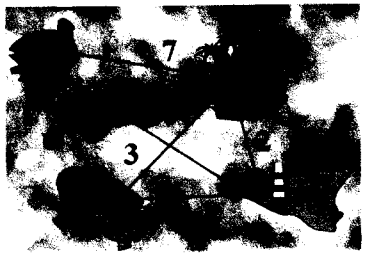
11

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 defined by a set $T \subseteq E$ of edges such that the graph (V,T) is a tree where $c(T) := \sum_{e \in T} c(e)$ is minimized. 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. Sections 11.6 and 11.8 discuss this further. An equally good reason to discuss MSTs in a textbook on algorithms is that there are simple, elegant, 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 partitions of a set of elements under two operations: "determine whether two elements are in the same subset" and "join two subsets". We shall discuss the union–find data structure in Sect. 11.4. This has many applications besides the construction of minimum spanning trees.

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

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?

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

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* in a connected graph is a subset E' of edges such that $G \setminus E'$ is not connected. Here, $G \setminus E'$ is an abbreviation for $(V, E \setminus E')$. If S is a set of nodes with $\emptyset \neq S \neq V$, the set of edges with exactly one endpoint in S forms a cut. Figure 11.1 illustrates the proofs of the following lemmas.

Lemma 11.1 (cut property). *Let E' be a cut and let e be a minimal-cost edge in E'. There is then an MST T of G that contains e. Moreover, if T' is a set of edges that is contained in some MST and T' contains no edge from E', then $T' \cup \{e\}$ is also contained in some MST.*

Proof. We shall prove the second claim. The first claim follows by setting $T' = \emptyset$. Consider any MST T of G with $T' \subseteq T$. Let u and v be the endpoints of e. Since T is a spanning tree, it contains a path from u to v, say p. Since E' is a cut separating u and v, p must contain an edge from E', say e'. 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 $c(e) \leq c(e')$, we have $c(T'') \leq c(T)$, and hence T'' is also an MST. □

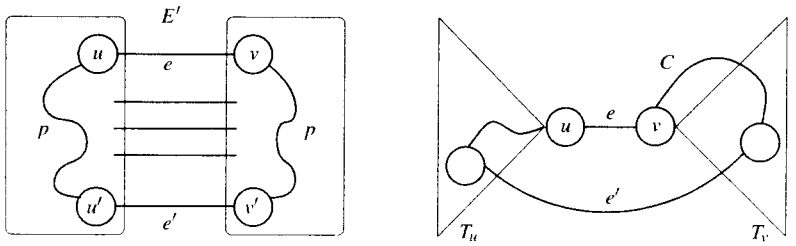


Fig. 11.1. Cut and cycle properties. The *left* part illustrates the proof of the cut property. e is an edge of minimum cost in the cut E', and p is a path in the MST connecting the endpoints of e. p must contain an edge in E'. The figure on the *right* illustrates the proof of the cycle property. C is a cycle in G, e is an edge of C of maximal 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

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

Proof. Consider any MST T of G. Suppose T contains $e = (u,v)$. Edge e splits T into two subtrees T_u and T_v. There must be another edge $e' = (u',v')$ from C such that $u' \in T_u$ and $v' \in T_v$. $T' := (T \setminus \{e\}) \cup \{e'\}$ is a spanning tree which does not contain e. Since $c(e') \leq c(e)$, T' is also an MST. □

The cut property yields a simple greedy algorithm for finding an MST. Start with an empty set T of edges. As long as T is not a spanning tree, let E' be a cut not containing any edge from T. Add a minimal-cost edge from E' to T.

Different choices of E' lead to different specific algorithms. We discuss two approaches in detail in the following sections and outline a third approach in Sect. 11.8. Also, 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 T to the set of all edges. As long as T is not a spanning tree, find a cycle in T and delete an edge of maximal cost from T. No efficient implementation of this approach is known, and we shall not discuss it further.

Exercise 11.4. 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.

11.2 The Jarník–Prim Algorithm

The Jarník–Prim (JP) algorithm [98, 158, 56] for MSTs is very similar to Dijkstra's algorithm for shortest paths.[2] Starting from an (arbitrary) source node s, the JP algorithm grows an MST by adding one node after another. At any iteration, S is the set of nodes already added to the tree, and the cut E' is the set of edges with exactly one endpoint in S. A minimum-cost edge leaving S is added to the tree in every iteration. The main challenge is to find this edge efficiently. To this end, the algorithm maintains the shortest connection between any node $v \in V \setminus S$ and S in a priority queue Q. The smallest element in Q gives the desired edge. When a node is added to S, its incident edges are checked to see whether they yield improved connections to nodes in $V \setminus S$. Fig. 11.2 illustrates the operation of the JP algorithm, and Figure 11.3 shows the pseudocode. When node u is added to S and an incident edge $e = (u,v)$ is inspected, the algorithm needs to know whether $v \in S$. A bitvector could be used to

[2] Actually, Dijkstra also described this algorithm in his seminal 1959 paper on shortest paths [56]. Since Prim described the same algorithm two years earlier, it is usually named after him. However, the algorithm actually goes back to a 1930 paper by Jarník [98].

encode this information. If all edge costs are positive, we can reuse the d-array for this purpose. For any node v, $d[v] = 0$ indicates $v \in S$ and $d[v] > 0$ encodes $v \notin S$.

In addition to the space savings, this trick also avoids a comparison in the innermost loop. Observe that $c(e) < d[v]$ is only true if $d[v] > 0$, i.e., $v \notin S$, and e is an improved connection from v to S.

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 $O(n \log n + m)$.

Exercise 11.5. Dijkstra's algorithm for shortest paths can use monotone priority queues. Show that monotone priority queues do *not* suffice for the JP algorithm.

***Exercise 11.6 (average-case analysis of the JP algorithm).** Assume that the edge costs $1, \ldots, m$ 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 10.6.

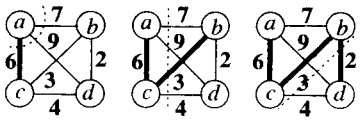


Fig. 11.2. A sequence of cuts (dotted lines) corresponding to the steps carried out by the Jarník–Prim algorithm with starting node a. The edges (a,c), (c,b), and (b,d) are added to the MST

Function *jpMST* : *Set* **of** *Edge*
  $d = \langle \infty, \ldots, \infty \rangle$: *NodeArray*$[1..n]$ **of** $\mathbb{R} \cup \{\infty\}$ // $d[v]$ is the distance of v from the tree
  parent : *NodeArray* **of** *NodeId* // parent$[v]$ is shortest edge between S and v
  Q : *NodePQ* // uses $d[\cdot]$ as priority
  $Q.insert(s)$ for some arbitrary $s \in V$
  while $Q \neq \emptyset$ **do**
    $u := Q.deleteMin$
    $d[u] := 0$ // $d[u] = 0$ encodes $u \in S$
    foreach *edge* $e = (u,v) \in E$ **do**
      if $c(e) < d[v]$ **then** // $c(e) < d[v]$ implies $d[v] > 0$ and hence $v \notin S$
        $d[v] := c(e)$
        $parent[v] := u$
        if $v \in Q$ **then** $Q.decreaseKey(v)$ **else** $Q.insert(v)$
    invariant $\forall v \in Q : d[v] = \min\{c((u,v)) : (u,v) \in E \wedge u \in S\}$
  return $\{(v, parent[v]) : v \in V \setminus \{s\}\}$

Fig. 11.3. The Jarník–Prim MST algorithm. Positive edge costs are assumed

11.3 Kruskal's Algorithm

The JP algorithm is probably the best general-purpose MST algorithm. Nevertheless, we shall now present an alternative algorithm, Kruskal's algorithm [116]. 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. Also, for sparse graphs with $m = O(n)$, its running time is competitive with the JP algorithm.

The pseudocode given in Fig. 11.4 is extremely compact. The algorithm scans over the edges of G in order of increasing cost and maintains a partial MST T; T is initially empty. The algorithm maintains the invariant that T can be extended to an MST. When an edge e is considered, it is either discarded or added to the MST. The decision is made on the basis of the cycle or cut property. The endpoints of e either belong to the same connected component of (V,T) or do not. In the former case, $T \cup e$ contains a cycle and e is an edge of maximum cost in this cycle. Since edges are considered in order of increasing cost, e can be discarded, by the cycle property. If e connects distinct components, e is a minimum-cost edge in the cut E' consisting of all edges connecting distinct components of (V,T); again, it is essential that edges are considered in order of increasing cost. We may therefore add e to T, by the cut property. The invariant is maintained. Figure 11.5 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.

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

Fig. 11.4. Kruskal's MST algorithm

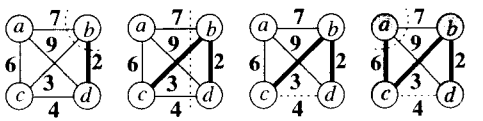


Fig. 11.5. In this example, Kruskal's algorithm first proves that (b,d) and (b,c) are MST edges using the cut property. Then (c,d) is excluded because it is the heaviest edge on the cycle $\langle b,c,d \rangle$, and, finally, (a,c) completes the MST

Exercise 11.7 (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 $O(V)$.
(*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 [182]).

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. 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 chooses one of its elements as its representative; the choice is made 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 *link*(i,j) applied to representatives of different blocks joins the blocks.

A simple solution is as follows. Each block is represented as a rooted tree[3], with the root being the representative of the block. Each element stores its parent in this tree (the array *parent*). We have self-loops at the roots.

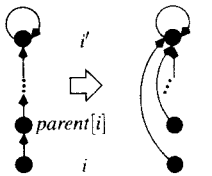
The implementation of *find*(i) is trivial. We follow parent pointers until we encounter a self-loop. The self-loop is located at the representative of i. The implementation of *link*(i,j) is equally simple. We simply make one representative the parent of the other. The latter has ceded its role to the former, which is now 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.

Exercise 11.8. Give an example of an n-node graph with $O(n)$ edges where a naive implementation of the union–find data structure without union by rank and path compression would lead to quadratic execution time for Kruskal's algorithm.

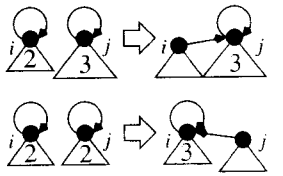
[3] Note that this tree may have a structure very different from the corresponding subtree in Kruskal's algorithm.

Class *UnionFind*$(n : \mathbb{N})$ // Maintain a partition of $1..n$
  parent $= \langle 1,2,\ldots,n \rangle$: *Array* $[1..n]$ **of** $1..n$
  rank $= \langle 0,\ldots,0 \rangle$: *Array* $[1..n]$ **of** $0..\log n$ // rank of representatives

Function *find*$(i : 1..n) : 1..n$
  if *parent*$[i] = i$ **then return** i
  else $i' :=$ *find*(*parent*$[i]$) // path compression
    parent$[i] := i'$
    return i'



Procedure *link*$(i,j : 1..n)$
  assert i and j are representatives of different blocks
  if *rank*$[i] <$ *rank*$[j]$ **then** *parent*$[i] := j$
  else
    parent$[j] := i$
    if *rank*$[i] =$ *rank*$[j]$ **then** *rank*$[i]$++



Procedure *union*$(i,j : 1..n)$
  if *find*$(i) \neq$ *find*(j) **then** *link*(*find*(i), *find*(j))

Fig. 11.6. An efficient union–find data structure that maintains a partition of the set $\{1,\ldots,n\}$

Therefore, Figure 11.6 introduces two optimizations. The first optimization limits the maximal 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 zero. When we link two representatives and their ranks are different, we make the representative of smaller rank a child of the representative of larger rank. When their ranks are the same, 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 11.9. Assume that the second optimization (described below) is not used. Show that the rank of a representative is the height of the tree rooted at it.

Theorem 11.3. *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. It therefore suffices to prove that the rank is bounded by $\log n$. We shall show 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 the root had at least 2^{k-1} descendants before the link 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 link operation. □

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 op-

eration *find*(i) redirect their parent pointers directly to the representative of i. In Fig. 11.6, 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. When the recursion stack is 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 11.10. Describe a nonrecursive implementation of *find*.

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

Theorem 11.4. *The union–find data structure of Fig. 11.6 performs m find and $n-1$ link operations in time* $O(m\alpha_T(m,n))$. *Here,*

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

where

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

Proof. The proof of this theorem is beyond the scope of this introductory text. We refer the reader to [186, 177]. □

You will probably find the formulae overwhelming. The function[4] 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 11.11. Estimate $A(5,1)$.

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

We close this section with an analysis of union–find with path compression but without union by rank. The analysis illustrates the power of path compression and also gives a glimpse of how Theorem 11.4 can be proved.

Theorem 11.5. *The union–find data structure with path compression but without union by rank processes m find and $n-1$ link operations in time* $O((m+n)\log n)$.

[4] The usage of the letter A is a reference to the logician Ackermann [3], who first studied a variant of this function in the late 1920s.

Proof. A link operation has cost one and adds one edge to the data structure. The total cost of all links is $O(n)$. The difficult part is to bound the cost of the finds. Note that the cost of a *find* is $O(1 + \text{number of edges constructed in path compression})$. So our task is to bound the total number of edges constructed.

In order to do so, every node v is assigned a weight $w(v)$ that is defined as the maximum number of descendants of v (including v) during the evolution of the data structure. Observe that $w(v)$ may increase as long as v is a representative, $w(v)$ reaches its maximal value when v ceases to be a representative (because it is linked to another representative), and $w(v)$ may decrease afterwards (because path compression removes a child of v to link it to a higher node). The weights are integers in the range $1..n$.

All edges that ever exist in our data structure go from nodes of smaller weight to nodes of larger weight. We define the span of an edge as the difference between the weights of its endpoints. We say that an edge has a class i if its span lies in the range $2^i..2^{i+1}-1$. The class of any edge lies between 0 and $\lceil \log n \rceil$.

Consider a particular node x. The first edge out of x is created when x ceases to be a representative. Also, x receives a new parent whenever a find operation passes through the edge $(x, parent(x))$ and this edge is not the last edge traversed by the find. The new edge out of x has a larger span.

We account for the edges out of x as follows. The first edge is charged to the union operation. Consider now any edge $e = (x,y)$ and the find operation which destroys it. Let e have class i. The find operation traverses a path of edges. If e is the last (= topmost) edge of class i traversed by the find, we charge the construction of the new edge out of x to the find operation; otherwise, we charge it to x. Observe that in this way, at most $1 + \lceil \log n \rceil$ edges are charged to any find operation (because there are only $1 + \lceil \log n \rceil$ different classes of edges). If the construction of the new edge out of x is charged to x, there is another edge $e' = (x',y')$ in class i following e on the find path. Also, the new edge out of x has a span at least as large as the sum of the spans of e and e', since it goes to an ancestor (not necessarily proper) of y'. Thus the new edge out of x has a span of at least $2^i + 2^i = 2^{i+1}$ and hence is in class $i+1$ or higher. We conclude that at most one edge in each class is charged to each node x. Thus the total number of edges constructed is at most $n + (n+m)(1 + \lceil \log n \rceil)$, and the time bound follows. □

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 [50].

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. 5.7. 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 on v. We add e, and somehow 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 on v to u, i.e., any edge (v,w) now becomes an edge (u,w). Figure 11.7 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' [50]:

for $v := 1$ **to** $n - n'$ **do**
  find the lightest edge (u,v) *incident on* v *and contract it*

Figure 11.7 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 on v and how we relink the other edges incident on v, i.e., how we inform the neighbors of v 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

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

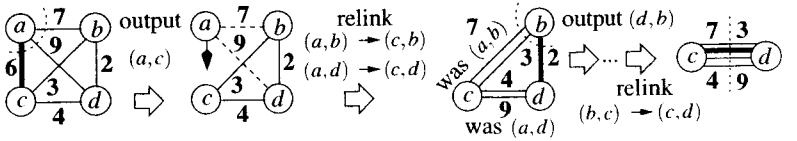


Fig. 11.7. An execution of Sibeyn's algorithm with $n' = 2$. The edge $(c,a,6)$ is the cheapest edge incident on a. We add it to the MST and merge a into c. The edge $(a,b,7)$ becomes an edge $(c,b,7)$ and $(a,d,9)$ becomes $(c,d,9)$. In the new graph, $(d,b,2)$ is the cheapest edge incident on b. We 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 weight 3, 4, 7, and 9, respectively

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 $e = (u,v) \in E$ **do**
    $Q.insert(\min\{\pi(u),\pi(v)\}, \max\{\pi(u),\pi(v)\}, c(e), u, v))$
  current $:= 0$ // we are just before processing node 1
  loop
    $(u,v,c,u_0,v_0) := \min Q$ // next edge
    if *current* $\neq u$ **then** // new node
      if $u = n - n' + 1$ **then** *break loop* // node reduction completed
      Q.deleteMin
      output (u_0,v_0) // the original endpoints define an MST edge
      $(current, relinkTo) := (u,v)$ // prepare for relinking remaining u-edges
    else if $v \neq relinkTo$ **then**
      $Q.insert((\min\{v, relinkTo\}, \max\{v, relinkTo\}, c, u_0, v_0))$ // relink
  $S := sort(Q)$ // sort by increasing edge weight
  apply semiexternal Kruskal to S

Fig. 11.8. Sibeyn's MST algorithm

in the priority queue. The ordering is lexicographic by the first and third components, i.e., edges are first ordered by the lower-numbered endpoint and then according to weight. The algorithm operates in phases. In each phase, we select all edges incident on the *current* node. The lightest edge (= first edge delivered by the queue), say $(current, relinkTo)$, is added to the MST, and all others are relinked. In order to relink an edge $(current, z, c, u_0, v_0)$ with $z \neq RelinkTo$, we add $(\min(z, RelinkTo), \max(z, RelinkTo), c, u_0, v_0)$ to the queue.

Figure 11.8 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 $(\min\{\pi(u),\pi(v)\}, \max\{\pi(u),\pi(v)\}, c(e), u, v))$ 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 11.6. *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* $O(\mathrm{sort}(m\ln(n/n')))$.

Proof. From Sect. 6.3, we know that an external-memory priority queue can execute K queue operations using $O(\mathrm{sort}(K))$ I/Os. Also, the semiexternal Kruskal step requires $O(\mathrm{sort}(m))$ 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. By the linearity of expectations, we have $\mathrm{E}[\sum_{1\le i\le n-n'} X_i] = \sum_{1\le i\le n-n'} \mathrm{E}[X_i]$. The number of edges in the contracted graph is at most m, so that the average degree of a graph with $n-i+1$ remaining nodes is at most $2m/(n-i+1)$. We obtain.

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

where $H_n := \sum_{1\le i\le n} 1/i = \ln n + \Theta(1)$ is the n-th harmonic number (see (A.12)). □

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

11.6 Applications

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

11.6.1 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 will, in general, have some of the uninhabited islands in the solution.

The Steiner tree problem is **NP**-complete (see Sect. 2.10). We shall show how to construct a solution which is within a factor of two of the optimum. We construct an auxiliary complete graph with node set S: for any pair u and v of nodes in S, the cost of the edge (u,v) in the auxiliary graph is their shortest-path distance in G. Let T_A be an MST of the auxiliary graph. We obtain a Steiner tree of G by replacing every edge of T_A by the path it represents in G. The resulting subgraph of G may contain cycles. We delete edges from cycles until the remaining subgraph is cycle-free. The cost of the resulting Steiner tree is at most the cost of T_A.

Theorem 11.7. *The algorithm above constructs a Steiner tree which has at most twice the cost of an optimal Steiner tree.*

Proof. The algorithm constructs a Steiner tree of cost at most $c(T_A)$. It therefore suffices to show that $c(T_A) \le 2c(T_{\mathrm{opt}})$, where T_{opt} is a minimum Steiner tree for S in G. To this end, it suffices to show that the auxiliary graph has a spanning tree of cost $2c(T_{\mathrm{opt}})$. Figure 11.9 indicates how to construct such a spanning tree. "Walking once around the Steiner tree" defines a cycle in G of cost $2c(T_{\mathrm{opt}})$; observe that every edge in T_{opt} occurs exactly twice in this path. Deleting the nodes outside S in this path gives us a cycle in the auxiliary graph. The cost of this path is at most $2c(T_{\mathrm{opt}})$, because edge costs in the auxiliary graph are shortest-path distances in G. The cycle in the auxiliary graph spans S, and therefore the auxiliary graph has a spanning tree of cost at most $2c(T_{\mathrm{opt}})$. □

Exercise 11.13. Improve the above bound to $2(1-1/|S|)$ times the optimum.

The algorithm can be implemented to run in time $O(m+n\log n)$ [126]. Algorithms with better approximation ratios exist [163].

Exercise 11.14. Outline an implementation of the algorithm above and analyze its running time.

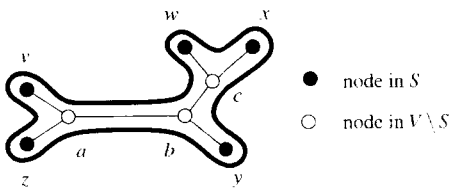


Fig. 11.9. Once around the tree. We have $S = \{v,w,x,y,z\}$, and the minimum Steiner tree is shown. The Steiner tree also involves the nodes a, b, and c in $V \setminus S$. Walking once around the tree yields the cycle $\langle v,a,b,c,w,c,x,c,b,y,b,a,z,a,v\rangle$. It maps into the cycle $\langle v,w,x,y,z,v\rangle$ in the auxiliary graph

11.6.2 Traveling Salesman Tours

The traveling salesman problem is one of the most intensively studied optimization problems [197, 117, 13]: 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. This is the path a traveling salesman would want to take whose goal it is to visit all nodes of the graph. 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. There is then always an optimal round trip which visits no node twice (because leaving it out would not increase the cost).

Theorem 11.8. *Let C_{opt} and C_{MST} be the cost 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. 11.9 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. □

In the remainder of this section, we shall briefly outline a technique for improving the lower bound of Theorem 11.8. We need two additional concepts: 2-trees and node potentials. Let G' be obtained from G by deleting node 1 and the edges incident on it. A minimum 2-tree consists of the two cheapest edges incident on node 1 and an MST of G'. Since deleting the two edges incident on node 1 from a tour C yields a spanning tree of G', we have $C_2 \leq C_{\text{opt}}$, where C_2 is the minimum cost of a 2-tree. A node potential is any real-valued function π defined on the nodes of G. Any node potential yields a modified cost function c_π by defining

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

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 on any node. Let T_π be a minimum 2-tree with respect to c_π. Then

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

and hence

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

This lower bound is known as the Held–Karp lower bound [88, 89]. 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 2-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 on nodes of degree larger than two a little more expensive and the edges incident on nodes of degree one 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 zero with increasing iteration number, and $\deg(v, T_\pi)$ is the degree of v in T_π. We next compute an optimal 2-tree with respect to π' and hope that it will yield a better lower bound.

11.7 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 [142] 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. 11.4. 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.

11.7.1 C++

LEDA [118] uses Kruskal's algorithm for computing MSTs. The union–find data structure is called *partition* in LEDA. The Boost graph library [27] gives a choice between Kruskal's algorithm and the JP algorithm. Boost offers no public access to the union–find data structure.

11.7.2 Java

JDSL [78] uses the JP algorithm.

11.8 Historical Notes and Further Findings

The oldest MST algorithm is based on the cut property and uses edge contractions. *Boruvka's algorithm* [28, 148] goes back to 1926 and hence represents one of the oldest graph algorithms. The algorithm operates in phases, and identifies many MST edges in each phase. In a phase, each node identifies the lightest incident edge. These

edges are added to the MST (here it is assumed that the edge costs are pairwise distinct) and then contracted. Each phase can be implemented to run in time $O(m)$. Since a phase at least halves the number of remaining nodes, only a single node is left after $O(\log n)$ phases, and hence the total running time is $O(m \log n)$. Boruvka's algorithm is not often used, because it is somewhat complicated to implement. It is nevertheless important as a basis for parallel MST algorithms.

There is a randomized linear-time MST algorithm that uses phases of Boruvka's algorithm to reduce the number of nodes [105, 111]. 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 rather difficult to implement efficiently. But, at least for rather dense graphs, this approach can yield a practical improvement [108]. The linear-time algorithm can also be parallelized [84]. An adaptation to 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 [35, 155] has the interesting property that it has optimal worst-case complexity, although it is not exactly known what this complexity is. Hence, if you 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.

Minimum spanning trees define a single path between any pair of nodes. Interestingly, this path is a *bottleneck shortest path* [8, 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 [8, 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 [46] 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 discussed the application of MSTs to the Steiner tree and the traveling salesman problem. We refer the reader to the books [8, 13, 117, 115, 200] for more information about these and related problems.