the time, since after finding the name x of the set containing v, we have to go back through the same path of pointers from v to x, and reset each of these pointers to point to x directly. But this additional work can at most double the time required, and so does not change the fact that a `Find` takes at most $O(\log n)$ time. The real gain from compression is in making subsequent calls to `Find` cheaper, and this can be made precise by the same type of argument we used in (4.23): bounding the total time for a sequence of n `Find` operations, rather than the worst-case time for any one of them. Although we do not go into the details here, a sequence of n `Find` operations employing compression requires an amount of time that is extremely close to linear in n; the actual upper bound is $O(n\alpha(n))$, where $\alpha(n)$ is an extremely slow-growing function of n called the *inverse Ackermann function*. (In particular, $\alpha(n) \leq 4$ for any value of n that could be encountered in practice.)

Implementing Kruskal's Algorithm

Now we'll use the `Union-Find` data structure to implement Kruskal's Algorithm. First we need to sort the edges by cost. This takes time $O(m \log m)$. Since we have at most one edge between any pair of nodes, we have $m \leq n^2$ and hence this running time is also $O(m \log n)$.

After the sorting operation, we use the `Union-Find` data structure to maintain the connected components of (V, T) as edges are added. As each edge $e = (v, w)$ is considered, we compute `Find`(u) and `Find`(v) and test if they are equal to see if v and w belong to different components. We use `Union(Find`(u),`Find`$(v))$ to merge the two components, if the algorithm decides to include edge e in the tree T.

We are doing a total of at most $2m$ `Find` and $n - 1$ `Union` operations over the course of Kruskal's Algorithm. We can use either (4.23) for the array-based implementation of `Union-Find`, or (4.24) for the pointer-based implementation, to conclude that this is a total of $O(m \log n)$ time. (While more efficient implementations of the `Union-Find` data structure are possible, this would not help the running time of Kruskal's Algorithm, which has an unavoidable $O(m \log n)$ term due to the initial sorting of the edges by cost.)

To sum up, we have

(4.25) *Kruskal's Algorithm can be implemented on a graph with n nodes and m edges to run in $O(m \log n)$ time.*

4.7 Clustering

We motivated the construction of minimum spanning trees through the problem of finding a low-cost network connecting a set of sites. But minimum

spanning trees arise in a range of different settings, several of which appear on the surface to be quite different from one another. An appealing example is the role that minimum spanning trees play in the area of *clustering*.

The Problem

Clustering arises whenever one has a collection of objects—say, a set of photographs, documents, or microorganisms—that one is trying to classify or organize into coherent groups. Faced with such a situation, it is natural to look first for measures of how similar or dissimilar each pair of objects is. One common approach is to define a *distance function* on the objects, with the interpretation that objects at a larger distance from one another are less similar to each other. For points in the physical world, distance may actually be related to their physical distance; but in many applications, distance takes on a much more abstract meaning. For example, we could define the distance between two species to be the number of years since they diverged in the course of evolution; we could define the distance between two images in a video stream as the number of corresponding pixels at which their intensity values differ by at least some threshold.

Now, given a distance function on the objects, the clustering problem seeks to divide them into groups so that, intuitively, objects within the same group are "close," and objects in different groups are "far apart." Starting from this vague set of goals, the field of clustering branches into a vast number of technically different approaches, each seeking to formalize this general notion of what a good set of groups might look like.

Clusterings of Maximum Spacing Minimum spanning trees play a role in one of the most basic formalizations, which we describe here. Suppose we are given a set U of n objects, labeled $p_1, p_2, \ldots, p_n$. For each pair, p_i and p_j, we have a numerical distance $d(p_i, p_j)$. We require only that $d(p_i, p_i) = 0$; that $d(p_i, p_j) > 0$ for distinct p_i and p_j; and that distances are symmetric: $d(p_i, p_j) = d(p_j, p_i)$.

Suppose we are seeking to divide the objects in U into k groups, for a given parameter k. We say that a *k-clustering* of U is a partition of U into k nonempty sets $C_1, C_2, \ldots, C_k$. We define the *spacing* of a k-clustering to be the minimum distance between any pair of points lying in different clusters. Given that we want points in different clusters to be far apart from one another, a natural goal is to seek the k-clustering with the maximum possible spacing.

The question now becomes the following. There are exponentially many different k-clusterings of a set U; how can we efficiently find the one that has maximum spacing?

Designing the Algorithm

To find a clustering of maximum spacing, we consider growing a graph on the vertex set U. The connected components will be the clusters, and we will try to bring nearby points together into the same cluster as rapidly as possible. (This way, they don't end up as points in different clusters that are very close together.) Thus we start by drawing an edge between the closest pair of points. We then draw an edge between the next closest pair of points. We continue adding edges between pairs of points, in order of increasing distance $d(p_i, p_j)$. In this way, we are growing a graph H on U edge by edge, with connected components corresponding to clusters. Notice that we are only interested in the connected components of the graph H, not the full set of edges; so if we are about to add the edge (p_i, p_j) and find that p_i and p_j already belong to the same cluster, we will refrain from adding the edge—it's not necessary, because it won't change the set of components. In this way, our graph-growing process will never create a cycle; so H will actually be a union of trees. Each time we add an edge that spans two distinct components, it is as though we have merged the two corresponding clusters. In the clustering literature, the iterative merging of clusters in this way is often termed *single-link clustering*, a special case of *hierarchical agglomerative clustering*. (*Agglomerative* here means that we combine clusters; *single-link* means that we do so as soon as a single link joins them together.) See Figure 4.14 for an example of an instance with $k = 3$ clusters where this algorithm partitions the points into an intuitively natural grouping.

What is the connection to minimum spanning trees? It's very simple: although our graph-growing procedure was motivated by this cluster-merging idea, our procedure is precisely Kruskal's Minimum Spanning Tree Algorithm. We are doing exactly what Kruskal's Algorithm would do if given a graph G on U in which there was an edge of cost $d(p_i, p_j)$ between each pair of nodes (p_i, p_j). The only difference is that we seek a k-clustering, so we stop the procedure once we obtain k connected components.

In other words, we are running Kruskal's Algorithm but stopping it just before it adds its last $k - 1$ edges. This is equivalent to taking the full minimum spanning tree T (as Kruskal's Algorithm would have produced it), deleting the $k - 1$ most expensive edges (the ones that we never actually added), and defining the k-clustering to be the resulting connected components $C_1, C_2, \ldots, C_k$. Thus, iteratively merging clusters is equivalent to computing a minimum spanning tree and deleting the most expensive edges.

Analyzing the Algorithm

Have we achieved our goal of producing clusters that are as spaced apart as possible? The following claim shows that we have.

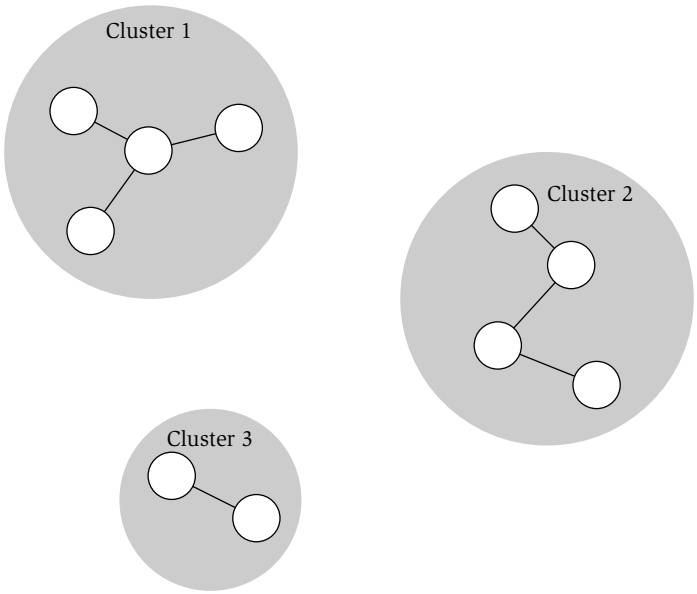

Figure 4.14 An example of single-linkage clustering with $k = 3$ clusters. The clusters are formed by adding edges between points in order of increasing distance.

(4.26) *The components $C_1, C_2, \ldots, C_k$ formed by deleting the $k-1$ most expensive edges of the minimum spanning tree T constitute a k-clustering of maximum spacing.*

Proof. Let $\mathcal{C}$ denote the clustering $C_1, C_2, \ldots, C_k$. The spacing of $\mathcal{C}$ is precisely the length d^* of the $(k-1)^{\text{st}}$ most expensive edge in the minimum spanning tree; this is the length of the edge that Kruskal's Algorithm would have added next, at the moment we stopped it.

Now consider some other k-clustering $\mathcal{C}'$, which partitions U into non-empty sets $C'_1, C'_2, \ldots, C'_k$. We must show that the spacing of $\mathcal{C}'$ is at most d^*.

Since the two clusterings $\mathcal{C}$ and $\mathcal{C}'$ are not the same, it must be that one of our clusters C_r is not a subset of any of the k sets C'_s in $\mathcal{C}'$. Hence there are points $p_i, p_j \in C_r$ that belong to different clusters in $\mathcal{C}'$—say, $p_i \in C'_s$ and $p_j \in C'_t \neq C'_s$.

Now consider the picture in Figure 4.15. Since p_i and p_j belong to the same component C_r, it must be that Kruskal's Algorithm added all the edges of a p_i-p_j path P before we stopped it. In particular, this means that each edge on

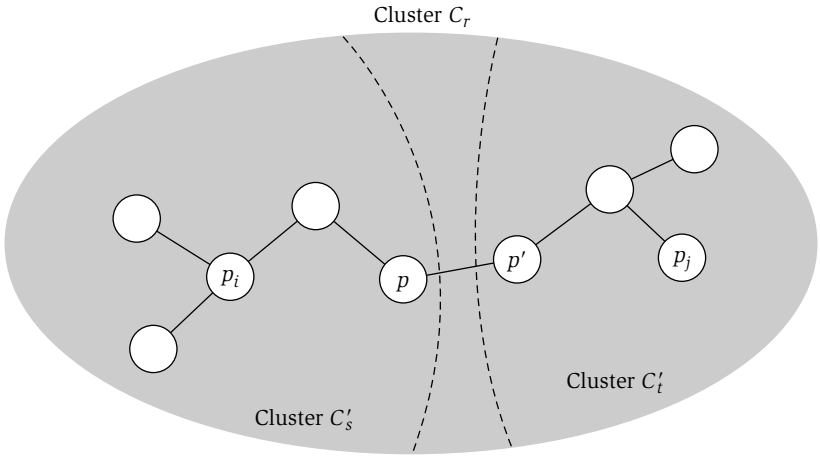

Figure 4.15 An illustration of the proof of (4.26), showing that the spacing of any other clustering can be no larger than that of the clustering found by the single-linkage algorithm.

P has length at most d^*. Now, we know that $p_i \in C'_s$ but $p_j \notin C'_s$; so let p' be the first node on P that does not belong to C'_s, and let p be the node on P that comes just before p'. We have just argued that $d(p, p') \leq d^*$, since the edge (p, p') was added by Kruskal's Algorithm. But p and p' belong to different sets in the clustering $\mathcal{C}'$, and hence the spacing of $\mathcal{C}'$ is at most $d(p, p') \leq d^*$. This completes the proof. ■

4.8 Huffman Codes and Data Compression

In the Shortest-Path and Minimum Spanning Tree Problems, we've seen how greedy algorithms can be used to commit to certain parts of a solution (edges in a graph, in these cases), based entirely on relatively short-sighted considerations. We now consider a problem in which this style of "committing" is carried out in an even looser sense: a greedy rule is used, essentially, to shrink the size of the problem instance, so that an equivalent smaller problem can then be solved by recursion. The greedy operation here is proved to be "safe," in the sense that solving the smaller instance still leads to an optimal solution for the original instance, but the global consequences of the initial greedy decision do not become fully apparent until the full recursion is complete.

The problem itself is one of the basic questions in the area of *data compression*, an area that forms part of the foundations for digital communication.