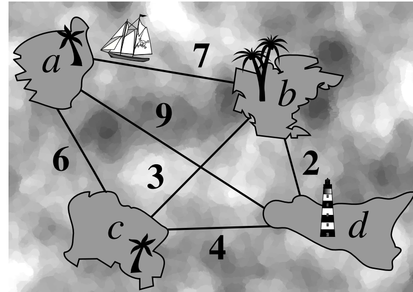
11

Minimum Spanning Trees



The atoll of Taka-Tuka-Land in the South Seas asks you for help.[1] *The people want to connect their islands by ferry lines. Since money is scarce, the total cost of the connections is to be minimized. It needs to be possible to travel between any two islands; direct connections are not necessary. You are given a list of possible connections together with their estimated costs. Which connections should be opened?*

More generally, we want to solve the following problem. Consider a connected undirected graph $G = (V, E)$ with real edge costs $c : E \to \mathbb{R}$. A *minimum spanning tree* (MST) of G is a set $T \subseteq E$ of edges such that the graph (V, T) is a tree minimizing $c(T) := \sum_{e \in T} c(e)$. In our example, the nodes are islands, the edges are possible ferry connections, and the costs are the costs of opening a connection. Throughout this chapter, G denotes an undirected connected graph.

Minimum spanning trees are perhaps the simplest variant of an important family of problems known as *network design problems*. Because MSTs are such a simple concept, they also show up in many seemingly unrelated problems such as clustering, finding paths that minimize the maximum edge cost used, and finding approximations for harder problems, for example the Steiner tree problem and the traveling salesman tour problem. Sections 11.7 and 11.9 discuss this further. An equally good reason to discuss MSTs in a textbook on algorithms is that there are simple, elegant, and fast algorithms to find them. We shall derive two simple properties of MSTs in Sect. 11.1. These properties form the basis of most MST algorithms. The Jarník–Prim algorithm grows an MST starting from a single node and will be discussed in Sect. 11.2. Kruskal's algorithm grows many trees in unrelated parts of the graph at once and merges them into larger and larger trees. This will be discussed in Sect. 11.3. An efficient implementation of the algorithm requires a data structure for maintaining a partition of a set into subsets. Two operations have to be supported by this data structure: "determine whether two elements are in the same subset" and "join two subsets". We shall discuss this union–find data structure in Sect. 11.4. It has many applications besides the construction of minimum spanning trees. Sec-

[1] The figure above was drawn by A. Blancani.

tions 11.5 and 11.6 discuss external-memory and parallel algorithms, respectively, for minimum spanning trees.

Exercise 11.1. If the input graph is not connected, we may ask for a *minimum spanning forest* – a set of edges that defines an MST for each connected component of G. Develop a way to find minimum spanning forests using a single call of an MST routine. Do not find connected components first. Hint: Insert $n-1$ additional edges.

Exercise 11.2 (spanning sets). A set T of edges *spans* a connected graph G if (V,T) is connected. Is a minimum-cost spanning set of edges necessarily a tree? Is it a tree if all edge costs are positive? For a cost function $c: E \to \mathbb{R}$, let $m = 1 + \max_{e \in E} |c(e)|$ and define a cost function c' by $c'(e) = m + c(e)$. Show that a minimum-cost spanning tree with respect to the cost function c is a minimum-cost spanning set of edges with respect to the cost function c' and vice versa.

Exercise 11.3. Reduce the problem of finding *maximum*-cost spanning trees to the minimum-spanning-tree problem.

Exercise 11.4. Suppose you have a highly tuned library implementation of an MST algorithm that you would like to reuse. However, this routine accepts only 32-bit integer edge weights while your graph uses 64-bit floating-point values. Show how to use the library routine provided that $m < 2^{32}$. Your preprocessing and postprocessing may take time $O(m \log m)$.

11.1 Cut and Cycle Properties

We shall prove two simple lemmas which allow one to add edges to an MST and to exclude edges from consideration for an MST. We need the concept of a cut in a graph. A *cut* is a partition $(S, V \setminus S)$ of the node set V into two nonempty parts. A cut

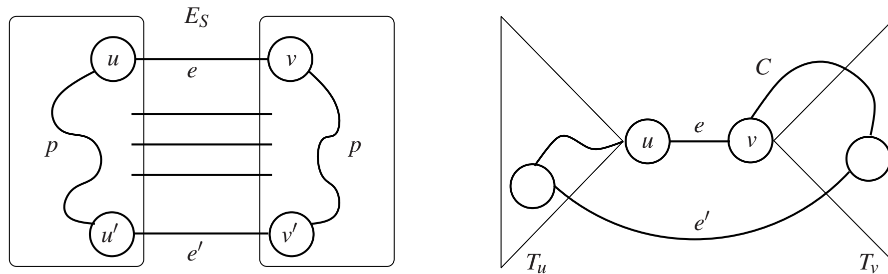


Fig. 11.1. Cut and cycle properties. The *left* part illustrates the proof of the cut property. Edge e has minimum cost in the cut E_S, and p is a path in the MST connecting the endpoints of e; p must contain an edge in E_S. The figure on the *right* illustrates the proof of the cycle property. C is a cycle in G, e is an edge of C of maximum weight, and T is an MST containing e. T_u and T_v are the components of $T \setminus e$, and e' is an edge in C connecting T_u and T_v.

determines the set $E_S = \{\{u,v\} \in E : u \in S, v \in V \setminus S\}$ of edges that connect S with $V \setminus S$. Figure 11.1 illustrates the proofs of the following lemmas.

Lemma 11.1 (cut property). *Let $(S, V \setminus S)$ be a cut and let e be a minimum cost edge in E_S. Consider a set T' of edges that is contained in some MST and contains no edge from E_S. Then $T' \cup \{e\}$ is also contained in some MST. In particular, there is an MST containing e.*

Proof. Consider any MST T of G with $T' \subseteq T$. Let $u \in S$ and $v \in V \setminus S$ be the end-points of e. Since T is a spanning tree, it contains a unique path p from u to v. Since p connects $u \in S$ with $v \in V \setminus S$, it must contain an edge $e' = \{u', v'\}$ with $u' \in S$ and $v' \in V \setminus S$, i.e., $e' \in E_S$. Note that the case $e' = e$ is possible. Recall that we assume $e' \notin T'$. Now, $T'' := (T \setminus \{e'\}) \cup \{e\}$ is also a spanning tree, because removal of e' splits T into two subtrees, which are then joined together by e. Since $c(e) \le c(e')$, we have $c(T'') \le c(T)$, and hence T'' is also an MST. Obviously, $T' \cup \{e\} \subseteq T''$.

The second claim is a special case of the first. Set $T' = \emptyset$. □

Lemma 11.2 (cycle property). *Consider any cycle $C \subseteq E$ and an edge $e \in C$ with maximum cost among all edges of C. Then any MST of $G' = (V, E \setminus \{e\})$ is also an MST of G.*

Proof. Note first that since the edge e that is removed is on a cycle in G, the graph G' is connected. Consider any MST T' of G' and assume that it is not an MST of G. Then there must be an MST T of G with $c(T) < c(T')$. If $e \notin T$, T is a spanning tree of G' cheaper than T', a contradiction. So T contains e. Let $e = \{u, v\}$. Removing e from T splits (V, T) into two subtrees (V_u, T_u) and (V_v, T_v) with $u \in V_u$ and $v \in V_v$. Since C is a cycle, there must be another edge $e' = \{u', v'\}$ in C such that $u' \in V_u$ and $v' \in V_v$. Replacing e by e' in T yields a spanning tree $T'' := (T \setminus \{e\}) \cup \{e'\}$ which does not contain e and for which $c(T'') = c(T) - c(e) + c(e') \le c(T) < c(T')$. So T'' is a spanning tree of G' cheaper than T', a contradiction. □

The cut property yields a simple greedy algorithm for finding an MST; see Fig. 11.2. We initialize T to the empty set of edges. As long as T is not a spanning tree, let $(S, V \setminus S)$ be a cut such that E_S and T are disjoint (S is the union of some but not all connected components of (V, T)), and add a minimum-cost edge from E_S to T.

```
Function genericMST(V, E, c) : Set of Edge
    T := ∅
    while |T| < n − 1 do                 // T is extendible to an MST, but no spanning tree yet
        let (S, V \ S) be a cut such that T and E_S are disjoint;
        let e be a minimum-cost edge in E_S;
        T := T ∪ {e};                                                        // enlarge T
    return T
```

Fig. 11.2. A generic MST algorithm

Lemma 11.3. *The generic MST algorithm is correct.*

Proof. The algorithm maintains the invariant that T is a subset of some minimum spanning tree of G. The invariant is clearly true when T is initialized to the empty set. When an edge is added to T, the invariant is maintained by the cut property (Lemma 11.1). When T has reached size $n-1$, it is a spanning tree contained in an MST, so it is itself an MST. □

Different choices of the cut $(S, V \setminus S)$ lead to different algorithms. We discuss three approaches in detail in the following sections. For each approach, we need to explain how to find a minimum-cost edge in the cut.

The cycle property also leads to a simple algorithm for finding an MST: Set E' to the set of all edges. As long as E' is not a spanning tree, find a cycle in E and delete an edge of maximum cost from T. No efficient implementation of this approach is known however.

Exercise 11.5. Show that the MST is uniquely defined if all edge costs are different. Show that in this case the MST does not change if each edge cost is replaced by its rank among all edge costs.

***Exercise 11.6.** We discuss how to check the cycle property for all non-tree edges. Let T be any spanning tree. Construct a tree whose leaves correspond to the vertices of the graph and whose inner nodes correspond to the edges of T. Start with a forest consisting of n trees, one for each node of the graph. Then process the edges of T in order of increasing cost. In order to process $e = \{u, v\}$ create a new node labelled e and make the roots of the trees containing u and v in the current forest the children of the new node. See Fig. 11.3 for an example. Let C be the resulting tree.

(a) The *lowest common ancestor* $lca_C(x, y)$ of two nodes x and y of C is the lowest node (= maximum depth) having x and y as descendants. Show that for any two nodes x and y of G, the edge associated with the node $lca_C(x, y)$ of C is the heaviest edge on the tree path in T connecting x and y.
(b) Show that T is an MST if for every non-tree edge $e' = \{x, y\}$, we have $c(e') \geq c(lca_C(x, y))$. Remark: C can be preprocessed in time $O(n)$ such that *lca*-queries can be answered in constant time [40].

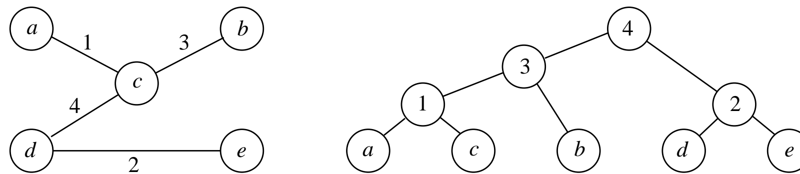


Fig. 11.3. A spanning tree T and the corresponding certification tree C. Observe that $c(lca_C(a, b)) = 3$ is the cost of the most costly edge on the path connecting a and b in T. Check that the analogous statement holds for any pair of nodes of T.

11.2 The Jarník–Prim Algorithm

The Jarník–Prim (JP) algorithm [96, 164, 256] for MSTs is very similar to Dijkstra's algorithm for shortest paths.[2] It grows a tree starting with an arbitrary source node. In each iteration of the algorithm, the cheapest edge connecting a node in the tree with a node outside the tree is added to the tree. The resulting spanning tree is returned. In the notation of the generic algorithm, T is a tree and S is the set of nodes of the tree. Initially, $S = \{s\}$, where s is an arbitrary node, and $T = \emptyset$. Generally, T and S are the edge and node sets of a tree, and E_S is the set of edges having exactly one endpoint in S. Let e be a cheapest such edge and assume $u \notin S$. Then u is added to S and e is added to T.

The main challenge is to find this edge e efficiently. To this end, the algorithm maintains a shortest edge between any node $v \in V \setminus S$ and a node in S in a priority queue Q. The key of v is the minimum cost of any edge connecting v to a node in S. If there is no edge connecting v to S, the key is ∞. The smallest element in Q then gives the desired edge and the node to be added to S.

Assume now that a node u is added to S. We inspect its incident edges $\{u,v\}$. If v lies outside S and the edge is a better connection for v to S, the key value of v is decreased accordingly. Figure 11.4 illustrates the operation of the JP algorithm, and Fig. 11.5 shows the pseudocode. The code uses two auxiliary arrays d and *parent*, where $d[v]$ stores the cost of the shortest edge from $v \in V \setminus S$ to a node in S and $parent[v]$ stores the endpoint in S of the corresponding edge. A value $d[v] = \infty$ means that no connection to S is available. Exploiting our assumption that all edge weights are positive, $d[v] = 0$ indicates that $v \in S$. Note that this convention for encoding membership in S allows us to combine two necessary tests in the inner loop of the pseudocode. Namely, an edge (u,v) only affects the priority queue if it connects S and $V \setminus S$ *and* improves the best connection found. Both conditions are true if and only if $c(e) < d[v]$. The test "**if** $w \in Q$" can be implemented by comparing the old value of $d[w]$ with ∞.

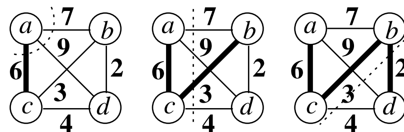


Fig. 11.4. A sequence of cuts (dotted lines) corresponding to the steps carried out by the Jarník–Prim algorithm with starting node a. The edges (a,c), (c,b), and (b,d) are added to the MST.

The only important difference from Dijkstra's algorithm is that the priority queue stores edge costs rather than path lengths. The analysis of Dijkstra's algorithm carries over to the JP algorithm, i.e., the use of a Fibonacci heap priority queue yields a running time $O(n \log n + m)$.

[2] Dijkstra also described this algorithm in his seminal 1959 paper on shortest paths [96]. Since Prim described the same algorithm two years earlier, it is usually named after him. However, the algorithm actually goes back to a paper from 1930 by Jarník [164].

```
Function jpMST : Set of NodeId
    d = ⟨∞,...,∞⟩ : NodeArray[1..n] of ℝ ∪ {∞}          // d[v] is the distance of v from the tree
    parent : NodeArray of Edge                   // (v, parent[v]) is shortest edge between S and v
    Q : NodePQ                                                        // uses d[·] as priority
    Q.insert(s) for some arbitrary s ∈ V
    while Q ≠ ∅ do
        u := Q.deleteMin
        d[u] := 0                                                     // d[u] = 0 encodes u ∈ S
        foreach edge e = {u, v} ∈ E do
            if c(e) < d[v] then                  // c(e) < d[v] implies d[v] > 0 and hence v ∉ S
                d[v] := c(e)
                parent[v] := u
                if v ∈ Q then Q.decreaseKey(v) else Q.insert(v)
        invariant ∀v ∈ Q : d[v] = min{c((u,v)) : (u,v) ∈ E ∧ u ∈ S}
    return {(v, parent[v]) : v ∈ V \ {s}}
```

Fig. 11.5. The Jarník–Prim MST algorithm. Positive edge costs are assumed.

Exercise 11.7. Dijkstra's algorithm for shortest paths can use monotone priority queues. Show that monotone priority queues do *not* suffice for the JP algorithm.

***Exercise 11.8 (average-case analysis of the JP algorithm).** Assume that the edge costs 1, ..., m are assigned randomly to the edges of G. Show that the expected number of *decreaseKey* operations performed by the JP algorithm is then bounded by $O(n\log(m/n))$. Hint: The analysis is very similar to the average-case analysis of Dijkstra's algorithm in Theorem 10.7.

11.3 Kruskal's Algorithm

The JP algorithm is a good general-purpose MST algorithm. Nevertheless, we shall now present an alternative algorithm, Kruskal's algorithm [190]. It also has its merits. In particular, it does not need a sophisticated graph representation, but works even when the graph is represented by its sequence of edges. For sparse graphs with $m = O(n)$, its running time is competitive with the JP algorithm.[3]

Kruskal's algorithm is also an instantiation of the generic algorithm. It grows a forest, i.e., in contrast to the JP algorithm, it grows several trees. In any iteration it adds the cheapest edge connecting two distinct components of the forest. In the notation of the generic algorithm, T is the set of edges already selected. Initially, T is the empty set. Let e be a cheapest edge connecting nodes in distinct subtrees of T. We let $(S, V-S)$ be any cut such that exactly one endpoint of e belongs to S. Then e is the cheapest edge in E_S. We add e to T.

[3] Kruskal's algorithm can be improved so that we get a very good algorithm for denser graphs also [245]. This *filterKruskal* algorithm needs average time $O(m + n\log n\log(m/n))$.

```
Function kruskalMST(V, E, c) : Set of Edge
  T := ∅
  invariant T is a subforest of an MST
  foreach (u,v) ∈ E in ascending order of cost do
    if u and v are in different subtrees of T then
      T := T ∪ {(u,v)}                                    // join two subtrees
  return T
```

Fig. 11.6. Kruskal's MST algorithm

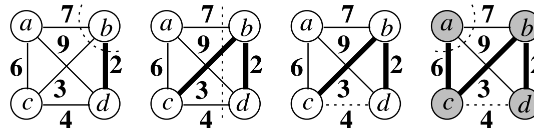


Fig. 11.7. In this example, Kruskal's algorithm first proves that (b,d) and (b,c) are MST edges using the cut property. Then (c,d) is excluded because it is the heaviest edge in the cycle $\langle b,c,d \rangle$, and, finally, (a,c) completes the MST.

How can we find the cheapest edge connecting two components of (V,T)? A first approach would be to first filter out the edges connecting two components and then to find the cheapest such edge. It is much simpler to combine both tasks. We iterate over the edges of G in order of increasing cost. When an edge is considered and its endpoints are in the same component of the current forest, the edge is discarded (filtering step), if its endpoints belong to distinct components, it is added to the forest (selection step). Figure 11.6 gives the pseudocode and Fig. 11.7 gives an example.

In an implementation of Kruskal's algorithm, we have to find out whether an edge connects two components of (V,T). In the next section, we shall see that this can be done so efficiently that the main cost factor is sorting the edges. This takes time $O(m \log m)$ if we use an efficient comparison-based sorting algorithm. The constant factor involved is rather small, so that for $m = O(n)$ we can hope to do better than the $O(m + n \log n)$ JP algorithm.

Exercise 11.9 (streaming MST). Suppose the edges of a graph are presented to you only once (for example over a network connection) and you do not have enough memory to store all of them. The edges do *not* necessarily arrive in sorted order.

(a) Outline an algorithm that nevertheless computes an MST using space $O(n)$.
(*b) Refine your algorithm to run in time $O(m \log n)$. Hint: Process batches of $O(n)$ edges (or use the *dynamic tree* data structure described by Sleator and Tarjan [299]).

11.4 The Union–Find Data Structure

A *partition* of a set M is a collection $M_1, \ldots, M_k$ of subsets of M with the property that the subsets are disjoint and cover M, i.e., $M_i \cap M_j = \emptyset$ for $i \neq j$ and $M = M_1 \cup \cdots \cup M_k$. The subsets M_i are called the *blocks* of the partition. For example, in Kruskal's algorithm, the forest T partitions V. The blocks of the partition are the connected components of (V,T). Some components may be trivial and consist of a single isolated node. Initially, all blocks are trivial. Kruskal's algorithm performs two operations on the partition: testing whether two elements are in the same subset (subtree) and joining two subsets into one (inserting an edge into T).

The *union–find data structure* maintains a partition of the set $1..n$ and supports these two operations. Initially, each element is a block on its own. Each block has a representative. This is an element of the block; it is determined by the data structure and not by the user. The function *find*(i) returns the representative of the block containing i. Thus, testing whether two elements are in the same block amounts to comparing their respective representatives. An operation *union*(r,s) applied to representatives of different blocks joins these blocks into a single block. The new block has r or s as its representative.

Exercise 11.10 (*union* versus *link*). In some other books *union* allows arbitrary parameters from $1..n$ and our restricted operation is called *link*. Explain how this generalized *union* operation can be implemented using *link* and *find*.

To implement Kruskal's algorithm using the union–find data structure, we refine the procedure shown in Fig. 11.6. Initially, the constructor of the class *UnionFind* initializes each node to represent its own block. The **if** statement is replaced by

$r := \mathit{find}(u);\ s := \mathit{find}(v);$
if $r \neq s$ **then** $T := T \cup \{\{u,v\}\};\ \mathit{union}(r,s);$

The union–find data structure is simple to implement as follows. Each block is represented as a rooted tree[4], with the root being the representative of the block. Each element stores its parent in this tree (stored in an array *parent*). A root of such a tree has itself as a *parent* (a self-loop).

The implementation of both operations is simple. We shall first describe unoptimized versions and later discuss optimizations that lead to the pseudocode shown in Fig. 11.8. For *find*(i), we follow parent pointers starting at i until we encounter a self-loop. The self-loop is located at the representative of i, which we return. The implementation of *union*(r,s) is equally simple. We simply make one representative the child of the other. The root of the resulting tree is the representative of the combined block. What we have described so far yields a correct but inefficient union–find data structure. The *parent* references could form long chains that are traversed again and again during *find* operations. In the worst case, each operation may take linear time $\Omega(n)$.

---

[4] Note that this tree may have a structure very different from the corresponding subtree in Kruskal's algorithm.

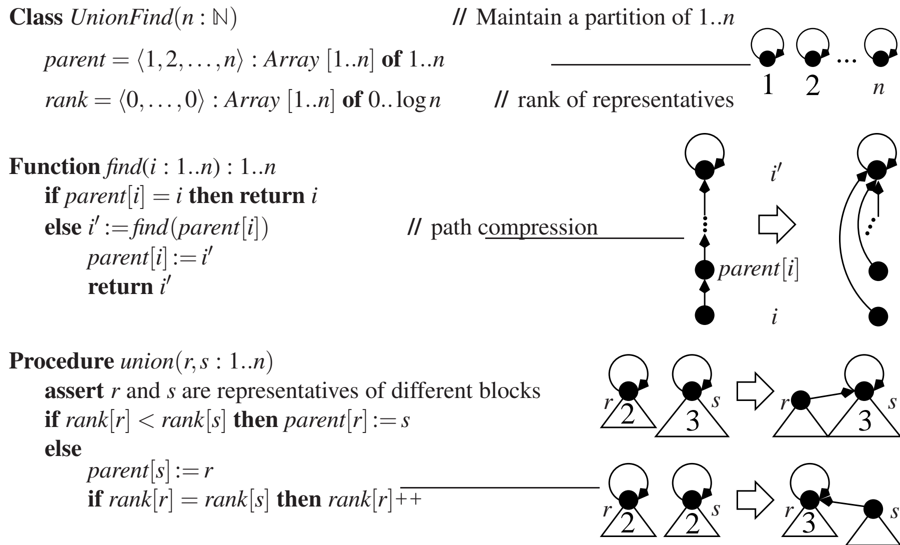


Fig. 11.8. An efficient union–find data structure that maintains a partition of the set $\{1, \ldots, n\}$

Exercise 11.11. Give an example of an n-node graph with $O(n)$ edges where a naive implementation of the union–find data structure as described so far would lead to quadratic execution time for Kruskal's algorithm.

Therefore, Fig. 11.8 introduces two optimizations. The first optimization leads to a limit on the maximum depth of the trees representing blocks. Every representative stores a nonnegative integer, which we call its *rank*. Initially, every element is a representative and has rank 0. When the *union* operation is applied to two representatives with different rank, we make the representative of smaller rank a child of the representative of larger rank. When the two representatives have the same rank, the choice of the parent is arbitrary; however, we increase the rank of the new root. We refer to the first optimization as *union by rank*.

Exercise 11.12. Assume that no *find* operations are called. Show that in this case, the rank of a representative is the height of the tree rooted at it.

The second optimization is called *path compression*. This ensures that a chain of parent references is never traversed twice. Rather, all nodes visited during an operation *find*(i) redirect their parent pointers directly to the representative of i. In Fig. 11.8, we have formulated this rule as a recursive procedure. This procedure first traverses the path from i to its representative and then uses the recursion stack to traverse the path back to i. While the recursion stack is being unwound, the parent pointers are redirected. Alternatively, one can traverse the path twice in the forward direction. In the first traversal, one finds the representative, and in the second traversal, one redirects the parent pointers.

Exercise 11.13. Describe a nonrecursive implementation of *find*.

Theorem 11.4. *Union by rank ensures that the depth of no tree exceeds* $\log n$.

Proof. Without path compression, the rank of a representative is equal to the height of the tree rooted at it. Path compression does not increase heights and does not change the ranks of roots. It therefore suffices to prove that ranks are bounded by $\log n$. We shall show inductively that a tree whose root has rank k contains at least 2^k elements. This is certainly true for $k = 0$. The rank of a root grows from $k-1$ to k when it receives a child of rank $k-1$. Thus, by the induction hypothesis, the root had at least 2^{k-1} descendants before the union operation, and it receives a child which also had at least 2^{k-1} descendants. So the root has at least 2^k descendants after the union operation. □

Union by rank and path compression make the union–find data structure "breath-takingly" efficient – the amortized cost of any operation is almost constant.

Theorem 11.5. *The union–find data structure of Fig. 11.8 performs m find and* $n-1$ *union operations in time* $O(m\alpha(m,n))$. *Here,*

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lceil m/n \rceil) \geq \log n\},$$

where

$$\begin{aligned} A(1,j) &= 2^j && \text{for } j \geq 1,\\ A(i,1) &= A(i-1,2) && \text{for } i \geq 2,\\ A(i,j) &= A(i-1, A(i,j-1)) && \text{for } i \geq 2 \text{ and } j \geq 2. \end{aligned}$$

You will probably find the formulae overwhelming. The function[5] A grows extremely rapidly. We have $A(1,j) = 2^j$, $A(2,1) = A(1,2) = 2^2 = 4$, $A(2,2) = A(1, A(2,1)) = 2^4 = 16$, $A(2,3) = A(1, A(2,2)) = 2^{16}$, $A(2,4) = 2^{2^{16}}$, $A(2,5) = 2^{2^{2^{16}}}$, $A(3,1) = A(2,2) = 16$, $A(3,2) = A(2, A(3,1)) = A(2,16)$, and so on.

Exercise 11.14. Estimate $A(5,1)$.

For all practical n, we have $\alpha(m,n) \leq 5$, and union–find with union by rank and path compression essentially guarantees constant amortized cost per operation.

The proof of Theorem 11.5 is beyond the scope of this introductory text. We refer the reader to [290, 306]. Here, we prove a slightly weaker result that is equally useful for all practical purposes. In order to be able to state the result, we first define the numbers T_k, $k \geq 0$: $T_0 = 1$, and $T_k = 2^{T_{k-1}}$ for $k \geq 1$. The first terms of this rapidly growing sequence of numbers are:

k	0	1	2	3	4	5	...	k
T_k	1	2	$4 = 2^2$	$16 = 2^{2^2}$	$65536 = 2^{2^{2^2}}$	$2^{65536} = 2^{2^{2^{2^2}}}$	...	$2^{2^{\cdot^{\cdot^{2}}}}$ of height k.

[5] The usage of the letter A is a reference to the logician Ackermann [4], who first studied a variant of this function in the late 1920s.

Note that $T_k = A(2, k-1)$ for $k \geq 2$ and that T_k is a "tower of twos" of height k.

For $x > 0$, we define $\log^* x$ as $\min\{k : T_k \geq x\}$. This is also the smallest nonnegative integer k such that $\log^{(k)} x := \log(\log(\ldots\log(x)\ldots))$ (taking the logarithm k times) is less than or equal to 1. The function $\log^* x$ grows extremely slowly. For example, for all $x < 2^{65536}$, we have $\log^* x \leq 5$.

Theorem 11.6. *The union–find data structure with path compression and union by rank completes m find operations and* $n-1$ *union operations in* $\mathrm{O}((m+n)\log^* n)$ *time.*

Proof. (This proof is based on [155].) Consider an arbitrary sequence of $n-1$ *union* and m *find* operations starting with the initialization of the set $1..n$. Since *union* operations take constant time, we can concentrate the analysis on the *find* operations.

The rank of a root can grow while the sequence of operations is being executed. Once a node ceases to be a root, its rank no longer changes. In fact, its rank is no longer important for the execution of the algorithm. However, we shall use it in the analysis. We refer to the rank of a node v at the end of the execution as its *final rank* $fr(v)$. If v ever becomes a child, its final rank is the rank at the time when it first became a child. We make the following observations:

(a) Along paths defined by *parent* pointers, the values of *fr* strictly increase.
(b) When a node v obtains its final rank h, its subtree has as least 2^h nodes.
(c) There are at most $n/2^h$ nodes with *fr* value h.

Proof of the observations: (a) This is an invariant of the data structure. Initially, there are no (non-self-loop) edges at all. When v becomes a child of u during a *union* operation, we have $fr(v) = rank(v) < rank(u)$ right after the operation. Also $rank(u) \leq fr(u)$. The path compression in *find* operations shortcuts paths which can only increase the difference between *fr*-values. (b) is already implied by our proof of Theorem 11.4. Note that a node may lose descendants by path compresssion. However, this happens only when the node is no longer a root. (c) For a fixed *fr* value h and any node v with $fr(v) = h$, let M_v denote the set of children of v just before the moment when v becomes the child of some other node (or at the end of the execution, when v never becomes a child). We prove that the sets M_v are disjoint (this implies observation (c) since, according to (b), each M_v has at least 2^h elements and since there are only n nodes overall). Assume otherwise; say, node w belongs to M_{v_1} and M_{v_2} for distinct nodes v_1 and v_2 with final rank equal to h. Then they cannot both be roots at the end of the execution. Say, v_1 becomes a child of some node u at some point. Then $fr(u) > h$ by (a). By subsequent *union* operations, w can obtain further ancestors. However, by (a), these ancestors all have *fr* values larger than h. Hence, w can never become a descendant of another node with *fr* value h.

We partition the nodes with positive final rank into *rank groups* $G_0, G_1, \ldots$. Rank group G_k contains all nodes v with $T_{k-1} < fr(v) \leq T_k$ (defining $T_{-1} := 0$). For example, G_4 contains all nodes with final ranks between 17 and 65 536. Since, by Theorem 11.4, ranks can never exceed $\log n$, it becomes apparent that only rank groups up to G_4 will be nonempty for practical values of n. Formally, for a node $v \in G_k$

with $k > 0$, $T_{k-1} < fr(v) \le \log n$. Hence, $T_k = 2^{T_{k-1}} < 2^{\log n} = n$, or, equivalently, $k < \log^* n$. Thus, there are at most $\log^* n$ nonempty rank groups.

We are now ready for an amortized analysis of the cost of *find* operations. For an operation *find*(v), we charge r units of cost, where r is the number of nodes on the path $\langle v = v_1, \ldots, v_{r-1}, v_r = s\rangle$ from v to its representative s. Note that the cost of the operation, including setting new *parent* pointers, is $\Theta(r)$ and hence we are covering the asymptotic cost. We distribute these r cost units as follows. We charge one unit to each node on the path with the following exceptions: Nodes v_1, v_{r-1}, v_r, and the nodes v_i whose parent is in a higher rank group are not charged. Note that the final rank of all nodes except maybe v_1 is positive and that all but nodes v_{r-1} and v_r get a new parent by path compression. Since, by observation (a), the *fr* values strictly increase along the path and since there are at most $\log^* n$ nonempty rank groups, the number of exceptions is $3 + \log^* n = \mathrm{O}(\log^* n)$. We charge the exceptions directly to the *find* operation. In this way, each *find* is charged $\mathrm{O}(\log^* n)$ for a total of $\mathrm{O}(m \log^* n)$ for all all *find* operations.

Now we have to take care of the costs charged to nodes. Consider a node v belonging to rank group G_k. When v is charged during a *find* operation, v has a parent u (also belonging to G_k) that is not the root s. Hence, v gets s as a new parent by path compression. By observation (a), $fr(u) < fr(s)$, i.e., whenever node v is charged, it gets a new parent with a larger final rank. Since the ranks in group G_k are bounded by T_k, v is charged at most T_k times. Once its parent is in a higher rank group, v is never charged again. Therefore, the overall cost charged to v is at most T_k, and the total cost charged to nodes in rank group G_k is at most $|G_k| \cdot T_k$.

The final ranks of the nodes in G_k are $T_{k-1}+1, \ldots, T_k$. By observation (3), there are at most $n/2^h$ nodes of final rank h. Therefore,

$$|G_k| \le \sum_{T_{k-1} < h \le T_k} \frac{n}{2^h} < \frac{n}{2^{T_{k-1}}} = \frac{n}{T_k},$$

by the definition of T_k. Equivalently, $|G_k| \cdot T_k < n$, i.e., the total cost charged to rank group k is at most n. Since there are at most $\log^* n$ nonempty rank groups, the total charge to all nodes is at most $n \log^* n$.

Adding the charges of the *find* operations and the nodes, we get $\mathrm{O}(m \log^* n) + n \log^* n = \mathrm{O}((n+m)\log^* n)$. □

11.5 *External Memory

The MST problem is one of the very few graph problems that are known to have an efficient external-memory algorithm. We shall give a simple, elegant algorithm that exemplifies many interesting techniques that are also useful for other external-memory algorithms and for computing MSTs in other models of computation. Our algorithm is a composition of techniques that we have already seen: external sorting, priority queues, and internal union–find. More details can be found in [90].

11.5.1 A Semiexternal Kruskal Algorithm

We begin with an easy case. Suppose we have enough internal memory to store the union–find data structure of Sect. 11.4 for n nodes. This is enough to implement Kruskal's algorithm in the external-memory model. We first sort the edges using the external-memory sorting algorithm described in Sect. 5.12. Then we scan the edges in order of increasing weight, and process them as described by Kruskal's algorithm. If an edge connects two subtrees, it is an MST edge and can be output; otherwise, it is discarded. External-memory graph algorithms that require $\Theta(n)$ internal memory are called *semiexternal* algorithms.

11.5.2 Edge Contraction

If the graph has too many nodes for the semiexternal algorithm of the preceding subsection, we can try to reduce the number of nodes. This can be done using *edge contraction*. Suppose we know that $e = (u,v)$ is an MST edge, for example because e is the least-weight edge incident to v. We add e to the output, and need to remember that u and v are already connected in the MST under construction. Above, we used the union–find data structure to record this fact; now we use edge contraction to encode the information into the graph itself. We identify u and v and replace them by a single node. For simplicity, we again call this node u. In other words, we delete v and *relink* all edges incident to v to u, i.e., any edge (v,w) now becomes an edge (u,w). Figure 11.9 gives an example. In order to keep track of the origin of relinked edges, we associate an additional attribute with each edge that indicates its *original* endpoints. With this additional information, the MST of the contracted graph is easily translated back to the original graph. We simply replace each edge by its original.

We now have a blueprint for an external MST algorithm: Repeatedly find MST edges and contract them. Once the number of nodes is small enough, switch to a semiexternal algorithm. The following subsection gives a particularly simple implementation of this idea.

11.5.3 Sibeyn's Algorithm

Suppose $V = 1..n$. Consider the following simple strategy for reducing the number of nodes from n to n' [90]:

for *current* $:= 1$ **to** $n - n'$ **do**
find the lightest edge incident to current and contract it

Figure 11.9 gives an example, with $n = 4$ and $n' = 2$. The strategy looks deceptively simple. We need to discuss how we find the cheapest edge incident to *current* and how we relink the other edges incident to it, i.e., how we inform its neighbors that they are receiving additional incident edges. We can use a priority queue for both purposes. For each edge $e = \{u,v\}$, we store the item

$$(\min(u,v), \max(u,v), \text{weight of } e, \text{origin of } e)$$

in the priority queue. The ordering is lexicographic by the first and third components, i.e., edges are ordered first by the lower-numbered endpoint and then according to weight. The algorithm operates in phases. In each phase, we process all edges incident to the *current* node, i.e., a phase begins when the first edge incident to *current* is selected from the queue and ends when the last such edge is selected. The edges incident to *current* are selected from the queue in increasing order of weight. Let $(current, relinkTo, *, \{u_0, v_0\})$ be the lightest edge (= first edge delivered by the queue in the phase) incident to *current*. We add its original $\{u_0, v_0\}$ to the MST. Consider any other edge $(current, z, c, \{u'_0, v'_0\})$ incident to *current*. If $z = RelinkTo$, we discard the edge because relinking would turn it into a self-loop. If $z \neq RelinkTo$, we add $(\min(z, RelinkTo), \max(z, RelinkTo), c, \{u'_0, v'_0\})$ to the queue.

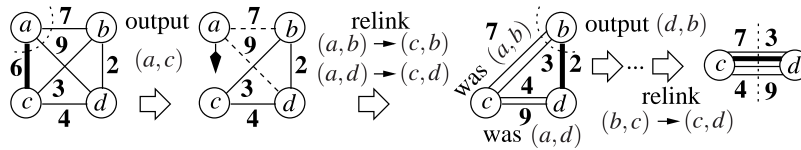


Fig. 11.9. An execution of Sibeyn's algorithm with $n' = 2$. The edge $(c,a,6)$ is the cheapest edge incident to a. We add it to the MST and merge a into c. The edge $(a,b,7)$ becomes an edge $(c,b,7)$ and $(a,d,9)$ becomes $(c,d,9)$. In the new graph, $(d,b,2)$ is the cheapest edge incident to b. We add it to the spanning tree and merge b into d. The edges $(b,c,3)$ and $(b,c,7)$ become $(d,c,3)$ and $(d,c,7)$, respectively. The resulting graph has two nodes that are connected by four parallel edges of weights 3, 4, 7, and 9.

```
Function sibeynMST(V, E, c) : Set of Edge
  let π be a random permutation of 1..n
  Q: priority queue                           // Order: min node, then min edge weight
  foreach e = (u,v) ∈ E do
    Q.insert(min{π(u),π(v)}, max{π(u),π(v)}, c(e), (u,v))
  current := 0                                // we are just before processing node 1
  loop
    (u,v,c,{u0,v0}) := min Q                                      // next edge
    if current ≠ u then                                           // new node
      if u = n − n' + 1 then break loop            // node reduction completed
      Q.deleteMin
      output (u0,v0)             // the original endpoints define an MST edge
      (current, relinkTo) := (u,v)  // prepare for relinking remaining u-edges
    else if v ≠ relinkTo then
      Q.insert((min{v, relinkTo}, max{v, relinkTo}, c, {u0,v0}))      // relink
  S := sort(Q)                                // sort by increasing edge weight
  apply semiexternal Kruskal to S
```

Fig. 11.10. Sibeyn's MST algorithm

Exercise 11.15. Let T be the partial MST just before the edges incident to *current* are inspected. Characterize the content of the queue, i.e., which edges $\{u_0, v_0\}$ have a representative in the queue and what is this representative? Hint: Show that every component of (V,T) contains exactly one node v with $v \geq current$. Call this node the representative of the component and use $rep(u_0)$ to denote the representative of the component containing u_0; then $\{u_0, v_0\}$ is represented by $\{rep(u_0), rep(v_0)\}$ if the two representatives are distinct and is not represented otherwise.

Figure 11.10 gives the details. For reasons that will become clear in the analysis, we renumber the nodes randomly before starting the algorithm, i.e., we chose a random permutation of the integers 1 to n and rename node v as $\pi(v)$. For any edge $e = \{u, v\}$ we store $(\min\{\pi(u), \pi(v)\}, \max\{\pi(u), \pi(v)\}, c(e), e))$ in the queue. The main loop stops when the number of nodes is reduced to n'. We complete the construction of the MST by sorting the remaining edges and then running the semiexternal Kruskal algorithm on them.

Theorem 11.7. *Let* $\mathrm{sort}(x)$ *denote the I/O complexity of sorting x items. The expected number of I/O steps needed by the algorithm sibeynMST is* $\mathrm{O}(\mathrm{sort}(m \ln(n/n')))$.

Proof. From Sect. 6.3, we know that an external-memory priority queue can execute K queue operations using $\mathrm{O}(\mathrm{sort}(K))$ I/Os. Also, the semiexternal Kruskal step requires $\mathrm{O}(\mathrm{sort}(m))$ I/Os. Hence, it suffices to count the number of operations in the reduction phases. Besides the m insertions during initialization, the number of queue operations is proportional to the sum of the degrees of the nodes encountered. Let the random variable X_i denote the degree of node i when it is processed. When i is processed, the contracted graph has $n - i + 1$ remaining nodes and at most m edges. Hence the average degree of each remaining node is at most $2m/(n-i+1)$. Owing to the random permutation of the nodes, each remaining node has the same probability of being removed next. Hence, $\mathrm{E}[X_i]$ coincides with the average degree. By the linearity of expectations, we have $\mathrm{E}[\sum_{1 \leq i \leq n-n'} X_i] = \sum_{1 \leq i \leq n-n'} \mathrm{E}[X_i]$. We obtain

$$\begin{aligned}
\mathrm{E}\left[\sum_{1 \leq i \leq n-n'} X_i\right] &= \sum_{1 \leq i \leq n-n'} \mathrm{E}[X_i] \leq \sum_{1 \leq i \leq n-n'} \frac{2m}{n-i+1} \\
&= 2m\left(\sum_{1 \leq i \leq n} \frac{1}{i} - \sum_{1 \leq i \leq n'} \frac{1}{i}\right) = 2m(H_n - H_{n'}) \\
&= 2m(\ln n - \ln n') + \mathrm{O}(1) = 2m \ln \frac{n}{n'} + \mathrm{O}(1),
\end{aligned}$$

where $H_n := \sum_{1 \leq i \leq n} 1/i = \ln n + \Theta(1)$ is the nth harmonic number (see (A.13)). □

Note that we could do without switching to the semiexternal Kruskal algorithm. However, then the logarithmic factor in the I/O complexity would become $\ln n$ rather than $\ln(n/n')$ and the practical performance would be much worse. Observe that $n' = \Theta(M)$ is a large number, say 10^8. For $n = 10^{12}$, $\ln n$ is three times $\ln(n/n')$.

Exercise 11.16. For any n, give a graph with n nodes and $\mathrm{O}(n)$ edges where Sibeyn's algorithm *without random renumbering* would need $\Omega(n^2)$ relink operations.

11.6 *Parallel Algorithms

The MST algorithms presented in the preceding sections add one edge after another to the MST and thus do not directly yield parallel algorithms. We need an algorithm that indentifies many MST edges at once. *Borůvka's algorithm* [51, 240] does just that. Interestingly, going back to 1926, it is also the oldest MST algorithm. For simplicity, let us assume that all edges have different weights. The algorithm is a recursive multilevel algorithm similar to the list-ranking algorithm presented in Sect. 3.3.2. Borůvka's algorithm applies the cut property to the simple cuts $\{v\}, V \setminus \{v\}$ for all $v \in V$, i.e., it finds the lightest edge incident to each node. Note that these edges cannot contain a cycle as the heaviest edge on the cycle is not the lightest edge incident to either endpoint. These edges are added to the MST and then contracted (see also Sect. 11.5.2). Recursively finding the MST of the contracted graph completes the MST.

It is relatively easy to see how to implement this algorithm sequentially such that each level of recursion takes linear time. We can also easily show that there are at most $\log n$ such levels: Each node finds one MST edge (recall that we assume the graph to be connected). Each new MST edge is found at most twice (once from each of its endpoints). Hence, at least $n/2$ distinct MST edges are identified. Contracting them at least halves the number of nodes.

An attractive feature of Borůvka's algorithm is that finding the new MST edges in each level of recursion is easy to parallelize. The contraction step is more complicated to parallelize, though. The difficulty is that we are not contracting single, unrelated edges but, instead, that the edges to be contracted form a graph. More concretely, consider the directed graph $H = (V, C)$, where $(u, v) \in C$ if $\{u, v\} \in E$ is the lightest edge incident to u. All nodes in H have outdegree 1. If you follow a path in H, the visited edges have nonincreasing weights. Since we assume the edge weights of G to be unique, this can only mean that any path in H ends in a cycle $u \rightleftharpoons v$ of length two, where the two edges of H, (u, v) and (v, u) represent the same edge $\{u, v\}$ of G. In other words, the components of H are "almost" rooted trees, except that there are two nodes pointing to each other instead of a single root. The most difficult step of our parallel algorithm is to transform these *pseudotrees* to *rooted stars*, i.e., rooted trees where all nodes point directly to the root.

The first, easy substep is to convert a pseudotree to a rooted tree. Assuming any ordering on the nodes, consider a cycle $u \rightleftharpoons v$ with $u < v$. We designate u as the root of the tree by replacing (u, v) by the self-loop (u, u).

Next, we have to "flatten" these trees. The pseudocode in Fig. 11.11 uses the doubling algorithm tha we have already seen for list ranking in Sect. 3.3.[6] The while-loop executing the doubling algorithm will perform $\log L$ iterations where $L < n$ is the longest path in H.

The resulting rooted trees are easy to contract. An edge $e = \{u, v\}$ in the input graph is transformed into an edge $\{u.R, v.R\}$ in the contracted graph. The weight remains the same. The endpoints of an edge in the input graph are seperately stored

[6] A work-efficient variant, using ideas similar to the independent-set algorithm for list ranking, is also possible [229].

so that they can be used for outputting the result. Some of the edges defined above are not required for the contracted graph: Those connecting nodes in the same component of H are not needed at all. Among parallel edges, only the lightest has to be kept. This can, for example, be achieved by inserting all edges into a hash table using their endpoints as keys. Then, within each bucket of the hash table, a minimum reduction is performed. Figure 11.12 gives an example.

```
// Input: edges are triples ({u,v}, c, {u,v}) where c is the weight
// Output: triples ({u,v}, c, {u′,v′}) where u′, v′ are the original endpoints
Function boruvkaMST(V, E) : Set of Edge
    if |V| = 1 then return ∅                                   // base case
    foreach u ∈ V do||                                         // find lightest incident edges
        u.e := min {({u,v}, c, o) : {u,v} ∈ E}                  // minimum c; also in parallel
        u.R := other end of u.e               // define pseudotrees – u.R is the sole successor of u
    T := {u.e : u ∈ V}                                         // remember MST edges
    foreach u ∈ V do||                                         // pseudotrees → rooted trees
        if u.R = v ∧ v.R = u ∧ u < v then u.R := u
    while ∃u : u.R ≠ u.R.R do                                  // trees → stars
        foreach u ∈ V do||
            if u.R ≠ u.R.R then u.R := u.R.R                   // doubling
    // contract
    V′ := {u ∈ V : u.R = u}                                    // roots
    E′ := {({u.R, v.R}, c, o) : ({u,v}, c, o) ∈ E ∧ u.R ≠ v.R}  // intertree edges
    optional: among parallel edges in (V′, E′), remove all but the lightest ones
    return T ∪ boruvkaMST(V′, E′)
```

Fig. 11.11. Loop-parallel CREW-PRAM pseudocode for Borůvka's MST algorithm

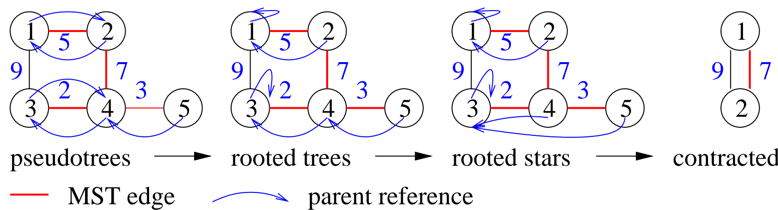


Fig. 11.12. One level of recursion of Borůvka's algorithm.

***Exercise 11.17.** Refine the pseudocode in Fig. 11.11. Make explicit how the recursive instance can work with consecutive integers for node identifiers. Hint: see Sect. 3.3.2. Also, work out how to identify parallel edges in parallel (use hashing or sorting) and how to balance load even when nodes can have very high degree. Each level of recursion should work in expected time $O(m/p + \log p)$.

On a PRAM, Borůvka's algorithm can be implemented to run in polylogarithmic time.

Theorem 11.8. *On a CRCW-PRAM, the algorithm in Fig. 11.11 can be implemented to run in expected time*

$$\mathrm{O}\left(\frac{m}{p}\log n + \log^2 n\right).$$

Proof. We analyze the algorithm without removal of parallel edges. There are at most $\log n$ levels of recursion since the number of nodes is at least halved in each level.

In each level, at most m edges are processed. Analogously to the BFS algorithm in Sect. 9.2.4, we can assign PEs to nodes using prefix sums such that each PE works on $\mathrm{O}(m/p)$ edges. Using local minimum reductions, we can then compute the locally lightest edges. All this takes time $\mathrm{O}(m/p + \log p)$ per level of recursion.

Transforming pseudotrees to rooted trees is possible in constant time. The while-loop for the doubling algorithm executes at most $\log n$ iterations. In recursion level i, there are at most $k = n/2^i$ nodes left. Performing one doubling step takes time $\mathrm{O}(\lceil k/p \rceil)$. Hence, the total time for the while-loops is bounded by

$$\sum_{i=1}^{\log n} \mathrm{O}\left(\left\lceil \frac{n}{p2^i} \right\rceil \log n\right) = \mathrm{O}\left(\frac{n}{p}\log n + \log^2 n\right).$$

Building the recursive instances in adjacency array representation can be done using expected linear work and logarithmic time; see Sect. 8.6.1.

Overall we get expected time

$$\mathrm{O}\left(\frac{m+n}{p}\log n + \log^2 n\right) = \mathrm{O}\left(\frac{m}{p}\log n + \log^2 n\right).$$

For the last simplification, we exploit the fact that the graph is connected and hence $m \geq n-1$. □

The optional removal of superfluous parallel edges can be implemented using bulk operations on a hash table where the endpoints of the edge form the key. The asymptotic work needed is comparable to that for Kruskal's algorithm.

Exercise 11.18 (graphs staying sparse under edge contractions). We say that a class of graphs stays sparse under edge contractions if there is a constant C such that for every graph $G = (V,E)$ in the class and every graph $G' = (V',E')$ that can be obtained from G by edge contractions (and keeping only one copy of a set of parallel edges), we have $|E'| \leq C \cdot |V'|$. The class of planar graphs is sparse under edge contractions. Show that the running time of the parallel MST algorithm improves to $\mathrm{O}\big(m/p + \log^2 n\big)$ for such graphs. Hint: Replace the sentence "In each level, at most m edges are processed" by "In level i of the recursion, there are at most $n/2^i$ nodes and hence $Cn/2^i$ edges left". Removal of parallel edges now becomes essential. Use the result of Exercise 11.17.

11.7 Applications

The MST problem is useful in attacking many other graph problems. We shall discuss the Steiner tree problem and the traveling salesman problem.

11.7.1 The Steiner Tree Problem

We are given a nonnegatively weighted undirected graph $G = (V,E)$ and a set S of nodes. The goal is to find a minimum-cost subset T of the edges that connects the nodes in S. Such a T is called a minimum Steiner tree. It is a tree connecting a set U with $S \subseteq U \subseteq V$. The challenge is to choose U so as to minimize the cost of the tree. The minimum-spanning-tree problem is the special case where S consists of all nodes. The Steiner tree problem arises naturally in our introductory example. Assume that some of the islands in Taka-Tuka-Land are uninhabited. The goal is to connect all the inhabited islands. The optimal solution may have some of the uninhabited islands in the solution.

The Steiner tree problem is **NP**-complete (see Sect. 2.13). We shall show how to construct a solution which is within a factor of two of the optimum. We construct an auxiliary complete graph H with node set S: For any pair u and v of nodes in S, the cost of the edge $\{u,v\}$ in H is their shortest-path distance in G. Let T_A be an MST of H and let c_H be its cost. Replacing every edge of T_A by the path it represents in G yields a subgraph of G connecting all the nodes in S. The resulting subgraph of G has cost c_H and may contain parallel edges and cycles. We remove parallel edges and delete edges from cycles until the remaining subgraph is cycle-free. The cost of the resulting Steiner tree is at most c_H.

Theorem 11.9. *The algorithm above constructs a Steiner tree which has at most twice the cost of an optimal Steiner tree.*

Proof. Let c_H be the cost of the MST in H. Recall that H is the complete graph on the node set S and that the cost of any edge $\{u,v\}$ of H is the cost of a shortest path connecting u and v in G. The algorithm constructs a Steiner tree for S of cost at most c_H. We show that $c_H \leq 2c(T_{\text{opt}})$, where T_{opt} is a minimum Steiner tree for S in G. To this end, it suffices to show that the auxiliary graph H has a spanning tree of cost at most $2c(T_{\text{opt}})$. Fig. 11.13 indicates how to construct such a spanning tree. "Walking once around the Steiner tree" defines a cycle C in G of cost $c(C) = 2c(T_{\text{opt}})$; observe that every edge in T_{opt} occurs exactly twice in this cycle. Deleting the nodes outside S in this cycle gives us a cycle in H. The cost of this cycle in H is at most $2c(T_{\text{opt}})$, because edge costs in H are shortest-path distances in G. More precisely, consider subpaths $P = \langle s, \ldots, t\rangle$ of C whose endpoints s and t are in S and whose interior nodes are outside S. The cost of the edge $\{s,t\}$ of H is the shortest-path distance between s and t in G and hence at most the cost of the path P in G. We have now shown the existence of a cycle through all nodes of H of cost at most $2c(T_{\text{opt}})$. Removing any edge of this cycle gives us a spanning tree of H. □

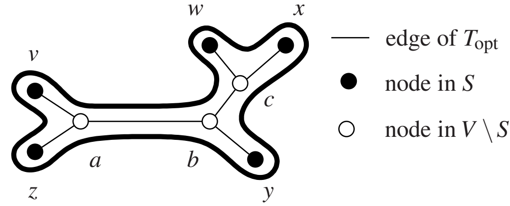


Fig. 11.13. "Once around the tree". The tree T_{opt} shown is a minimum Steiner tree for $S = \{v,w,x,y,z\}$. It also contains the nodes a, b, and c in $V \setminus S$. Walking once around the tree yields the cycle $\langle v,a,b,c,w,c,x,c,b,y,b,a,z,a,v\rangle$ in G, of cost $2c(T_{\text{opt}})$. Removal of the nodes outside S yields the cycle $\langle v,w,x,y,z,v\rangle$ in the auxiliary graph H. The cost of this cycle is at most $2c(T_{\text{opt}})$. Removal of any edge of the cycle yields a spanning tree of the auxiliary graph.

Exercise 11.19. Improve the above bound to $2(1-1/|S|)$ times the optimum.

The algorithm can be implemented to run in time $\mathrm{O}(m+n\log n)$ [212]. Algorithms with better approximation ratios exist [265].

Exercise 11.20. Outline an implementation of the algorithm above and analyze its running time. Aim for running time $\mathrm{O}(|S|(m+n\log n))$.

11.7.2 Traveling Salesman Tours

The traveling salesman problem is one of the most intensively studied optimization problems [18, 193, 317]: Given an undirected complete graph on a node set V with edge weights $c(e)$, the goal is to find the minimum-weight simple cycle passing through all nodes (also called a *tour*). This is the path a traveling salesman would want to take if his goal was to visit all nodes of the graph exactly once. We assume in this section that the edge weights satisfy the triangle inequality, i.e., $c(u,v)+c(v,w) \geq c(u,w)$ for all nodes u, v, and w. An important consequence of this assumption is that nonsimple cycles cannot lead to better tours than simple cycles, as dropping repeated nodes from a cycle does not increase its weight. There is a simple relation between the cost of MSTs and that of travaling salesman tours.

Theorem 11.10. *Let C_{opt} and C_{MST} be the costs of an optimal tour and of an MST, respectively. Then*

$$C_{\text{MST}} \leq C_{\text{opt}} \leq 2C_{\text{MST}}.$$

Proof. Let C be an optimal tour. Deleting any edge from C yields a spanning tree. Thus $C_{\text{MST}} \leq C_{\text{opt}}$. Conversely, let T be an MST. Walking once around the tree as shown in Fig. 11.13 gives us a cycle of cost at most $2C_{\text{MST}}$, passing through all nodes. It may visit nodes several times. Deleting an extra visit to a node does not increase the cost, owing to the triangle inequality. More precisely, replacing the cycle edges (u,v) and (v,w), where v is a node visited more than once, by (u,w) does not increase the cost. □

In the remainder of this section, we shall briefly outline a technique for improving the lower bound of Theorem 11.10. We need two additional concepts: 1-trees and node potentials. Let G' be obtained from G by deleting node 1 and the edges incident to it. A minimum 1-tree consists of the two cheapest edges incident to node 1 and an MST of G'. Since deleting the two edges incident to node 1 from a tour C yields a spanning tree of G', we have $C_1 \le C_{\text{opt}}$, where C_1 is the minimum cost of a 1-tree. A node potential is any real-valued function π defined on the nodes of G. We have also used node potentials in Sect. 10.7. A node potential π yields a modified cost function c_π defined as

$$c_\pi(u,v) = c(u,v) + \pi(v) + \pi(u)$$

for any pair u and v of nodes. For any tour C, the costs under c and c_π differ by $2S_\pi := 2\sum_v \pi(v)$, since a tour uses exactly two edges incident to any node. Let T_π be a minimum 1-tree with respect to c_π. Then

$$c_\pi(T_\pi) \le c_\pi(C_{\text{opt}}) = c(C_{\text{opt}}) + 2S_\pi,$$

and hence

$$c(C_{\text{opt}}) \ge \max_\pi \left(c_\pi(T_\pi) - 2S_\pi\right).$$

This lower bound is known as the Held–Karp lower bound [146, 147]. The maximum is over all node potential functions π. It is hard to compute the lower bound exactly. However, there are fast iterative algorithms for approximating it. The idea is as follows, and we refer the reader to the original papers for details. Assume we have a potential function π and the optimal 1-tree T_π with respect to it. If all nodes of T_π have degree two, we have a traveling salesman tour and stop. Otherwise, we make the edges incident to nodes of degree larger than two a little more expensive and the edges incident to nodes of degree 1 a little cheaper. This can be done by modifying the node potential of v as follows. We define a new node potential π' by

$$\pi'(v) = \pi(v) + \varepsilon \cdot (\deg(v, T_\pi) - 2),$$

where ε is a parameter which goes to 0 with increasing iteration number, and $\deg(v, T_\pi)$ is the degree of v in T_π. We next compute an optimal 1-tree with respect to π' and hope that it will yield a better lower bound.

11.8 Implementation Notes

The minimum-spanning-tree algorithms discussed in this chapter are so fast that the running time is usually dominated by the time required to generate the graphs and appropriate representations. The JP algorithm works well for all m and n if an adjacency array representation (see Sect. 8.2) of the graph is available. Pairing heaps [230] are a robust choice for the priority queue. Kruskal's algorithm may be faster for sparse graphs, in particular if only a list or array of edges is available or if we know how to sort the edges very efficiently.

The union–find data structure can be implemented more space-efficiently by exploiting the observation that only representatives need a rank, whereas only nonrepresentatives need a parent. We can therefore omit the array *rank* in Fig. 11.6. Instead, a root of rank g stores the value $n+1+g$ in *parent*. Thus, instead of two arrays, only one array with values in the range $1..n+1+\lceil\log n\rceil$ is needed. This is particularly useful for the semiexternal algorithm [90].

11.8.1 C++

LEDA [194] uses Kruskal's algorithm for computing MSTs. The union–find data structure is called *partition* in LEDA. The Boost graph library [50] and LEMON graph library [200] give choices of algorithms for computing MSTs. They also provide the union–find data structure.

11.8.2 Java

The JGraphT [166] library gives a choice between Kruskal's and the JP algorithm. It also provides the union–find data structure.

11.9 Historical Notes and Further Findings

There is a randomized linear-time MST algorithm that uses phases of Borůvka's algorithm to reduce the number of nodes [175, 182]. The second building block of this algorithm reduces the number of edges to about $2n$: We sample $O(m/2)$ edges randomly, find an MST T' of the sample, and remove edges $e \in E$ that are the heaviest edge in a cycle in $e \cup T'$. The last step is difficult to implement efficiently. But, at least for rather dense graphs, this approach can yield a practical improvement [178]. An adaptation for the external-memory model [2] saves a factor $\ln(n/n')$ in the asymptotic I/O complexity compared with Sibeyn's algorithm but is impractical for currently interesting values of n owing to its much larger constant factor in the O-notation.

The theoretically best *deterministic* MST algorithm [65, 253] has the interesting property that it has optimal worst-case complexity, although it is not known exactly what this complexity is. Hence, if you were to come up with a completely different deterministic MST algorithm and prove that your algorithm runs in linear time, then we would know that the old algorithm also runs in linear time.

There has been a lot of work on parallel MST algorithms. In particular, the linear-work algorithm [175] can be parallelized [76, 141]. Bader and Cong [26] achieved speedup in practice using a hybrid of Borůvka's algorithm and Prim's algorithm which starts growing trees from multiple source nodes in parallel. Zhou [333] gave an efficient shared-memory implementation of Borůvka's algorithm by exploiting the *priority update principle* [295]. The *filterKruskal* algorithm mentioned above can also be partially parallelized [245]. When a graph is partitioned into blocks, it

suffices to consider the cut edges and the MST edges of the blocks. Wassenberg et al. developed a parallel version of Kruskal's algorithm for image segmentation based on this observation [326].

Minimum spanning trees define a single path between any pair of nodes. Interestingly, this path is a *bottleneck shortest path* [9, Application 13.3], i.e., it minimizes the maximum edge cost for all paths connecting the nodes in the original graph. Hence, finding an MST amounts to solving the all-pairs bottleneck-shortest-path problem in much less time than that for solving the all-pairs shortest-path problem.

A related and even more frequently used application is clustering based on the MST [9, Application 13.5]: By dropping $k-1$ edges from the MST, it can be split into k subtrees. The nodes in a subtree T' are far away from the other nodes in the sense that all paths to nodes in other subtrees use edges that are at least as heavy as the edges used to cut T' out of the MST.

Many applications lead to MST problems on complete graphs. Frequently, these graphs have a compact description, for example if the nodes represent points in the plane and the edge costs are Euclidean distances (these MSTs are called Euclidean minimum spanning trees). In these situations, it is an important concern whether one can rule out most of the edges as too heavy without actually looking at them. This is the case for Euclidean MSTs. It can be shown that Euclidean MSTs are contained in the Delaunay triangulation [85] of the point set. This triangulation has linear size and can be computed in time $O(n \log n)$. This leads to an algorithm of the same time complexity for Euclidean MSTs.

We have discussed the application of MSTs to the Steiner tree and the traveling salesman problem. We refer the reader to the books [9, 18, 189, 193, 322] for more information about these and related problems.