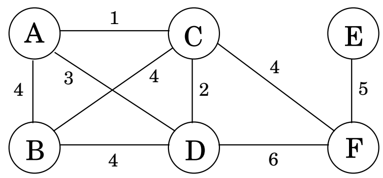
Greedy algorithms[1]

1 Minimum spanning trees

Suppose you are asked to network a collection of computers by linking selected pairs of them. This translates into a graph problem in which nodes are computers, undirected edges are potential links, and the goal is to pick enough of these edges so that the nodes are connected. But this is not all, because each link also has a maintenance cost, reflected in that edge's weight. What is the cheapest possible network?



One immediate observation is that the optimal set of edges cannot contain a cycle, because removing an edge from this cycle would reduce the cost without compromising connectivity:

1. Removing a cycle edge cannot disconnect a graph.

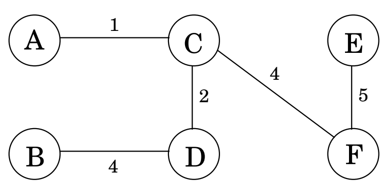
So the solution must be connected and acyclic: undirected graphs of this kind are called *trees*. The particular tree we want is the one with minimum total weight, known as the *minimum spanning tree*. Here is its formal definition.

Input: An undirected graph $G = (V, E)$; edge weights w_e.

Output: A tree $T = (V, E')$, which minimizes

$$\text{weight}(T) \;=\; \sum_{e \in E'} w_e.$$

In the example above, the minimum spanning tree has a cost of 16:



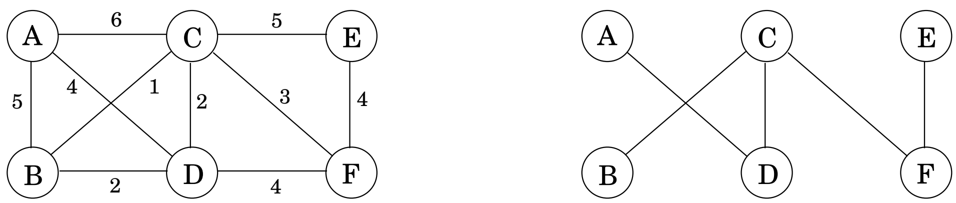
However, this is not the only optimal solution. Can you spot another?

1.1 A greedy approach

Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from E according to the following rule.

Repeatedly add the next lightest edge which doesn't produce a cycle.

Figure 1.1 Kruskal's algorithm.



In other words, it constructs the tree edge by edge, and apart from taking care to avoid cycles, simply picks whichever edge is cheapest at the moment. This is a *greedy* algorithm: every decision it makes is the one with the most obvious immediate advantage.

Figure 1.1 shows an example. We start with an empty graph and then attempt to add edges in increasing order of weight:

$$B-C,\ C-D,\ B-D,\ C-F,\ D-F,\ E-F,\ A-D,\ A-B,\ C-E,\ A-C.$$

(Some ties have been broken arbitrarily.) The first two succeed but the third, $B-D$, would produce a cycle if added. So we ignore it and move along. The final result is a tree with cost 14, the minimum possible.

At the outset of Kruskal's algorithm, each node is in a connected component all by itself. Then edges start being added, and the connected components get bigger and fewer in number. An edge $e = \{u, v\}$ is incorporated only if it doesn't produce a cycle, that is, only if u, v lie in different components. The precise effect of adding the edge is to merge these two connected components, and thereby to reduce the overall number of components by one. Suppose there are n nodes. Over the course of this incremental process, the number of connected components decreases from n to one, which means that $n-1$ edges must have been added along the way.

This consideration does not apply solely to the trees produced by Kruskal's algorithm. For *any* tree, imagine starting with an empty graph (with nodes but no edges), and then inserting the tree's edges one by one, in any arbitrary order. Our earlier argument still holds.

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

What happens when an edge e is removed from a tree? Well, we can pretend that e was the last edge to be added when the tree was being built (recall that the ordering was irrelevant). By rewinding the tree construction process one step, we find we must have two connected components. Neither of them can contain cycles, so they must be trees.

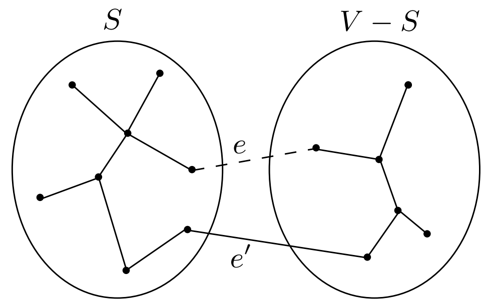
3. Removing an edge from a tree breaks it into two smaller trees.

Many more properties of trees can be derived by reasoning in this way. We now use similar logic to establish a simple rule which justifies the correctness of a whole slew of greedy minimum spanning tree algorithms, including Kruskal's.

1.2 The cut property

Suppose that in the process of building a minimum spanning tree (henceforth abbreviated MST), we have already chosen some edges $X \subseteq E$ and are so far on the right track: there is

Figure 1.2 $T \cup \{e\}$. 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 $(S, V - S)$.



some MST containing these edges. Which edge should we add next? The following property gives us a lot of flexibility in our choice.

> **Cut property.** Let $X \subseteq E$ be part of some minimum spanning tree of graph $G = (V, E)$. Pick any set of nodes $S \subset V$ such that there is no edge in X which connects a node in S to one in $V - S$. If $e \in E$ is the minimum-weight edge between S and $V - S$, then $X \cup \{e\}$ is also part of some MST.

Let's see why this holds. The starting set X is part of some MST T; if the new edge e happens to be part of T, then we are in luck and there is nothing to prove. So assume e is not in T. We have to exhibit a different MST T' which contains $X \cup \{e\}$. We will do so by altering T slightly, changing just one of its edges.

What happens when edge e is added to T? Since T is connected, it already has a path between the endpoints of e, so the addition of this edge produces a cycle. This cycle (call it C) contains e, and must therefore also have some other edge e' across the cut $(S, V - S)$ (Figure 1.2). Define $T' = T \cup \{e\} - \{e'\}$, and let's check that it is a tree. We know from property (3) that $T - \{e'\}$ consists of two disjoint smaller trees, each containing an endpoint of e'. The addition of edge e unites these two trees: it creates a path (specifically, $C - \{e'\}$) between the endpoints of e'.

Is T' 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 weight$(T') \leq$ weight(T). Since T is a MST, so it T'.

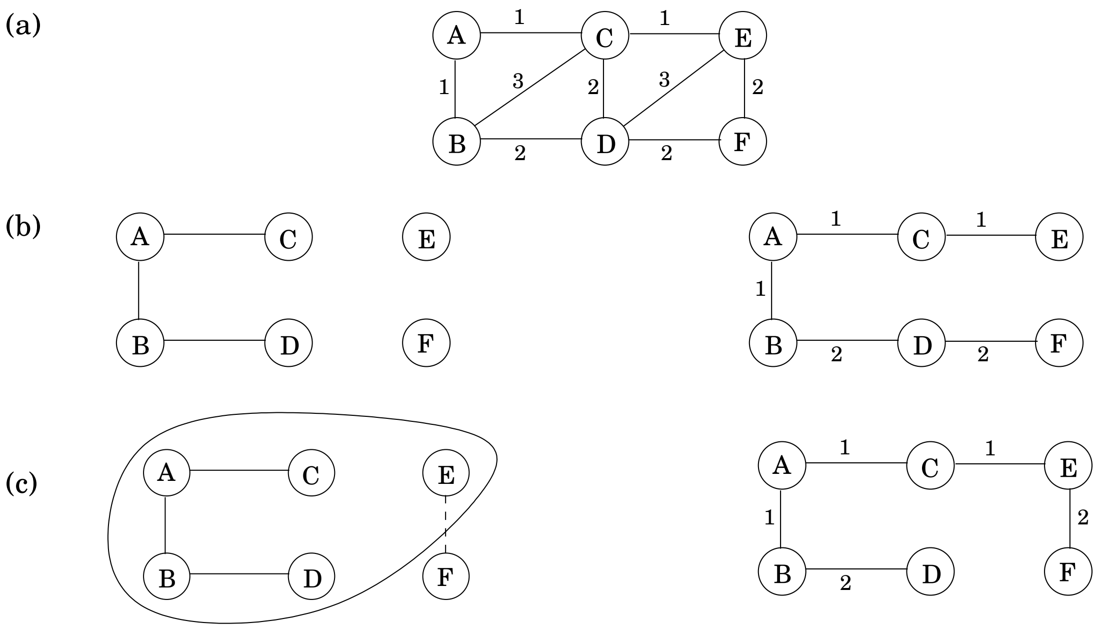
Figure 1.3 shows an example of the cut property. Which edge is e'?

1.3 Kruskal's algorithm

Kruskal's algorithm is justified by the cut property. Whenever it adds an edge e that unites two connected components C_1, C_2, we are sure that e is the lightest edge between C_1 and $V - C_1$. Therefore the final tree returned is guaranteed to be optimal.

Now we fill in some implementation details. Throughout the algorithm, we have to test candidate edges $u - v$ to see whether their endpoints lie in different connected components.

Figure 1.3 The cut property. (a) An undirected graph. (b) Set X has three edges, and is part of the MST T on the right. (c) If $S = \{A, B, C, D, E\}$, then one of the minimum-weight edges across the cut $(S, V - S)$ is $e = \{E, F\}$. $X \cup \{e\}$ is part of MST T', shown on the right.



To efficiently perform this check, we will maintain a collection of (disjoint) sets, each of which contains the nodes of a particular connected component.

We need three set operations. 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?

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

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

The final algorithm is shown in Figure 1.4. It needs $O(|E| \log |E|)$ time to sort the edges, plus time for $|V|$ `makeset`, $2|E|$ `find` and $|V| - 1$ `union` operations. We will next see how to perform all the disjoint-set operations in time $O(|E| \log^* |V|)$, where $\log^* |V|$ (to be defined shortly) is much smaller than $\log |V|$, and in fact, is for all practical purposes a constant. Therefore the total running time is $O(|E| \log |V|)$.

1.4 A data structure for disjoint sets

1.4.1 Union by rank

One way to store a set is as a directed tree (Figure 1.5). Nodes of the tree are elements of the set, arranged in no particular order, and each with parent pointers which eventually lead up to the root of the tree. This root element is a convenient *representative*, or *name*, for the

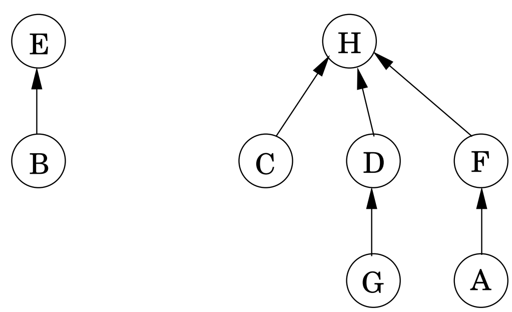
Figure 1.4 Kruskal's minimum spanning tree algorithm.

```
procedure kruskal(G, w)
Input:    An 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)
```

Figure 1.5 A directed-tree representation of two sets $\{B, E\}$ and $\{A, C, D, F, G, H\}$.



whole set. It is distinguished from the other elements by the fact that its parent pointer is a self-loop.

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

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

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

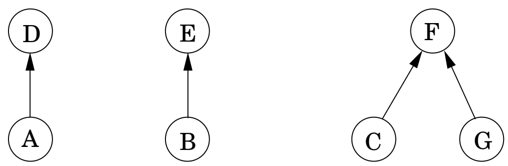
`Makeset` is, of course, a constant time operation. On the other hand, `find` follows parent pointers to the root of the tree and therefore takes time proportional to the height of the tree. The tree actually gets built via the third operation, `union`, and so we must make sure that this procedure keeps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

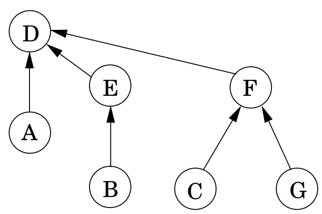
Figure 1.6 A sequence of operations on the disjoint set data structure.

makeset(A), makeset$(B), \ldots,$ makeset(G)

union(A, D), union(B, E), union(C, F), union(C, G)



union(E, A), union(B, G)



or the other way round? 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 the 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)
$r_x = \texttt{find}(x)$
$r_y = \texttt{find}(y)$
if $r_x = r_y$: return
if $\texttt{rank}(r_y) > \texttt{rank}(r_x)$:
  $\pi(r_x) = r_y$
else:
  $\pi(r_y) = r_x$
  if $\texttt{rank}(r_x) = \texttt{rank}(r_y)$: $\texttt{rank}(r_x) = \texttt{rank}(r_x) + 1$

See Figure 1.6 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 towards a root node, the *rank* values along the way are strictly increasing.

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

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

This extends to internal (non-root) nodes as well: a node of rank k has at least 2^k descendants. After all, any internal node was once a root, and its set of descendants has not changed since then. Moreover, different rank-k nodes cannot share common descendants, since by property (1) any element has at most one ancestor of rank k. Which means

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 all operations.

1.4.2 Path compression

Can the trees in the data structure be made even shorter? One idea is to be opportunistic: during a `find`(x) operation, when a series of parent pointers is followed up to the root of a tree, change all of these pointers so that they point directly to the root (Figure). After all, we spent all this time discovering the tree of all these nodes, not just of x, and it makes sense to somwhow remember this information. This is called *path compression*. It only doubles the amount of time needed for a `find`, and is easy to code — here is a slick recursive implementation:

procedure find(x)
`if` $x \neq \pi(x):\quad \pi(x) = $ `find`$(\pi(x))$
`return` x

This simple alteration makes a tremendous difference when there are multiple `find`'s along overlapping paths. To capture this effect, we will analyze *sequences* of `find` (or `union`) operations, and show that the *average* time per operation is only slightly more than $O(1)$.

Boxed: Cheaper by the dozen: Amortized analysis

What does *not* change under path compression? To figure this out, let's divide nodes into two categories: those which are roots of trees, and those which aren't. In the absence of path compression, these two groups have the following properties.

- *Non-root nodes:* their rank never changes, and they can never again resurface as roots.
- *Root nodes:* their ranks can change, and the composition of this layer of nodes can change (as new elements as added, and others cease to be roots), but all such changes depend only on the other root nodes.

Path compression only changes parent pointers of non-root nodes. Therefore,

4. The ranks of nodes, and the identities of the root elements, are exactly the same as if there were no path compression —but of course, due to path compression, the rank of a node is now just a quantity useful in the analysis, not the longest path from a leaf to it.

This means that properties (1)–(3) continue to hold.

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

$$[0], [1], [2, 3], [4, 5, \ldots, 15], [16, 17, \ldots, 2^{16} - 1 = 65535], \ldots$$

Each group is of the form $(k, k + 1, \ldots, 2^k - 1]$. The number of groups is a number we denote by $\log^* n$, defined to be the number of successive $\log$ operations you need to apply to n to bring it down to one (or below). For instance, $\log^* 1000 = 4$ since $\log\log\log\log 1000 \leq 1$. In practice there will just be the five intervals shown above; more are needed only if we need to handle more than 2^{65536} elements — that is to say, never.

A `find` operation spends $O(1)$ time on each non-root node x that it encounters on its path. The parent pointer of x is changed, as a result of which the rank of its parent increases. Accordingly, divide the lifetime of x, from the moment it ceases to be a root, into two phases.

Phase one: rank($\pi(x)$) lies in the same interval as rank(x).

Phase two: rank($\pi(x)$) is in a higher interval than rank(x).

(If x has rank zero, phase one is skipped altogether.) To calculate the time taken by a sequence of m `find` operations, let's separately consider the time taken on phase-one nodes and phase-two nodes.

If x is in the group $(k, 2^k]$, then it can be visited at most 2^k times by `find` operations before moving on to phase two. Moreover, by property (3) the number of nodes with rank $\in (k, 2^k]$ is bounded by

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

So, phase-one nodes in this particular group receive a combined total of at most n visits from `find` operations. Since there are $\log^* n$ distinct groups, the overall time spent on phase-one nodes is $\leq n \log^* n$.

As for phase-two nodes, in any given `find`, the path being updated can have at most $\log^* n$ of them (why?). Therefore the time taken on these nodes, over m operations, is $O(m \log^* n)$. The grand total for m `find` (or `union`) operations is then $O((m + n) \log^* n)$, almost linear.

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}
```

Prim's algorithm is a somewhat more organized realization of this template than Kruskal's: the intermediate set of edges X is always connected and therefore forms a tree. S is chosen to be the set of this tree's vertices.

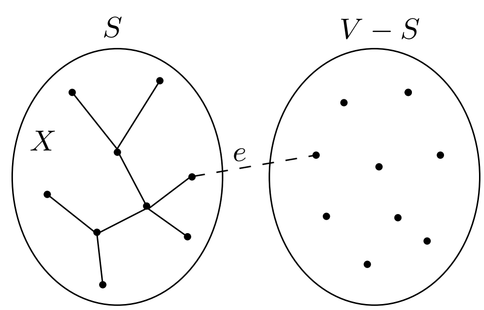
Figure 1.8 Prim's minimum spanning tree algorithm.

```
procedure prim(G, w)
Input:    An 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 = makeheap(V)    (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)
```

Figure 1.7 Prim's algorithm: the edges X form a tree, and S consists of its vertices.



On each iteration, the tree defined by X *grows* by one edge, namely the lightest edge between a vertex in S (a tree-vertex) and a vertex outside S (Figure 1.7), whose weight is

$$\min_{v \notin S} \min_{u \in S} w(u, v).$$

We can equivalently think of S as growing to include the vertex $v \notin S$ of smallest `cost`:

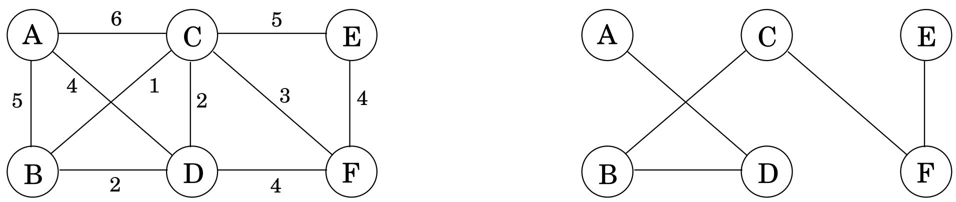
$$\text{cost}(v) \quad = \quad \min_{u \in S} w(u, v).$$

The assimilation of v in S is accomplished via an edge whose other endpoint is

$$\text{prev}(v) \quad = \quad \text{the node closest to } v \text{ in } S.$$

When this occurs, the `cost` values do not need to be recomputed from scratch, but merely updated to include edges out of v.

Figure 1.9 An illustration of Prim's algorithm, starting at node A. Also shown are a table of `cost/prev` values, and the final MST.



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					$4/F$	

This sounds a lot like Dijkstra's algorithm, and in fact the pseudocode is almost identical (Figure 1.8). The only difference is in the key values by which the heap is prioritized. In Dijkstra's algorithm, the value of a node is the length of an entire path to that node from the starting point, since in the shortest paths problem we are interested in the length of the whole path and not just of the last edge. Nonetheless, the two algorithms are similar enough that they have the same running time, which depends on the particular heap implementation.

Figure 1.9 shows Prim's algorithm at work, on a small six-node graph. Notice how the final MST is completely specified by the `prev` array.

2 Huffman encoding

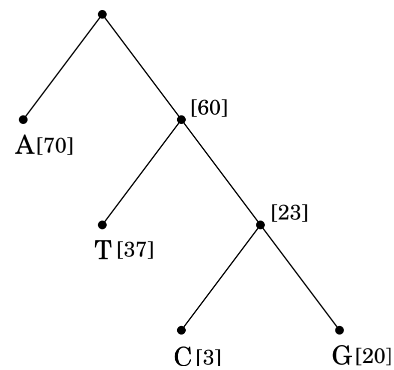
Suppose you are asked to store a string of 130 million characters, all of them A, C, G, or T – the map of a chromosome, say. The default option is to use one byte per character, but this seems wasteful when just two bits will suffice: 00 for A, 01 for C, 10 for G, and 11 for T. This way, you need only 260 megabits in total.

Taking a closer look at the string, you find that the four characters are not equally abundant. The most frequent is A, which appears 70 million times, while C appears just 3 million times, G 20 million times, and T 37 million times. Perhaps there is an even more economical encoding, for instance one that assigns a shorter bit string to A than to C? The only thing to be careful of is that these variable-length encodings must obey the *prefix* property: none of (four) codes should be a prefix of another code. If A were 0 and C were 001, the decoding of strings like $0010\cdots$ might be ambiguous, or at the very least complicated.

Here's an encoding that works well: 0 for A, 110 for C, 111 for G and 10 for T. It can be represented by a binary tree (Figure 2.1) in which the characters are at the leaves and the bit string is generated by the path from root to leaf, interpreting "left" as 0 and "right" as 1. The total size of the encoding drops to 213 megabits, a 17% improvement.

The tree can also be thought of as specifying a decoding procedure. Given some long string

Figure 2.1 Code: $A = 0, C = 110, G = 111, T = 10$. Frequencies are shown in square brackets.



of bits which you want to translate into characters, start at the root of the tree and use the string to guide you down. Each time you reach a leaf, output that character and move back to the root.

The tree of Figure 2.1 happens to be optimal. In general, given a bunch of characters $\sigma_1, \ldots, \sigma_n$, and their frequencies $f_1, \ldots, f_n$, how can we find such a tree, or to start with an even more basic question, what properties does it possess? Well, each leaf corresponds to a character, and its depth is the length of that character's code. The optimal tree is the one which minimizes the overall size of the encoding,

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

Although we are only given frequencies for the leaves, we can also assign frequencies to any internal node: the number of times it is visited is precisely the sum of the frequencies of its descendant leaves. The cost function can then be rewritten even more simply:

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

This is because each edge of the tree corresponds to a bit, and the number of times that bit is written is exactly the frequency of the node to which the edge leads.

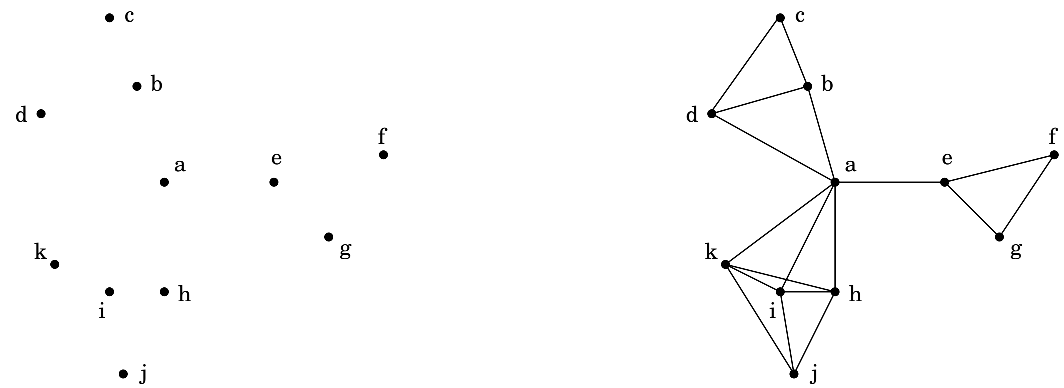
Both formulations of the cost function will turn out to be handy. The first one tells us an extremely useful property of the optimal tree.

> The two characters with the smallest frequencies must be together at the bottom of the tree, as children of the lowest internal node in the tree.

Otherwise, the encoding could be improved by swapping.

This suggests that we start constructing the tree *greedily*: find the smallest f_i, f_j and make them the children of an internal node. This new node, which we'll call σ_{ij}, has a frequency of $f_i + f_j$; for instance, see the parent of C, G in the figure. Together with σ_i and σ_j, it forms a little subtree (call it T_{mini}) which contains two edges and which, crucially, is guaranteed to be part of the optimal tree T. This is a good start, but what about the rest of T? Well, if we remove

Figure 3.1 *Left:* Eleven towns. *Right:* Towns which are within thirty miles of each other.



the two edges from it, we are left with a tree that has $n-1$ leaves, namely the remaining σ_k's and the new node σ_{ij}. Call this T_{rest}. Our second formulation of the cost function then tells us that the cost of T is simply the cost of T_{mini}, which is now fixed, plus that of T_{rest}. Therefore T_{rest} is itself the optimal tree for its particular set of leaves, and we can find it by recursing.

function buildtree($L = [(\sigma_1, f_1), \ldots, (\sigma_n, f_n)]$)
(Returns a tree with leaves $\sigma_1, \ldots, \sigma_n$)
let f_i, f_j be the two smallest frequencies
remove $(\sigma_i, f_i), (\sigma_j, f_j)$ from L
$T \leftarrow$ buildtree($L \circ [(\sigma_{ij}, f_i + f_j)]$)
set the children of σ_{ij} to σ_i and σ_j
return T

Returning to the chromosome example, we do not mean to suggest that there is no better way to compress the string. The claim is that it is impossible to do better by *encoding single characters*. Chances are there are far better encodings, for instance using 010 to denote the frequently occurring substring AAGATTA, or something of the sort. However, once the text is scanned, and blocks of characters to be encoded are identified, Huffman's algorithm gives the best encoding. Many applications, including image and video transmission, use it as their basic building block.

3 Set cover

The dots in Figure 3.1 are intended to 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 thirty 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 thirty miles of it. A school at x will essentially "cover" these other towns. The question is then, how many sets S_x need to be picked in order to cover all the towns in the county?

SET COVER

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

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. However, it does not necessarily give the optimal solution. In our earlier example, this greedy scheme would first place a school at town a, since this covers the largest number of other towns. However, this would eventually force it to place four schools, whereas three suffice – for instance, at b, e, i.

Although the greedy approach is not optimal, it is fairly close.

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

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

By repeatedly applying this, and using the rule $1 - x < e^{-x}$ (for $x > 0$), we get

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

At $t = k \ln n$, this number is less than one.

We say that the greedy algorithm has an *approximation ratio* of $\ln n$, the maximum ratio between its solution and the optimal solution. This guarantee is reassuring, but seems to leave a lot of room for improvement. Such hopes are unjustified: it turns out that under certain widely held assumptions (which we will clarify in a later chapter), there is provably no polynomial-time algorithm with a smaller approximation ratio.

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