Trees

A *tree* is an undirected graph that is connected and acyclic. Much of what makes trees so useful is the simplicity of their structure. For instance,

Property 2 *A tree on n nodes has $n - 1$ edges.*

This can be seen by building the tree one edge at a time, starting from an empty graph. Initially each of the n nodes is disconnected from the others, in a connected component by itself. As edges are added, these components merge. Since each edge unites two different components, exactly $n - 1$ edges are added by the time the tree is fully formed.

In a little more detail: When a particular edge $\{u, v\}$ comes up, we can be sure that u and v lie in separate connected components, for otherwise there would already be a path between them and this edge would create a cycle. Adding the edge then merges these two components, thereby reducing the total number of connected components by one. Over the course of this incremental process, the number of components decreases from n to one, meaning that $n - 1$ edges must have been added along the way.

The converse is also true.

Property 3 *Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.*

We just need to show that G is acyclic. One way to do this is to run the following iterative procedure on it: while the graph contains a cycle, remove one edge from this cycle. The process terminates with some graph $G' = (V, E'), E' \subseteq E$, which is acyclic and, by Property 1 (from page 139), is also connected. Therefore G' is a tree, whereupon $|E'| = |V| - 1$ by Property 2. So $E' = E$, no edges were removed, and G was acyclic to start with.

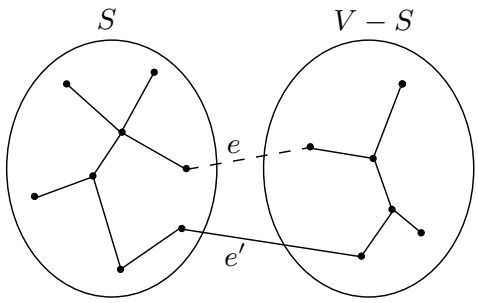
In other words, we can tell whether a connected graph is a tree just by counting how many edges it has. Here's another characterization.

Property 4 *An undirected graph is a tree if and only if there is a unique path between any pair of nodes.*

In a tree, any two nodes can only have one path between them; for if there were two paths, the union of these paths would contain a cycle.

On the other hand, if a graph has a path between any two nodes, then it is connected. If these paths are unique, then the graph is also acyclic (since a cycle has two paths between any pair of nodes).

Figure 5.2 $T \cup \{e\}$. The addition of e (dotted) to T (solid lines) produces a cycle. This cycle must contain at least one other edge, shown here as e', across the cut $(S, V - S)$.



5.1.2 The cut property

Say that in the process of building a minimum spanning tree (MST), we have already chosen some edges and are so far on the right track. Which edge should we add next? The following lemma gives us a lot of flexibility in our choice.

Cut property *Suppose edges X are part of a minimum spanning tree of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V - S$, and let e be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.*

A *cut* is any partition of the vertices into two groups, S and $V - S$. What this property says is that it is always safe to add the lightest edge across any cut (that is, between a vertex in S and one in $V - S$), provided X has no edges across the cut.

Let's see why this holds. Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove. So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle. This cycle must also have some other edge e' across the cut $(S, V - S)$ (Figure 8.3). If we now remove this edge, we are left with $T' = T \cup \{e\} - \{e'\}$, which we will show to be a tree. T' is connected by Property 1, since e' is a cycle edge. And it has the same number of edges as T; so by Properties 2 and 3, it is also a tree.

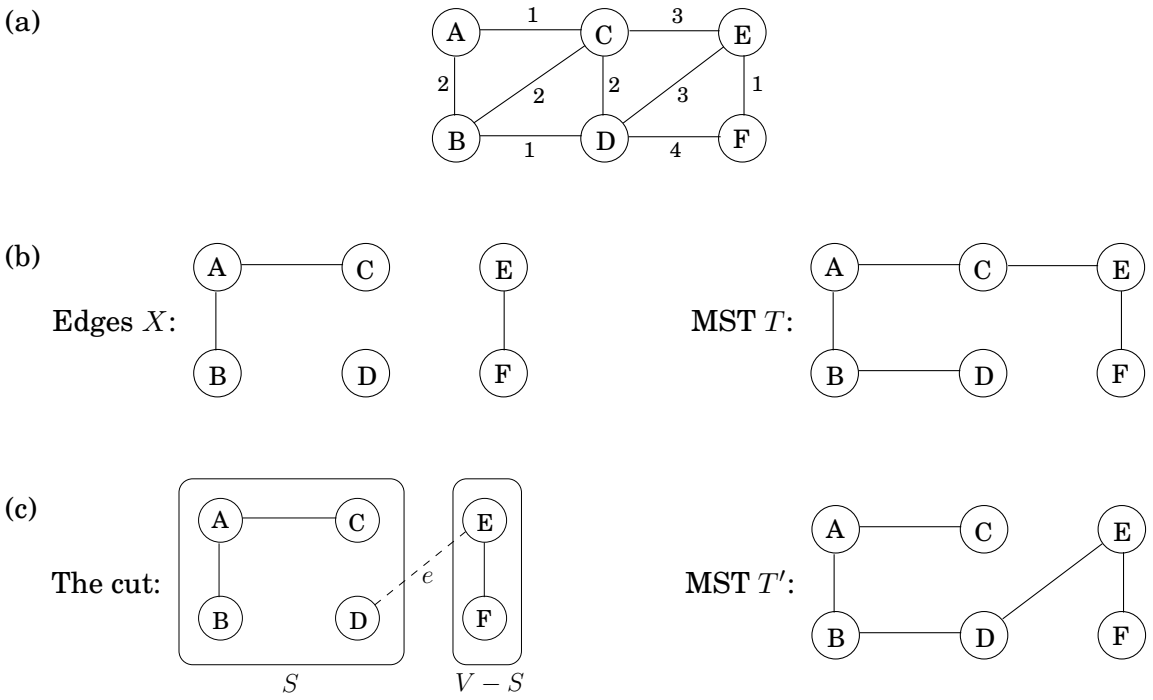
Moreover, T' is a minimum spanning tree. Compare its weight to that of T:

$$\text{weight}(T') \;=\; \text{weight}(T) + w(e) - w(e').$$

Both e and e' cross between S and $V - S$, and e is specifically the lightest edge of this type. Therefore $w(e) \leq w(e')$, and $\text{weight}(T') \leq \text{weight}(T)$. Since T is an MST, it must be the case that $\text{weight}(T') = \text{weight}(T)$ and that T' is also an MST.

Figure 5.3 shows an example of the cut property. Which edge is e'?

Figure 5.3 The cut property at work. (a) An undirected graph. (b) Set X has three edges, and is part of the MST T on the right. (c) If $S = \{A, B, C, D\}$, then one of the minimum-weight edges across the cut $(S, V - S)$ is $e = \{D, E\}$. $X \cup \{e\}$ is part of MST T', shown on the right.



5.1.3 Kruskal's algorithm

We are ready to justify Kruskal's algorithm. At any given moment, the edges it has already chosen form a partial solution, a collection of connected components each of which has a tree structure. The next edge e to be added connects two of these components; call them T_1 and T_2. Since e is the lightest edge that doesn't produce a cycle, it is certain to be the lightest edge between T_1 and $V - T_1$ and therefore satisfies the cut property.

Now we fill in some implementation details. At each stage, the algorithm chooses an edge to add to its current partial solution. To do so, it needs to test each candidate edge $u - v$ to see whether the endpoints u and v lie in different components; otherwise the edge produces a cycle. And once an edge is chosen, the corresponding components need to be merged. What kind of data structure supports such operations?

We will model the algorithm's state as a collection of *disjoint sets*, each of which contains the nodes of a particular component. Initially each node is in a component by itself:

`makeset`(x): create a singleton set containing just x.

We repeatedly test pairs of nodes to see if they belong to the same set.

`find`(x): to which set does x belong?

Figure 5.4 Kruskal's minimum spanning tree algorithm.

```
procedure kruskal(G, w)
Input:    A connected undirected graph G = (V, E) with edge weights w_e
Output:   A minimum spanning tree defined by the edges X

for all u ∈ V:
   makeset(u)

X = {}
Sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u, v} to X
      union(u, v)
```

And whenever we add an edge, we are merging two components.

`union`(x, y): merge the sets containing x and y.

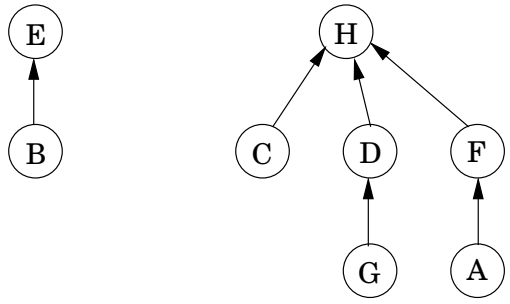
The final algorithm is shown in Figure 5.4. It uses $|V|$ `makeset`, $2|E|$ `find`, and $|V| - 1$ `union` operations.

5.1.4 A data structure for disjoint sets

Union by rank

One way to store a set is as a directed tree (Figure 5.5). Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree. This root element is a convenient *representative*, or *name*, for the set. It is distinguished from the other elements by the fact that its parent pointer is a self-loop.

Figure 5.5 A directed-tree representation of two sets $\{B, E\}$ and $\{A, C, D, F, G, H\}$.



In addition to a parent pointer π, each node also has a *rank* that, for the time being, should be interpreted as the height of the subtree hanging from that node.

```
procedure makeset(x)
π(x) = x
rank(x) = 0

function find(x)
while x ≠ π(x): x = π(x)
return x
```

As can be expected, `makeset` is a constant-time operation. On the other hand, `find` follows parent pointers to the root of the tree and therefore takes time proportional to the height of the tree. The tree actually gets built via the third operation, `union`, and so we must make sure that this procedure keeps trees shallow.

Merging two sets is easy: make the root of one point to the root of the other. But we have a choice here. If the representatives (roots) of the sets are r_x and r_y, do we make r_x point to r_y or the other way around? Since tree height is the main impediment to computational efficiency, a good strategy is to *make the root of the shorter tree point to the root of the taller tree*. This way, the overall height increases only if the two trees being merged are equally tall. Instead of explicitly computing heights of trees, we will use the *rank* numbers of their root nodes—which is why this scheme is called *union by rank*.

```
procedure union(x, y)
r_x = find(x)
r_y = find(y)
if r_x = r_y: return
if rank(r_x) > rank(r_y):
    π(r_y) = r_x
else:
    π(r_x) = r_y
    if rank(r_x) = rank(r_y): rank(r_y) = rank(r_y) + 1
```

See Figure 5.6 for an example.

By design, the *rank* of a node is exactly the height of the subtree rooted at that node. This means, for instance, that as you move up a path toward a root node, the *rank* values along the way are strictly increasing.

Property 1 *For any x, rank$(x) <$ rank$(\pi(x))$.*

A root node with rank k is created by the merger of two trees with roots of rank $k-1$. It follows by induction (try it!) that

Property 2 *Any root node of rank k has at least 2^k nodes in its tree.*

This extends to internal (nonroot) nodes as well: a node of rank k has at least 2^k descendants. After all, any internal node was once a root, and neither its rank nor its set of descendants has changed since then. Moreover, different rank-k nodes cannot have common descendants, since by Property 1 any element has at most one ancestor of rank k. Which means

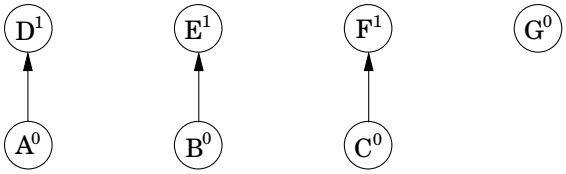
Property 3 *If there are n elements overall, there can be at most $n/2^k$ nodes of rank k.*

This last observation implies, crucially, that the maximum rank is $\log n$. Therefore, all the trees have height $\leq \log n$, and this is an upper bound on the running time of `find` and `union`.

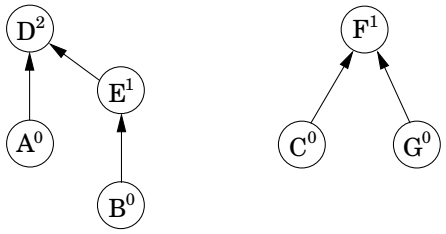
Figure 5.6 A sequence of disjoint-set operations. Superscripts denote rank.

After `makeset`(A), `makeset`(B), ..., `makeset`(G):

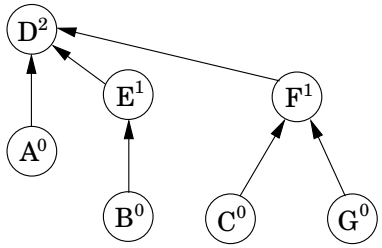
After `union`(A, D), `union`(B, E), `union`(C, F):



After `union`(C, G), `union`(E, A):



After `union`(B, G):



Path compression

With the data structure as presented so far, the total time for Kruskal's algorithm becomes $O(|E| \log |V|)$ for sorting the edges (remember, $\log |E| \approx \log |V|$) plus another $O(|E| \log |V|)$ for the `union` and `find` operations that dominate the rest of the algorithm. So there seems to be little incentive to make our data structure any more efficient.

But what if the edges are given to us sorted? Or if the weights are small (say, $O(|E|)$) so that sorting can be done in linear time? Then the data structure part becomes the bottleneck, and it is useful to think about improving its performance beyond $\log n$ per operation. As it turns out, the improved data structure is useful in many other applications.

But how can we perform `union`'s and `find`'s faster than $\log n$? The answer is, by being a little more careful to maintain our data structure in good shape. As any housekeeper knows, a little extra effort put into routine maintenance can pay off handsomely in the long run, by forestalling major calamities. We have in mind a particular maintenance operation for our union-find data structure, intended to keep the trees short— during each `find`, when a series of parent pointers is followed up to the root of a tree, we will change all these pointers so that they point directly to the root (Figure 5.7). This *path compression* heuristic only slightly increases the time needed for a `find` and is easy to code.

```
function find(x)
if x ≠ π(x): π(x) = find(π(x))
return π(x)
```

The benefit of this simple alteration is long-term rather than instantaneous and thus necessitates a particular kind of analysis: we need to look at *sequences* of `find` and `union` operations, starting from an empty data structure, and determine the average time per operation. This *amortized cost* turns out to be just barely more than $O(1)$, down from the earlier $O(\log n)$.

Think of the data structure as having a "top level" consisting of the root nodes, and below it, the insides of the trees. There is a division of labor: `find` operations (with or without path compression) only touch the insides of trees, whereas `union`'s only look at the top level. Thus path compression has no effect on `union` operations and leaves the top level unchanged.

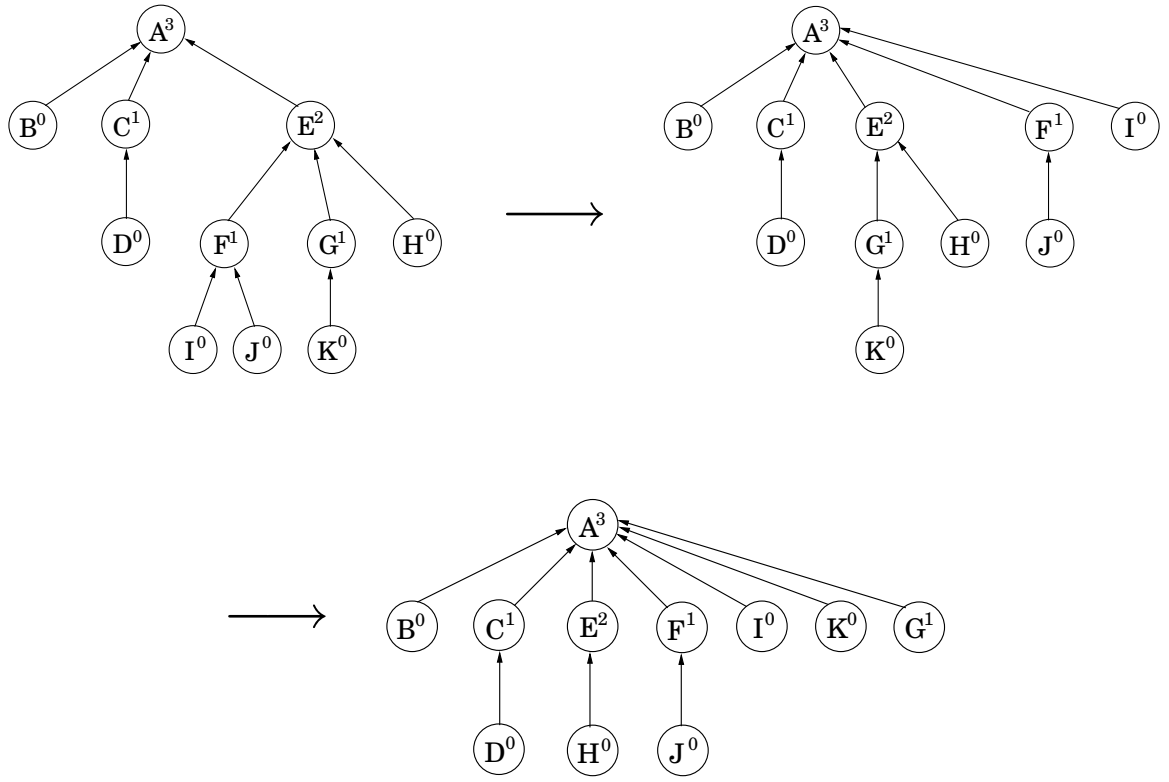
We now know that the ranks of root nodes are unaltered, but what about *nonroot* nodes? The key point here is that once a node ceases to be a root, it never resurfaces, and its rank is forever fixed. Therefore the ranks of all nodes are unchanged by path compression, even though these numbers can no longer be interpreted as tree heights. In particular, properties 1–3 (from page 145) still hold.

If there are n elements, their rank values can range from 0 to $\log n$ by Property 3. Let's divide the nonzero part of this range into certain carefully chosen intervals, for reasons that will soon become clear:

$$\{1\}, \{2\}, \{3, 4\}, \{5, 6, \ldots, 16\}, \{17, 18, \ldots, 2^{16} = 65536\}, \{65537, 65538, \ldots, 2^{65536}\}, \ldots$$

Each group is of the form $\{k + 1, k + 2, \ldots, 2^k\}$, where k is a power of 2. The number of groups is $\log^* n$, which is defined to be the number of successive $\log$ operations that need to be applied

Figure 5.7 The effect of path compression: `find`(I) followed by `find`(K).



to n to bring it down to 1 (or below 1). For instance, $\log^* 1000 = 4$ since $\log\log\log\log 1000 \leq 1$. In practice there will just be the first five of the intervals shown; more are needed only if $n \geq 2^{65536}$, in other words never.

In a sequence of `find` operations, some may take longer than others. We'll bound the overall running time using some creative accounting. Specifically, we will give each node a certain amount of pocket money, such that the total money doled out is at most $n \log^* n$ dollars. We will then show that each `find` takes $O(\log^* n)$ steps, plus some additional amount of time that can be "paid for" using the pocket money of the nodes involved—one dollar per unit of time. Thus the overall time for m `find`'s is $O(m \log^* n)$ plus at most $O(n \log^* n)$.

In more detail, a node receives its allowance as soon as it ceases to be a root, at which point its rank is fixed. If this rank lies in the interval $\{k+1, \ldots, 2^k\}$, the node receives 2^k dollars. By Property 3, the number of nodes with rank $> k$ is bounded by

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \;\leq\; \frac{n}{2^k}.$$

Therefore the total money given to nodes in this particular interval is at most n dollars, and since there are $\log^* n$ intervals, the total money disbursed to all nodes is $\leq n \log^* n$.

Now, the time taken by a specific `find` is simply the number of pointers followed. Consider the ascending rank values along this chain of nodes up to the root. Nodes x on the chain fall into two categories: either the rank of $\pi(x)$ is in a higher interval than the rank of x, or else it lies in the same interval. There are at most $\log^* n$ nodes of the first type (do you see why?), so the work done on them takes $O(\log^* n)$ time. The remaining nodes—whose parents' ranks are in the same interval as theirs—have to pay a dollar out of their pocket money for their processing time.

This only works if the initial allowance of each node x is enough to cover all of its payments in the sequence of `find` operations. Here's the crucial observation: each time x pays a dollar, its parent changes to one of higher rank. Therefore, if x's rank lies in the interval $\{k + 1, \ldots, 2^k\}$, it has to pay at most 2^k dollars before its parent's rank is in a higher interval; whereupon it never has to pay again.

A randomized algorithm for minimum cut

We have already seen that spanning trees and cuts are intimately related. Here is another connection. Let's remove the last edge that Kruskal's algorithm adds to the spanning tree; this breaks the tree into two components, thus defining a cut $(S, \overline{S})$ in the graph. What can we say about this cut? Suppose the graph we were working with was unweighted, and that its edges were ordered uniformly at random for Kruskal's algorithm to process them. Here is a remarkable fact: with probability at least $1/n^2$, $(S, \overline{S})$ is the minimum cut in the graph, where the size of a cut $(S, \overline{S})$ is the number of edges crossing between S and $\overline{S}$. This means that repeating the process $O(n^2)$ times and outputting the smallest cut found yields the minimum cut in G with high probability: an $O(mn^2 \log n)$ algorithm for unweighted minimum cuts. Some further tuning gives the $O(n^2 \log n)$ minimum cut algorithm, invented by David Karger, which is the fastest known algorithm for this important problem.

So let us see why the cut found in each iteration is the minimum cut with probability at least $1/n^2$. At any stage of Kruskal's algorithm, the vertex set V is partitioned into connected components. The only edges eligible to be added to the tree have their two endpoints in distinct components. The number of edges incident to each component must be at least C, the size of the minimum cut in G (since we could consider a cut that separated this component from the rest of the graph). So if there are k components in the graph, the number of eligible edges is at least $kC/2$ (each of the k components has at least C edges leading out of it, and we need to compensate for the double-counting of each edge). Since the edges were randomly ordered, the chance that the next eligible edge in the list is from the minimum cut is at most $C/(kC/2) = 2/k$. Thus, with probability at least $1 - 2/k = (k-2)/k$, the choice leaves the minimum cut intact. But now the chance that Kruskal's algorithm leaves the minimum cut intact all the way up to the choice of the last spanning tree edge is at least

$$\frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3} = \frac{1}{n(n-1)}.$$

5.1.5 Prim's algorithm

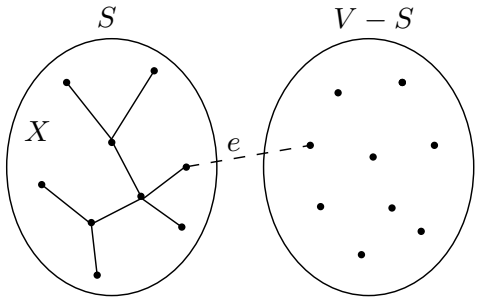
Let's return to our discussion of minimum spanning tree algorithms. What the cut property tells us in most general terms is that any algorithm conforming to the following greedy schema is guaranteed to work.

```
X = { } (edges picked so far)
repeat until |X| = |V| − 1:
  pick a set S ⊂ V for which X has no edges between S and V − S
  let e ∈ E be the minimum-weight edge between S and V − S
  X = X ∪ {e}
```

A popular alternative to Kruskal's algorithm is Prim's, in which the intermediate set of edges X always forms a subtree, and S is chosen to be the set of this tree's vertices.

On each iteration, the subtree defined by X *grows* by one edge, namely, the lightest edge between a vertex in S and a vertex outside S (Figure 5.8). We can equivalently think of S as

Figure 5.8 Prim's algorithm: the edges X form a tree, and S consists of its vertices.



growing to include the vertex $v \notin S$ of smallest `cost`:

$$\texttt{cost}(v) \;=\; \min_{u \in S} w(u, v).$$

This is strongly reminiscent of Dijkstra's algorithm, and in fact the pseudocode (Figure 5.9) is almost identical. The only difference is in the key values by which the priority queue is ordered. In Prim's algorithm, the value of a node is the weight of the lightest incoming edge from set S, whereas in Dijkstra's it is the length of an entire path to that node from the starting point. Nonetheless, the two algorithms are similar enough that they have the same running time, which depends on the particular priority queue implementation.

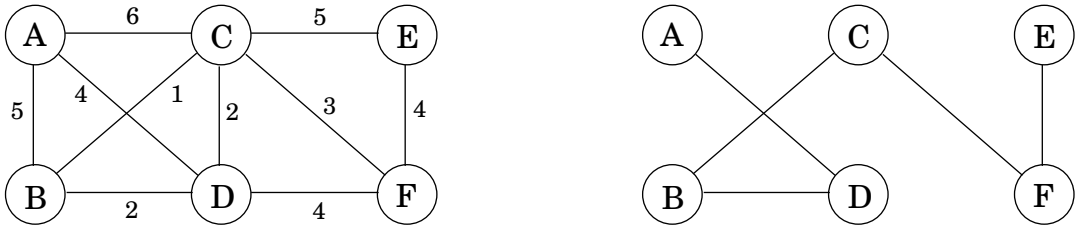
Figure 5.9 shows Prim's algorithm at work, on a small six-node graph. Notice how the final MST is completely specified by the `prev` array.

Figure 5.9 *Top:* Prim's minimum spanning tree algorithm. *Below:* An illustration of Prim's algorithm, starting at node A. Also shown are a table of cost/prev values, and the final MST.

```
procedure prim(G, w)
Input:    A connected undirected graph G = (V, E) with edge weights w_e
Output:   A minimum spanning tree defined by the array prev

for all u ∈ V:
   cost(u) = ∞
   prev(u) = nil
Pick any initial node u_0
cost(u_0) = 0

H = makequeue(V)     (priority queue, using cost-values as keys)
while H is not empty:
   v = deletemin(H)
   for each {v, z} ∈ E:
      if cost(z) > w(v, z):
         cost(z) = w(v, z)
         prev(z) = v
         decreasekey(H, z)
```



Set S	A	B	C	D	E	F
$\{\}$	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		$5/A$	$6/A$	$4/A$	∞/nil	∞/nil
A, D		$2/D$	$2/D$		∞/nil	$4/D$
A, D, B			$1/B$		∞/nil	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

5.2 Huffman encoding

In the MP3 audio compression scheme, a sound signal is encoded in three steps.

1. It is digitized by sampling at regular intervals, yielding a sequence of real numbers $s_1, s_2, \ldots, s_T$. For instance, at a rate of $44{,}100$ samples per second, a 50-minute symphony would correspond to $T = 50 \times 60 \times 44{,}100 \approx 130$ million measurements.[1]

2. Each real-valued sample s_t is *quantized*: approximated by a nearby number from a finite set Γ. This set is carefully chosen to exploit human perceptual limitations, with the intention that the approximating sequence is indistinguishable from $s_1, s_2, \ldots, s_T$ by the human ear.

3. The resulting string of length T over alphabet Γ is encoded in binary.

It is in the last step that Huffman encoding is used. To understand its role, let's look at a toy example in which T is 130 million and the alphabet Γ consists of just four values, denoted by the symbols A, B, C, D. What is the most economical way to write this long string in binary? The obvious choice is to use 2 bits per symbol—say codeword 00 for A, 01 for B, 10 for C, and 11 for D—in which case 260 megabits are needed in total. Can there possibly be a better encoding than this?

In search of inspiration, we take a closer look at our particular sequence and find that the four symbols are not equally abundant.

Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Is there some sort of *variable-length encoding*, in which just *one* bit is used for the frequently occurring symbol A, possibly at the expense of needing three or more bits for less common symbols?

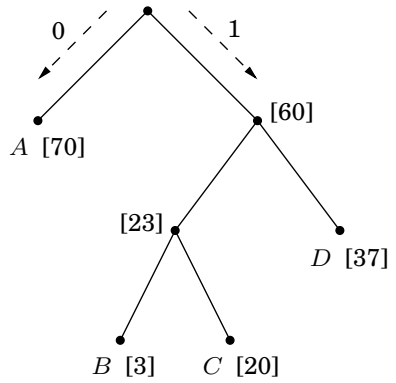
A danger with having codewords of different lengths is that the resulting encoding may not be uniquely decipherable. For instance, if the codewords are $\{0, 01, 11, 001\}$, the decoding of strings like 001 is ambiguous. We will avoid this problem by insisting on the *prefix-free* property: no codeword can be a prefix of another codeword.

Any prefix-free encoding can be represented by a *full* binary tree—that is, a binary tree in which every node has either zero or two children—where the symbols are at the leaves, and where each codeword is generated by a path from root to leaf, interpreting left as 0 and right as 1 (Exercise 5.28). Figure 5.10 shows an example of such an encoding for the four symbols A, B, C, D. Decoding is unique: a string of bits is decrypted by starting at the root, reading the string from left to right to move downward, and, whenever a leaf is reached, outputting the corresponding symbol and returning to the root. It is a simple scheme and pays off nicely

[1] For stereo sound, two channels would be needed, doubling the number of samples.

Figure 5.10 A prefix-free encoding. Frequencies are shown in square brackets.

Symbol	Codeword
A	0
B	100
C	101
D	11



for our toy example, where (under the codes of Figure 5.10) the total size of the binary string drops to 213 megabits, a 17% improvement.

In general, how do we find the optimal coding tree, given the frequencies $f_1, f_2, \ldots, f_n$ of n symbols? To make the problem precise, we want a tree whose leaves each correspond to a symbol and which minimizes the overall length of the encoding,

$$\text{cost of tree} \quad = \quad \sum_{i=1}^{n} f_i \cdot (\text{depth of } i\text{th symbol in tree})$$

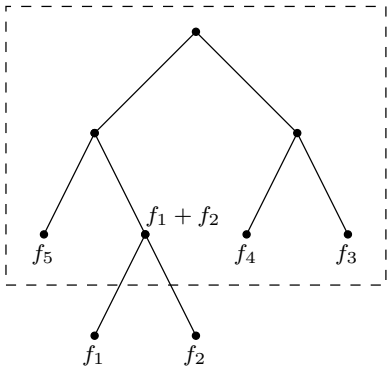
(the number of bits required for a symbol is exactly its depth in the tree).

There is another way to write this cost function that is very helpful. Although we are only given frequencies for the leaves, we can define the frequency of any *internal* node to be the sum of the frequencies of its descendant leaves; this is, after all, the number of times the internal node is visited during encoding or decoding. During the encoding process, each time we move down the tree, one bit gets output for every nonroot node through which we pass. So the total cost—the total number of bits which are output—can also be expressed thus:

> The cost of a tree is the sum of the frequencies of all leaves and internal nodes, except the root.

The first formulation of the cost function tells us that *the two symbols with the smallest frequencies must be at the bottom of the optimal tree*, as children of the lowest internal node (this internal node has two children since the tree is *full*). Otherwise, swapping these two symbols with whatever is lowest in the tree would improve the encoding.

This suggests that we start constructing the tree *greedily*: find the two symbols with the smallest frequencies, say i and j, and make them children of a new node, which then has frequency $f_i + f_j$. To keep the notation simple, let's just assume these are f_1 and f_2. By the second formulation of the cost function, any tree in which f_1 and f_2 are sibling-leaves has cost $f_1 + f_2$ plus the cost for a tree with $n - 1$ leaves of frequencies $(f_1 + f_2), f_3, f_4, \ldots, f_n$:



The latter problem is just a smaller version of the one we started with. So we pull f_1 and f_2 off the list of frequencies, insert $(f_1 + f_2)$, and loop. The resulting algorithm can be described in terms of priority queue operations (as defined on page 120) and takes $O(n \log n)$ time if a binary heap (Section 4.5.2) is used.

```
procedure Huffman(f)
Input:  An array f[1···n] of frequencies
Output:  An encoding tree with n leaves

let H be a priority queue of integers, ordered by f
for i = 1 to n:  insert(H, i)
for k = n+1 to 2n−1:
  i = deletemin(H),  j = deletemin(H)
  create a node numbered k with children i, j
  f[k] = f[i] + f[j]
  insert(H, k)
```

Returning to our toy example: can you tell if the tree of Figure 5.10 is optimal?

Entropy

The annual county horse race is bringing in three thoroughbreds who have never competed against one another. Excited, you study their past 200 races and summarize these as probability distributions over four outcomes: `first` ("first place"), `second`, `third`, and `other`.

Outcome	Aurora	Whirlwind	Phantasm
`first`	0.15	0.30	0.20
`second`	0.10	0.05	0.30
`third`	0.70	0.25	0.30
`other`	0.05	0.40	0.20

Which horse is the most predictable? One quantitative approach to this question is to look at *compressibility*. Write down the history of each horse as a string of 200 values (`first`, `second`, `third`, `other`). The total number of bits needed to encode these track-record strings can then be computed using Huffman's algorithm. This works out to 290 bits for Aurora, 380 for Whirlwind, and 420 for Phantasm (check it!). Aurora has the shortest encoding and is therefore in a strong sense the most predictable.

The inherent unpredictability, or *randomness*, of a probability distribution can be measured by the extent to which it is possible to compress data drawn from that distribution.

$$\text{more compressible} \quad \equiv \quad \text{less random} \quad \equiv \quad \text{more predictable}$$

Suppose there are n possible outcomes, with probabilities $p_1, p_2, \ldots, p_n$. If a sequence of m values is drawn from the distribution, then the ith outcome will pop up roughly mp_i times (if m is large). For simplicity, assume these are exactly the observed frequencies, and moreover that the p_i's are all powers of 2 (that is, of the form $1/2^k$). It can be seen by induction (Exercise 5.19) that the number of bits needed to encode the sequence is $\sum_{i=1}^{n} mp_i \log(1/p_i)$. Thus the average number of bits needed to encode a single draw from the distribution is

$$\sum_{i=1}^{n} p_i \log \frac{1}{p_i}.$$

This is the *entropy* of the distribution, a measure of how much randomness it contains.

For example, a fair coin has two outcomes, each with probability $1/2$. So its entropy is

$$\tfrac{1}{2}\log 2 + \tfrac{1}{2}\log 2 \quad = \quad 1.$$

This is natural enough: the coin flip contains one bit of randomness. But what if the coin is not fair, if it has a $3/4$ chance of turning up heads? Then the entropy is

$$\tfrac{3}{4}\log \tfrac{4}{3} + \tfrac{1}{4}\log 4 \quad = \quad 0.81.$$

A biased coin is more predictable than a fair coin, and thus has lower entropy. As the bias becomes more pronounced, the entropy drops toward zero.

We explore these notions further in Exercise 5.18 and 5.19.

5.3 Horn formulas

In order to display human-level intelligence, a computer must be able to perform at least some modicum of logical reasoning. Horn formulas are a particular framework for doing this, for expressing logical facts and deriving conclusions.

The most primitive object in a Horn formula is a *Boolean variable*, taking value either `true` or `false`. For instance, variables x, y, and z might denote the following possibilities.

$$\begin{aligned} x &\equiv \text{the murder took place in the kitchen} \\ y &\equiv \text{the butler is innocent} \\ z &\equiv \text{the colonel was asleep at 8 pm} \end{aligned}$$

A *literal* is either a variable x or its negation $\overline{x}$ ("NOT x"). In Horn formulas, knowledge about variables is represented by two kinds of *clauses*:

1. *Implications*, whose left-hand side is an AND of any number of positive literals and whose right-hand side is a single positive literal. These express statements of the form "if the conditions on the left hold, then the one on the right must also be true." For instance,

   $$(z \wedge w) \Rightarrow u$$

   might mean "if the colonel was asleep at 8 pm and the murder took place at 8 pm then the colonel is innocent." A degenerate type of implication is the *singleton* "$\Rightarrow x$," meaning simply that x is `true`: "the murder definitely occurred in the kitchen."

2. Pure *negative clauses*, consisting of an OR of any number of negative literals, as in

   $$(\overline{u} \vee \overline{v} \vee \overline{y})$$

   ("they can't all be innocent").

Given a set of clauses of these two types, the goal is to determine whether there is a consistent explanation: an assignment of `true`/`false` values to the variables that satisfies all the clauses. This is also called a *satisfying assignment*.

The two kinds of clauses pull us in different directions. The implications tell us to set some of the variables to `true`, while the negative clauses encourage us to make them `false`. Our strategy for solving a Horn formula is this: We start with all variables `false`. We then proceed to set some of them to `true`, one by one, but very reluctantly, and only if we absolutely have to because an implication would otherwise be violated. Once we are done with this phase and all implications are satisfied, only then do we turn to the negative clauses and make sure they are all satisfied.

In other words, our algorithm for Horn clauses is the following greedy scheme (*stingy* is perhaps more descriptive):

```
Input:  a Horn formula
Output:  a satisfying assignment, if one exists
```

```
set all variables to false

while there is an implication that is not satisfied:
  set the right-hand variable of the implication to true

if all pure negative clauses are satisfied:  return the assignment
else:  return ''formula is not satisfiable''
```

For instance, suppose the formula is

$$(w \wedge y \wedge z) \Rightarrow x, \;\; (x \wedge z) \Rightarrow w, \;\; x \Rightarrow y, \;\; \Rightarrow x, \;\; (x \wedge y) \Rightarrow w, \;\; (\overline{w} \vee \overline{x} \vee \overline{y}), \;\; (\overline{z}).$$

We start with everything `false` and then notice that x must be `true` on account of the singleton implication $\Rightarrow x$. Then we see that y must also be `true`, because of $x \Rightarrow y$. And so on.

To see why the algorithm is correct, notice that if it returns an assignment, this assignment satisfies both the implications and the negative clauses, and so it is indeed a satisfying truth assignment of the input Horn formula. So we only have to convince ourselves that if the algorithm finds no satisfying assignment, then there really is none. This is so because our "stingy" rule maintains the following invariant:

> If a certain set of variables is set to `true`, then they must be `true` in *any* satisfying assignment.

Hence, if the truth assignment found after the *while* loop does not satisfy the negative clauses, there can be no satisfying truth assignment.

Horn formulas lie at the heart of Prolog ("programming by logic"), a language in which you program by specifying desired properties of the output, using simple logical expressions. The workhorse of Prolog interpreters is our greedy satisfiability algorithm. Conveniently, it can be implemented in time linear in the length of the formula; do you see how (Exercise 5.32)?

5.4 Set cover

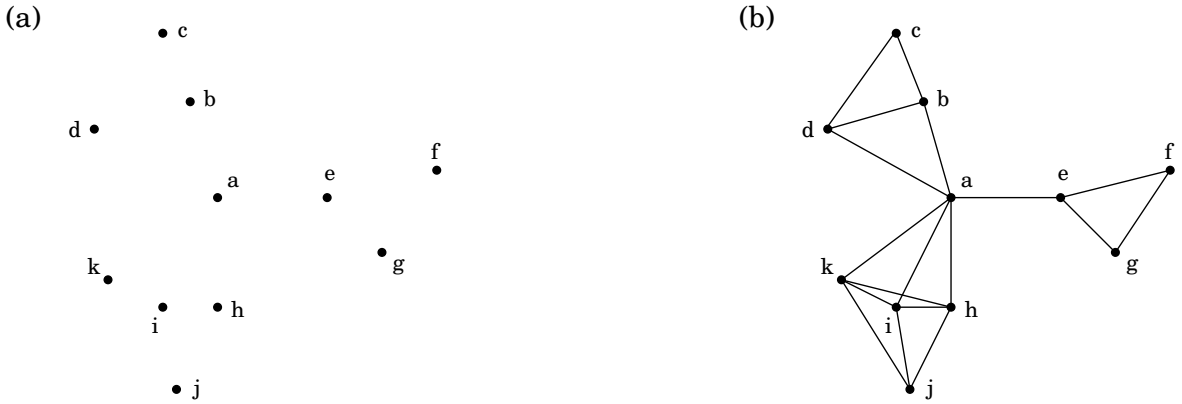
The dots in Figure 5.11 represent a collection of towns. This county is in its early stages of planning and is deciding where to put schools. There are only two constraints: each school should be in a town, and no one should have to travel more than 30 miles to reach one of them. What is the minimum number of schools needed?

This is a typical *set cover* problem. For each town x, let S_x be the set of towns within 30 miles of it. A school at x will essentially "cover" these other towns. The question is then, how many sets S_x must be picked in order to cover all the towns in the county?

SET COVER

Input: A set of elements B; sets $S_1, \ldots, S_m \subseteq B$

Figure 5.11 (a) Eleven towns. (b) Towns that are within 30 miles of each other.



Output: A selection of the S_i whose union is B.

Cost: Number of sets picked.

(In our example, the elements of B are the towns.) This problem lends itself immediately to a greedy solution:

Repeat until all elements of B are covered:

Pick the set S_i with the largest number of uncovered elements.

This is extremely natural and intuitive. Let's see what it would do on our earlier example: It would first place a school at town a, since this covers the largest number of other towns. Thereafter, it would choose three more schools—c, j, and either f or g—for a total of four. However, there exists a solution with just three schools, at b, e, and i. The greedy scheme is not optimal!

But luckily, it isn't too far from optimal.

Claim *Suppose B contains n elements and that the optimal cover consists of k sets. Then the greedy algorithm will use at most $k \ln n$ sets.*[2]

Let n_t be the number of elements still not covered after t iterations of the greedy algorithm (so $n_0 = n$). Since these remaining elements are covered by the optimal k sets, there must be some set with at least n_t/k of them. Therefore, the greedy strategy will ensure that

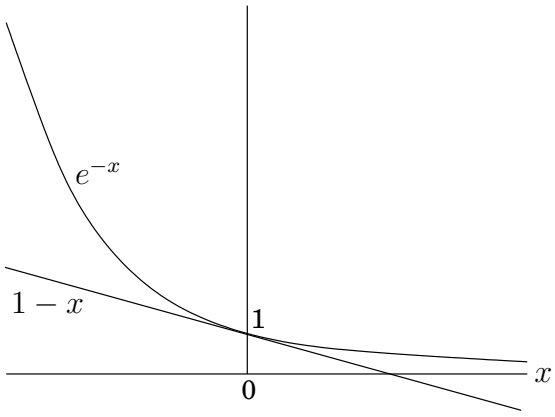
$$n_{t+1} \;\leq\; n_t - \frac{n_t}{k} \;=\; n_t\left(1 - \frac{1}{k}\right),$$

which by repeated application implies $n_t \leq n_0(1 - 1/k)^t$. A more convenient bound can be obtained from the useful inequality

$$1 - x \leq e^{-x} \text{ for all } x, \text{ with equality if and only if } x = 0,$$

[2] $\ln$ means "natural logarithm," that is, to the base e.

which is most easily proved by a picture:



Thus

$$n_t \;\leq\; n_0\left(1-\frac{1}{k}\right)^t \;<\; n_0(e^{-1/k})^t \;=\; ne^{-t/k}.$$

At $t = k\ln n$, therefore, n_t is strictly less than $ne^{-\ln n} = 1$, which means no elements remain to be covered.

The ratio between the greedy algorithm's solution and the optimal solution varies from input to input but is always less than $\ln n$. And there are certain inputs for which the ratio is very close to $\ln n$ (Exercise 5.33). We call this maximum ratio the *approximation factor* of the greedy algorithm. There seems to be a lot of room for improvement, but in fact such hopes are unjustified: it turns out that under certain widely-held complexity assumptions (which will be clearer when we reach Chapter 8), there is provably no polynomial-time algorithm with a smaller approximation factor.