$, we must have $w(f)>w(e)$. But then $(T\cup\{f\})\setminus\{e\}$ would be a tree of larger weight than T. □

Exercise 4.5.5. Determine a maximal spanning tree and the maximal capacities for the network of Figure 4.1.