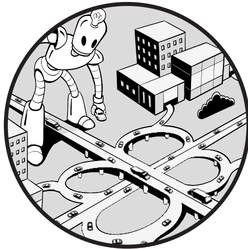
10

MINIMUM SPANNING TREES



The *minimum spanning tree* of a weighted, undirected graph is the set of edges with the smallest total weight that connects all the nodes. We can use this concept to model and optimize a variety of real-world problems, from designing power grids to hypothesizing how chipmunks should be constructing their burrows.

This chapter introduces two classical algorithms for constructing minimum spanning trees. Prim's algorithm is a nodewise agglomerative algorithm that builds a bigger and bigger set of connected nodes. Kruskal's algorithm constructs a minimum spanning tree from a sorted list of edges by adding one edge at a time.

After discussing how minimum spanning trees can be applied to several real-world problems, we consider two additional algorithms closely related to minimum spanning trees: grid-based maze generation and single-linkage clustering. We show how these tasks can be mapped into graph problems and solved using variations of the algorithms from this chapter.

The Structure of Minimum Spanning Trees

A *spanning tree* of a graph is a set of edges that connects all the nodes in the graph without forming any cycles. We can visualize spanning trees as the backbone of a real-world infrastructure network—the minimum connections needed to make every node reachable from any other node. These might be power lines, roads, links in a computer network, or the tunnels between holes in a chipmunk burrow. The *minimum spanning tree* is the set of edges that connect all the nodes while minimizing the sum of the edge weights.

We can picture these requirements in terms of an especially well-organized chipmunk's burrow, as shown in Figure 10-1. The chipmunk constructs their domain as a series of holes (nodes) linked by tunnels (edges). As in a graph, each tunnel directly links exactly two holes in a straight line. The chipmunk imposes two additional requirements. First, each hole to the surface needs to be reachable through its tunnels from any other hole. After all, what good are multiple entrances if they don't let you vanish into one and pop out of another? Second, the total distances of tunnels must be minimized. The chipmunk is lazy and would prefer to expend its energy randomly popping out of the ground at various points rather than digging new tunnels.

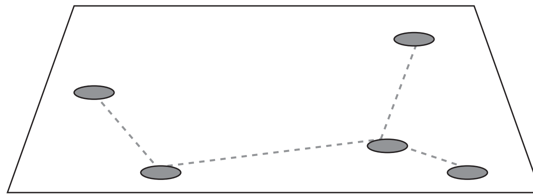


Figure 10-1: Five chipmunk holes connected as a minimum spanning tree

Formally we define the problem of finding the minimum spanning tree in a weighted, undirected graph as follows:

> Given a graph with a set of nodes V and edges E, find the set of edges $E' \subseteq E$ that connects every node in V while minimizing the sum of edge weights $\sum_{e \in E'} weight(e)$.

By definition, the minimum spanning tree will have $|V| - 1$ edges, the minimum number needed to connect $|V|$ nodes. Any more edges would add cycles and unnecessary weight.

Use Cases

This section introduces a few real-world examples of using the minimum spanning tree concept to design cost-efficient physical networks or optimize communications in a social network.

Physical Networks

Minimum spanning trees are useful in determining the minimum cost set of links that we need to build to fully connect a physical network. Imagine that the Algorithmic Coffee Shop Company is looking to build a state-of-the-art pneumatic tube system for delivering beans between its locations. After promising to serve over 10,000 varieties of coffee, the company quickly realizes that it lacks the storage space in some locations to keep such a vast variety on hand. Instead, it decides to build a central warehouse and ship small packets of beans to each store as needed. Every store will now boast an unparalleled selection.

The planners quickly realize that it is prohibitively expensive to build pneumatic tubes from every store to the warehouse. The two stores in Javaville are each located over 10 miles from the distribution center, but only two blocks from each other. It is much cheaper to build a single tube from the distribution center to the Main Street location and then a second tube from Main Street to the Coffee Boulevard location. A request for the Coffee Boulevard location can be satisfied by first sending the beans to the Main Street location and then forwarding them to Coffee Boulevard.

This multistep routing turns the design of the pneumatic delivery system into a minimum spanning tree problem, as shown in Figure 10-2. Each of the Algorithmic Coffee Shop Company's buildings is a node and each potential tube between any pair is an edge.

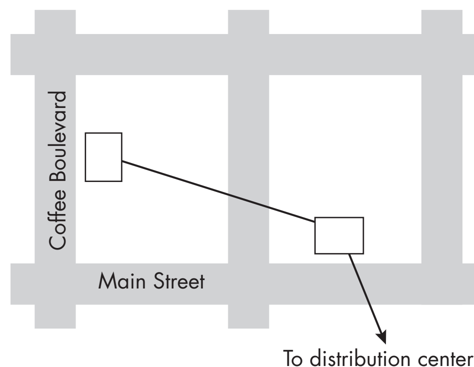


Figure 10-2: Two coffee shops on a minimum spanning tree delivery network

In Figure 10-2, the weight of an edge is the cost it would take to build the pneumatic tube between the two buildings. While often a factor of distance, the cost can also increase due to environmental factors. For example, building a tube that cuts through the center of a city is much more expensive than the same length tube under a farm. The planners need to find the set of edges (tubes to construct) that connects all the buildings while minimizing the cost.

Aside from pneumatic coffee tubes, more typical applications of minimum-cost spanning trees to physical networks include the following:

Constructing highways Nodes are cities, edges are highways, and edge weight is the cost to construct a highway between two points.

Power grids Nodes are cities, edges are transmission lines, and edge weight is the cost to construct the transmission lines between two points.

Bridging an archipelago Nodes are islands in the archipelago, edges are physical bridges between two islands, and edge weight is the cost to construct a bridge between two islands.

Design of airline networks Nodes are airports, edges are flights, and edge weight is the cost of flying between two airports.

Social Networks

Minimum spanning trees also apply to non-physical networks. For example, imagine a Society for Personal Communication Between Data Structure Experts that does not believe in bulk emails. Such announcement methods are much too impersonal. Instead, the organizers insist that each message be passed by a personal call from member to member. However, like in any organization consisting of experts, there exists a range of old friendships and feuds. Last year, Alice Hash Table had a falling out with Bob Binary Search Tree, and they no longer talk.

Every year, the organization develops an elaborate phone tree allowing the organization to spread the news of its upcoming conference while minimizing the discomfort of its members. Each member is represented as a node with edges to each other member. The cost of an edge is the level of discomfort two members have with talking to one another. In the best case, a chat among friends, the weight is minimal to represent the time cost of the phone call. However, in the worst case, the cost between two feuding members results in days of lost productivity and angry muttering. The organization needs to find the set of pairwise communications that informs every member about the conference details while minimizing overall angst. This requires all nodes to be connected using the minimum number and cost of edges.

Prim's Algorithm

Constructing a minimum spanning tree requires an algorithm to select a minimum cost subset of the edges from the full graph such that the resulting graph is fully connected. One method of finding a graph's minimum spanning tree is *Prim's algorithm*, which was independently proposed by multiple people including computer scientist R.C. Prim and mathematician Vojtěch Jarník. The algorithm operates very similarly to Dijkstra's algorithm

in Chapter 7, working through an unvisited set and building up a minimum spanning tree one node at a time.

Prim's algorithm starts with an unvisited set of all nodes and arbitrarily chooses one to visit. This visited node forms the start of the minimum spanning tree. On each iteration, the algorithm finds the unvisited node with the minimum edge weight to *any* of the nodes that it has previously visited, asking, "Which node is closest to our set's periphery and thus can be added with the least cost?" The algorithm removes this new node from the unvisited set and adds the corresponding edge to the minimum-cost spanning tree. It keeps adding nodes and edges, one per iteration, until it has visited every node.

Prim's algorithm will visit each node at most once and consider each edge at most twice (once from each end). Additionally, for each node, we may see a cost proportional to the logarithm of $|V|$ to insert or update a node in the priority queue implemented as a standard heap. The total cost of the algorithm therefore scales as $(|V| + |E|) \times \log(|V|)$.

We can picture Prim's algorithm as a construction company hired to upgrade bridges between islands in an archipelago. The company plans to replace the rotting wooden bridges connecting the archipelago with fully modern versions. Because the old wooden bridges will not support the weight of the construction equipment, from the company's point of view, only islands joined by a new bridge are truly connected. Their contract specifies that, in the end, any pair of islands must be reachable with a new modern bridge.

The builders start at a single island and work outward, connecting more and more islands with new bridges. At each step, they choose to upgrade the shortest wooden bridge that joins an island in the current connected set to an island outside that set. By always starting new bridges from an island in the connected set, the builders can move their equipment to the new edge's origin using modern bridges. By always ending bridges on islands outside the connected set, the builders increase the coverage of the connected set at every stage.

The Code

At each step of Prim's algorithm, we track the unconnected nodes along with the best edge weight seen that would connect them. We maintain this data using a custom `PriorityQueue` implementation that provides an efficient mechanism for looking up values in the queue and modifying priorities. For the purposes of this code, you need to understand only the basics of inserting items into the priority queue, removing items from the priority queue, and modifying priorities. However, if you're curious, you can review the details in Appendix B.

The code itself loops over the nodes in the priority queue until it is empty. Every time it removes a new node from the priority queue (the unvisited set), it examines that node's unvisited neighbors and checks whether the current node provides better (that is, lower cost) edges to any of its

unconnected neighbors. If so, it updates the neighbor's information with the new edge and weight:

```
def prims(g: Graph) -> Union[list, None]:
    pq: PriorityQueue = PriorityQueue(min_heap=True)
    last: list = [-1] * g.num_nodes
    mst_edges: list = []

  ❶ pq.enqueue(0, 0.0)
    for i in range(1, g.num_nodes):
        pq.enqueue(i, float('inf'))

  ❷ while not pq.is_empty():
        index: int = pq.dequeue()
        current: Node = g.nodes[index]

      ❸ if last[index] != -1:
            mst_edges.append(current.get_edge(last[index]))
        elif index != 0:
            return None

      ❹ for edge in current.get_edge_list():
            neighbor: int = edge.to_node
            if pq.in_queue(neighbor):

                if edge.weight < pq.get_priority(neighbor):
                    pq.update_priority(neighbor, edge.weight)
                    last[neighbor] = index

    return mst_edges
```

The code starts by creating a trio of helper data structures, including a min-heap-based priority queue of unconnected nodes (`pq`), an array indicating the last node visited before a given node (`last`), and the final set of edges for the minimum spanning tree (`mst_edges`). The code requires importing the custom `PriorityQueue` class defined in Appendix B, as well as importing `Union` from Python's `typing` library.

All nodes are inserted into the priority queue at the start of the algorithm ❶. The starting node (0) is given priority 0.0 and the rest are given infinite priority. The code then proceeds like Dijkstra's algorithm, processing the unvisited nodes one at a time. A `while` loop iterates until the priority queue of unvisited nodes is empty ❷. During each iteration, the node with the minimum distance to any of the visited nodes is chosen and dequeued from the priority queue. As we will see, this effectively removes the node from the unvisited set.

The code next checks whether there exists an edge back to one of the nodes in the connected set ❸. There are two cases in which the node's `last` entry might be `-1`. The first is node 0, which does not have a predecessor by virtue of being explored first. The second case is in a disconnected component where `index` is not reachable from node 0. In this latter case, because

all the nodes cannot be connected, the graph does not have a minimum spanning tree and the function returns `None`.

After adding the new node to the visited set (by dequeuing it), a `for` loop iterates over each of the node's neighbors ❹, checking whether the neighbor is unvisited (still in the priority queue). If so, the code checks whether it has found a better edge to the node by comparing the previous best edge weight with that of the new edge. The code finishes by returning the set of edges making up the minimum spanning tree.

Note that if a graph is disconnected, each connected component has its own minimum spanning tree. An alternative approach to the code presented here is to return the list of edges that create the minimum spanning trees for each connected component. We can implement this by removing the `elif` check ❸ and its corresponding return. The code will then move on to the next component by selecting a node from the priority queue and continue selecting edges.

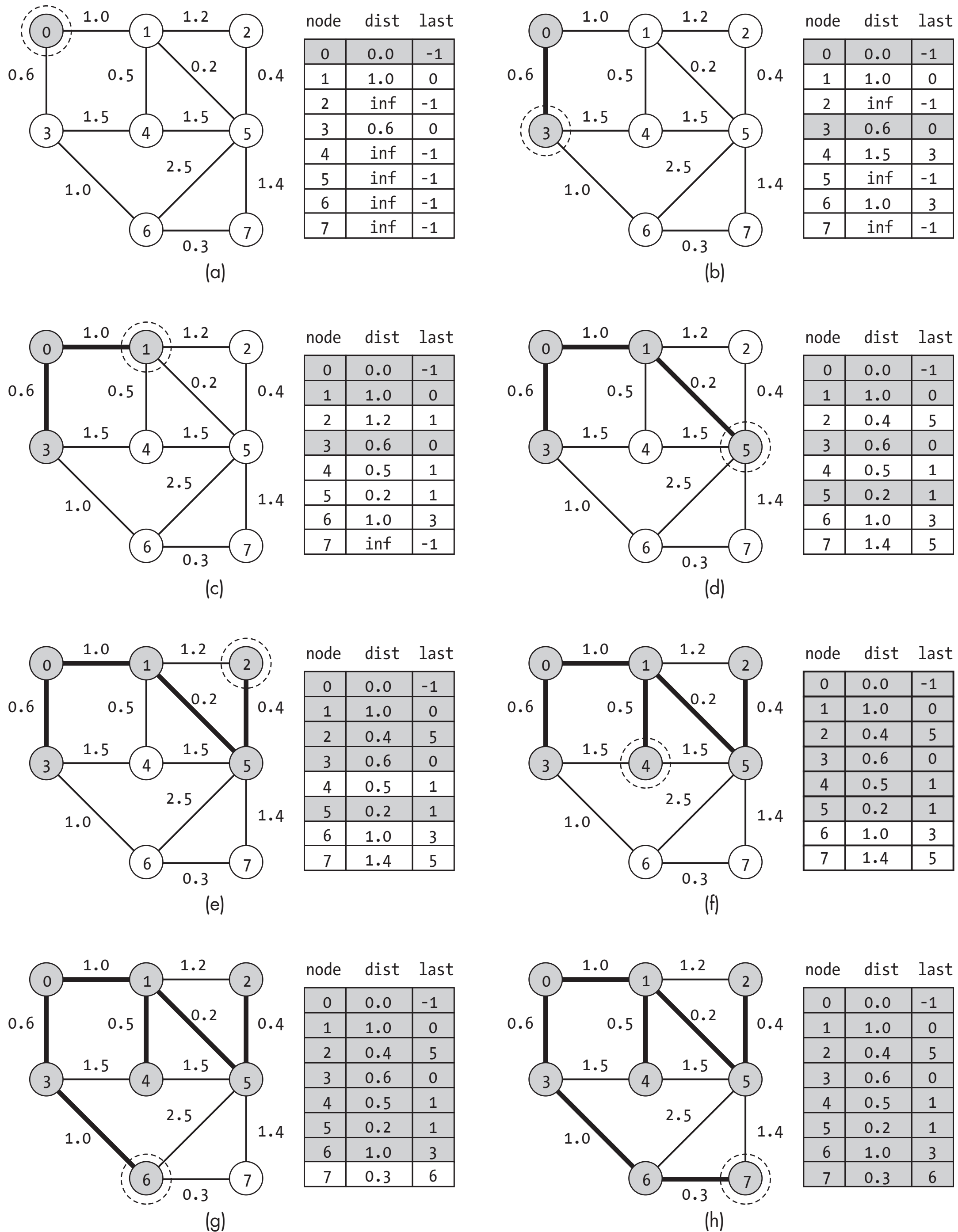
An Example

Figure 10-3 shows an illustration of Prim's algorithm on a graph with eight nodes. The table to the right of each subfigure shows the information tracked for each node, including the node's ID, the distance to that node from the connected set of nodes as stored by the node's priority, and the closest member of the current connected subset as stored in the `last` list. All nodes except the first one start with an infinite distance and a `last` node pointer of `-1` to indicate that we have yet to find a path that leads to that node. After removing a node from the priority queue, we gray out its row to indicate it is no longer under consideration.

The search starts at node 0 in Figure 10-3(a). This corresponds to our island bridge building company setting up operations at its headquarters on its home island. The search removes this node from the priority queue, checks each of node 0's neighbors, and updates the information accordingly. Node 1 is assigned a distance of 1.0 and node 3 a distance of 0.6. Both neighbors' `last` values now point back to node 0 as the closest node in the connected subset.

In Figure 10-3(b), the search progresses to the closest node that is not in the connected subset. This corresponds to building the first bridge between islands. The algorithm dequeues node 3 with a distance (priority) of 0.6, adds it to the connected subset, and checks its neighbors 4 and 6. These are both newly reachable via an edge from node 3. The search updates both nodes' priorities and `last` values.

The search next explores node 1 in Figure 10-3(c). While checking the neighbors of node 1, it finds a shorter edge connecting to node 4. This is equivalent to the workers noticing the old wooden bridge (1, 4) is shorter and thus cheaper to upgrade than the other wooden bridge (3, 4) that is currently slated for an upgrade. The search thus updates the distance from node 4 to 0.5 and updates its `last` pointer to node 1 to reflect the origin of the connecting edge. The search is now scheduled to use the edge from (1, 4) to join node 4 to our connected set instead of the previous edge (3, 4).



(a)

node	dist	last
0	0.0	-1
1	1.0	0
2	inf	-1
3	0.6	0
4	inf	-1
5	inf	-1
6	inf	-1
7	inf	-1

(b)

node	dist	last
0	0.0	-1
1	1.0	0
2	inf	-1
3	0.6	0
4	1.5	3
5	inf	-1
6	1.0	3
7	inf	-1

(c)

node	dist	last
0	0.0	-1
1	1.0	0
2	1.2	1
3	0.6	0
4	0.5	1
5	0.2	1
6	1.0	3
7	inf	-1

(d)

node	dist	last
0	0.0	-1
1	1.0	0
2	0.4	5
3	0.6	0
4	0.5	1
5	0.2	1
6	1.0	3
7	1.4	5

(e)

node	dist	last
0	0.0	-1
1	1.0	0
2	0.4	5
3	0.6	0
4	0.5	1
5	0.2	1
6	1.0	3
7	1.4	5

(f)

node	dist	last
0	0.0	-1
1	1.0	0
2	0.4	5
3	0.6	0
4	0.5	1
5	0.2	1
6	1.0	3
7	1.4	5

(g)

node	dist	last
0	0.0	-1
1	1.0	0
2	0.4	5
3	0.6	0
4	0.5	1
5	0.2	1
6	1.0	3
7	0.3	6

(h)

node	dist	last
0	0.0	-1
1	1.0	0
2	0.4	5
3	0.6	0
4	0.5	1
5	0.2	1
6	1.0	3
7	0.3	6

Figure 10-3: An illustration of Prim's algorithm

In the next five subfigures, the search progresses to node 5, node 2, node 4, node 6, and node 7, respectively, checking each node's unvisited neighbors and updating any for which it finds shorter edges. The size of the connected subgraph grows by one each step until all nodes are connected.

Kruskal's Algorithm

An alternative to the node-by-node approach of Prim's algorithm is to take an edge-centric approach to constructing minimum spanning trees. Kruskal's algorithm, invented by multidisciplinary scholar Joseph B. Kruskal, works by looping over a sorted list of edge weights and progressively adding edges to build the minimum spanning tree. Intuitively, we want to add the graph's smaller edges, since they are the least expensive connections between nodes. If we maintain a list of edges sorted by weight, we can proceed through it, adding the next edge that would help build the minimum spanning tree. This loop over a sorted list forms the core of Kruskal's algorithm.

Kruskal's algorithm's cost scales proportional to $|E| \log (|E|)$. The algorithm starts by extracting and sorting each edge, requiring time proportional to $|E| \log (|E|)$. Using an efficient implementation of the union-find algorithm, we can combine the sets in $|E| \log (|V|)$ time. As long as $|E| \geq |V|$, the algorithm will scale as $|E| \log (|E|)$.

We can visualize Kruskal's algorithm in the context of a pet owner building a complex living space for their beloved hamster. The hamster already has several large habitats that the owner decides to connect using clear tubes, giving their pet free range to roam between cages. The habitats' arrangement within the room is fixed. The owner, looking to minimize the total tubing needed, measures each pairwise distance between habitats, sorts the list, and determines which tube to add next. Unlike the island building example, the pet owner does not need to worry about transporting construction equipment from node to node. They can easily move between any pair of nodes to build the connection.

Union-Find

Beyond finding the next lowest-cost edge, we need to answer one additional question when considering each new edge: does this edge join nodes from currently disconnected clusters? If not, the edge is redundant. Remember that the key word here is *minimum*. If we already have edges (A, B) and (B, C), the edge (A, C) doesn't help, as node C was already reachable from node A through node B.

To efficiently implement Kruskal's algorithm, we make use of a new helper data structure, `UnionFind`. This data structure allows us to represent a collection of different sets, which we will use to track the connected

components of the graph. The data structure facilitates a few efficient, set-based operations, including the following:

are_disjoint(i, j) Determines whether two elements **i** and **j** are in different sets. We use this to test whether two nodes are part of the same connected set.

union_sets(i, j) Merges the set with element **i** and the set with element **j** into a single set. We use this to connect two sets of nodes when adding an edge.

The data structure also tracks a count of the disjoint sets that is updated with each operation (`num_disjoint_sets`).

For the purposes of the algorithms in this book, it is not necessary to dive into the details of `UnionFind`. It is sufficient to treat it as a module that facilitates the operations described. Interested readers can find a basic description and the code sufficient to implement the algorithms in this book in Appendix C.

The Code

Given the helper data structure, the code for Kruskal's algorithm consists of two main steps. First, we create a list of all the graph's edges and sort it. Second, we iterate through that list by checking whether the current edge joins disconnected components and, if so, adding it to our minimum spanning tree:

```
def kruskals(g: Graph) -> Union[list, None]:
    djs: UnionFind = UnionFind(g.num_nodes)
    all_edges: list = []
    mst_edges: list = []

  ❶ for idx in range(g.num_nodes):
        for edge in g.nodes[idx].get_edge_list():
          ❷ if edge.to_node > edge.from_node:
                all_edges.append(edge)
  ❸ all_edges.sort(key=lambda edge: edge.weight)

    for edge in all_edges:
      ❹ if djs.are_disjoint(edge.to_node, edge.from_node):
            mst_edges.append(edge)
            djs.union_sets(edge.to_node, edge.from_node)

  ❺ if djs.num_disjoint_sets == 1:
        return mst_edges
    else:
        return None
```

The code starts by creating a series of helper data structures, including a `UnionFind` data structure representing the current disjoint sets (`djs`) to determine which points already belong to the same cluster, a list (`all_edges`) that will store the *sorted* list of edges, and an empty list (`mst_edges`) to hold the resulting edges for the minimum spanning tree. The code then loops over every node in the graph to fill these helper data structures ❶. For each node, it inserts each of the node's edges into the list of all edges.

Since our representation of an undirected graph includes the edge (A, B) in the adjacency lists for both node A and node B, the code uses a simple check to avoid adding the same edge twice ❷. (Note that this check is only needed to improve the efficiency when using this representation of an undirected graph. The code would still work correctly without the check but would include twice the number of edges in `all_edges`.)

After the full list of edges is assembled, the code sorts the edges in order of increasing weight ❸. The code iterates over each edge in the sorted list with a single `for` loop, then uses the `UnionFind` data structure to check whether the edge connects two currently unconnected components ❹. If so, the edge is useful. The code adds it to the set of edges from the minimum spanning tree (`mst_edges`) and merges the disjoint sets for the edge's nodes.

Finally, the code checks whether it was able to connect all the nodes into a single connected component ❺. If so, it returns the list of edges for the minimum spanning tree. Otherwise, it returns `None`. If we remove this final check, the code will instead return the edges from the individual minimum spanning trees for graphs that are not a single connected component.

An Example

Figure 10-4 shows an example of Kruskal's algorithm running on a graph with 8 nodes and 12 edges.

The search begins with an empty set of edges and thus a disconnected set of nodes. In Figure 10-4(a), the search selects the edge with the lowest weight from our graph. This corresponds to the edge (1, 5) with a weight of 0.2. The edge in the figure is marked in bold to indicate it is part of the minimum-cost spanning tree. Nodes 1 and 5 are now part of the same connected subset, and the search has reduced the number of disjoint sets from eight to seven.

The search continues in Figure 10-4(b) by choosing the edge with the next lowest weight. This time it connects nodes 6 and 7 through an edge with weight 0.3. It has reduced the number of disjoint sets to six.

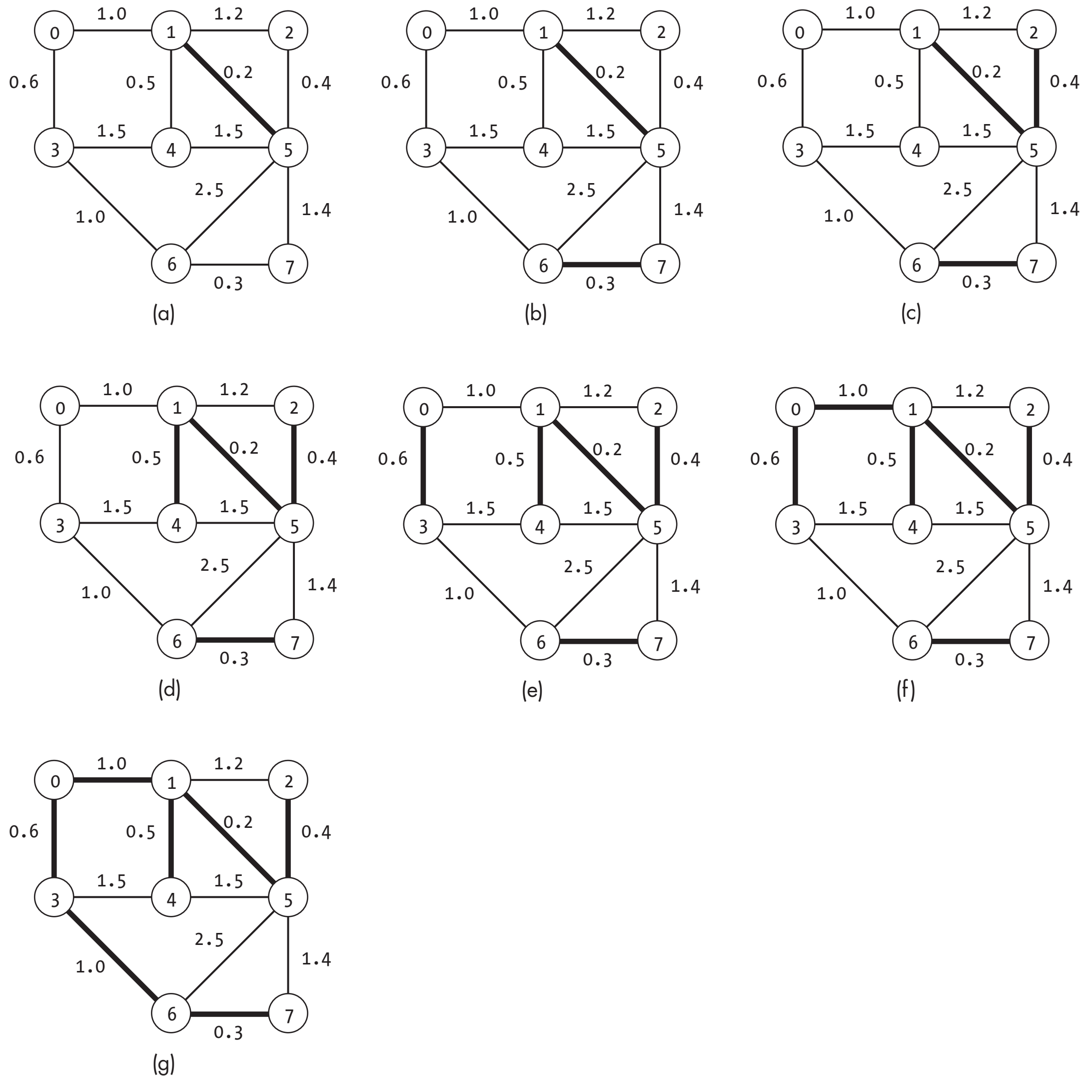


Figure 10-4: An illustration of Kruskal's algorithm

In the next two subfigures, the search adds nodes 2 and 4 to the first connected subset {1, 5}, resulting in a connected set consisting of {1, 2, 4, 5}. In Figure 10-4(e), the algorithm merges another two singleton nodes by joining nodes 0 and 3 via the edge with weight 0.6. It then joins up the remaining three disjoint sets by adding the edges (0, 1) and (3, 6) in the following two subfigures. At this point, we are down to a single set, which means our minimum-cost spanning tree edges connect all the nodes in the graph.

Maze Generation

While the graph searches presented in preceding chapters allow us to algorithmically solve mazes, they cannot help us generate mazes in the first place. In this section, we take a detour from the more canonical uses of minimum spanning tree algorithms, such as building transportation networks, to show how we can extend Kruskal's algorithm to create random but always solvable mazes. To make the mazes sufficiently fun, we ensure that each has exactly one valid solution.

Imagine we are given the task of generating a maze for the children's place mat at a local family restaurant. Our design can be simple but must be solvable, with only one path through the maze. The restaurant owners wisely do not want to challenge young patrons with impossible mazes, lest this results in screaming and thrown food.

Representing Grid-Based Mazes

For simplicity of the code in this section, we represent our mazes using a regular grid of squares like the ones on graph paper. After hours of careful consideration about how to draw our mazes, we decide to shade individual edges to represent the maze's walls. The player can move between any two adjacent squares that do not have a wall between them. As we draw each line, we eliminate a potential option for leaving that square and perhaps chuckle at the difficult task we are creating.

Figure 10-5(a) shows an example grid-based maze. We can equivalently represent this grid structure using a graph, as shown in Figure 10-5(b).

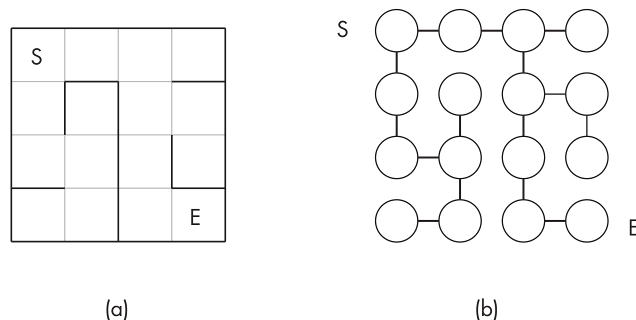


Figure 10-5: A grid-based maze and its graph representation

In Figure 10-5(b), each square in the maze corresponds to a single graph node. We add undirected edges between any two adjacent nodes without a wall so that an edge indicates the ability to travel from one node to another.

Generating Mazes

We construct our maze by starting with a grid-based graph and building a randomized spanning tree algorithm based on Kruskal's algorithm to connect all the nodes. The grid-based initial structure gives us connections based on adjacency. Each node has up to four connections to the nodes

above, below, left, and right of it. Generating a spanning tree allows us to ensure that each node is reachable from any other node and that we can reach the ending node from the starting one.

We define the valid edges using a connected grid-based graph, as shown in Figure 10-6. Like the grids we generated in Chapter 5, this graph represents all the nodes we need to connect and the set of potential edges we can use to connect them. If our grid has a width of w and a height of h, it contains $h \times w$ nodes and undirected edges (with equal weights of 1) connecting neighboring nodes.

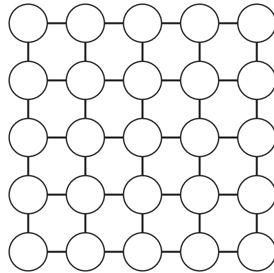


Figure 10-6: A grid-based graph

If we used the graph in Figure 10-6 for our final maze, there would be a huge number of potential paths between any two locations. In other words, the graph does not make for a particularly fun or challenging maze. Beginning at the start node, we could traverse directly to the end node by moving the minimum number of steps in one horizontal and one vertical direction. Real-world equivalents would be a hedge maze implemented as an empty lawn or a blank maze on our place mat. To construct an interesting maze, we need to use a minimum subset of these edges.

As in Kruskal's algorithm, we start with an empty spanning tree where none of the nodes are connected. In the case of our grid-based place mat, we start with a grid of boxes. One by one, we add edges to our spanning tree and erase the lines between adjacent boxes. We can alternatively visualize the connection of two components as a cartoon figure removing a physical wall between two adjacent rooms by using an oversized sledgehammer or simply bursting through the wall. As our cartoon character gleefully opens up passageways (or we carefully erase grid lines), the disparate components connect and a path through the maze forms.

The key to generating a random maze is, intuitively, to choose the next edge randomly. Both Kruskal's and Prim's algorithms rely on some method to break ties among equal-weight edges. In this case, however, all edges have the same edge weight (1.0), so we can just pick one at random. If the chosen edge connects two disjoint components, we keep it. This edge opens a path between two previously unreachable components. Otherwise, in the case where the chosen edge joins two already connected components, we discard it, since adding multiple paths between components would result in loops and break the maze convention of having a single path.

The Code

The following code allows us to randomly create a set of maze edges from a grid-based graph:

```
def randomized_kruskals(g: Graph) -> list:
  ❶ djs: UnionFind = UnionFind(g.num_nodes)
    all_edges: list = []
    maze_edges: list = []

  ❷ for idx in range(g.num_nodes):
        for edge in g.nodes[idx].get_edge_list():
            if edge.to_node > edge.from_node:
                all_edges.append(edge)

  ❸ while djs.num_disjoint_sets > 1:
        num_edges: int = len(all_edges)
      ❹ edge_ind: int = random.randint(0, num_edges - 1)
        new_edge: Edge = all_edges.pop(edge_ind)

      ❺ if djs.are_disjoint(new_edge.to_node, new_edge.from_node):
            maze_edges.append(new_edge)
            djs.union_sets(new_edge.to_node, new_edge.from_node)

    return maze_edges
```

The function takes a full grid-based graph (`g`) to define the list of edges. The code starts by setting up helper data structures, including a `UnionFind` data structure representing the disjoint sets (`djs`), a list of all edges (`all_edges`), and a list of the maze or spanning tree edges (`maze_edges`) ❶. As in Kruskal's algorithm, the code extracts the comprehensive list of edges from the graph ❷.

The algorithm iterates through a single `while` loop until all nodes are the same set (and thus reachable) ❸. During each iteration of the loop, the algorithm selects an edge randomly ❹, using Python's `random` library's `randint()` function (which requires us to include `import random` at the start of the file). It then removes the selected edge from the list of all edges and checks whether it joins two previously disjoint sets ❺. If so, the edge is added to the list of maze edges and the corresponding sets are merged. Otherwise, the edge is ignored. The algorithm completes after all the nodes are merged into a single set, returning the list of edges that defines the maze: the minimum spanning tree.

An Example

Figure 10-7 shows an example of the first few steps of this algorithm. The left diagram of each subfigure shows the current maze as defined by the walls that have been removed, while the right diagram shows the maze as defined by edges that have been added to a graph. During each step (each iteration of the `while` loop), one edge at most is added.

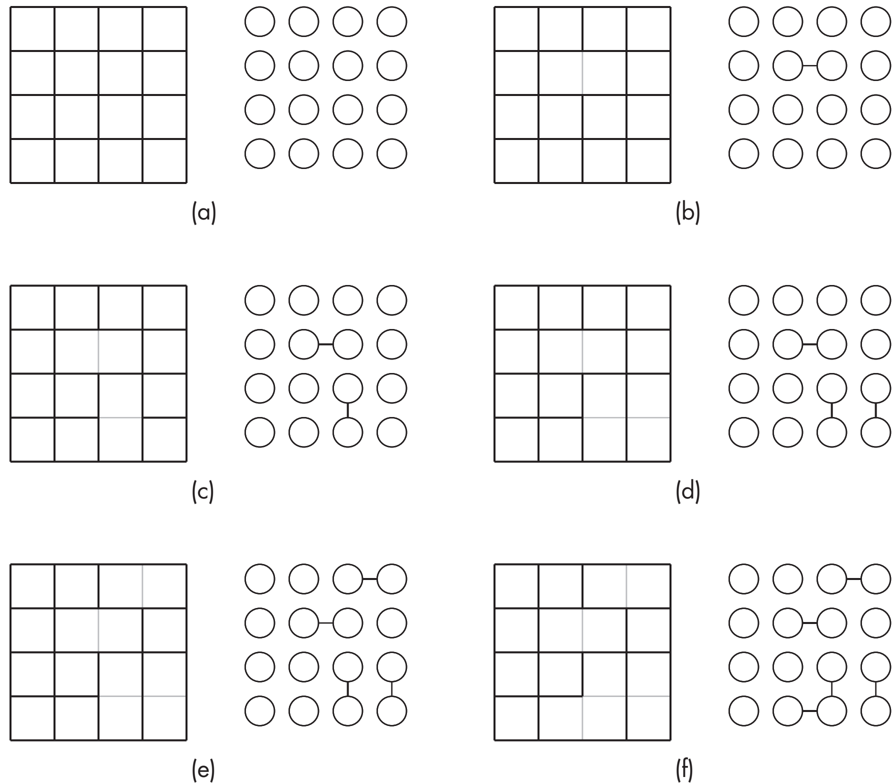


Figure 10-7: Six steps of the maze construction algorithm

It is not strictly necessary to construct the full grid-based graph (`g`) ahead of time. Instead, we could just programmatically fill the `all_edges` list based on computed adjacencies, as we did when constructing grids in Chapter 5, for example. However, for the purposes of this chapter, starting with the full grid-based graph makes the code's connection to Kruskal's algorithm more apparent and keeps the function simpler.

The randomized Kruskal's algorithm is a simplistic approach to generating mazes that makes no guarantee that the ending node is at the end of a deep path with a bunch of turns. It may result in quite boring mazes such as the ones shown in Figures 10-8(a), 10-8(b), and 10-8(c). We can only be sure that the algorithm will *not* produce a maze where the end is unreachable, such as the one shown in Figure 10-8(d).

Graph Algorithms the Fun Way (Sample Chapter) © 2024 by Jeremy Kubica

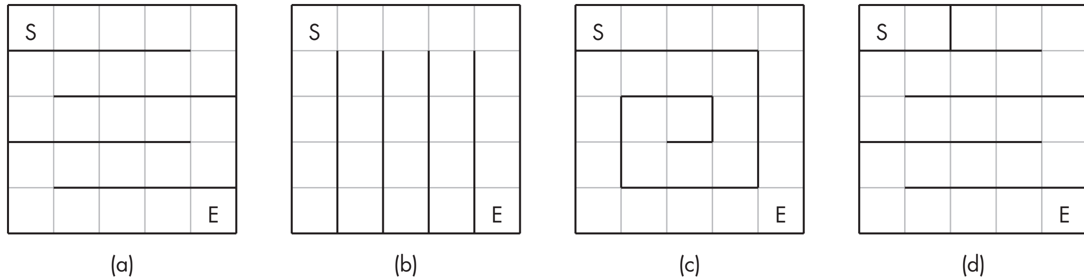


Figure 10-8: Three overly simple mazes and one unsolvable maze

Beyond the exciting commercial opportunities involved in designing children's place mats, the maze-generation algorithm in this section shows how we can extend the basic components of minimum spanning trees and Kruskal's algorithm. Further, it demonstrates how randomization can be used within an algorithm to create different spanning trees.

Single-Linkage Hierarchical Clustering

We can also adapt Kruskal's algorithm to handle the seemingly different problem of clustering spatial points. *Clustering* is a common unsupervised data-mining and machine-learning approach that assigns data points to clusters such that the points within each cluster are similar (for some given definition of similar). For example, we might cluster cafés based on geographic proximity so that all the coffee shops in Anchorage are placed together in one cluster, while cafés in Honolulu are placed in another. The resulting clusters provide a partitioning of data points that can help us discover structure in the data or classify similar data points.

There is a wide range of clustering techniques that vary in how they define similar points and how points are assigned to clusters. As its name implies, hierarchical clustering is an approach that creates a hierarchy of clusters by merging two "nearby" clusters at each level of the hierarchy. Each data point initially defines its own cluster; these clusters are iteratively joined until all the points are part of the same cluster. Even within hierarchical clustering, there are various approaches to determining which clusters to merge, including the following:

- Computing the mean position over each cluster's points and merging the clusters with the closest centers
- Finding the farthest of any pair of points from two clusters and merging the clusters whose maximum distance is the smallest
- Finding the closest pair of points from two clusters and merging the clusters whose minimum distance is the smallest

This section focuses on the last approach, called *single-linkage clustering*, which joins the two clusters with the closest pair of individual points. We

present an algorithm to implement it that is nearly identical to Kruskal's algorithm on graphs.

Figure 10-9 shows an example of single-linkage clustering. The left-hand figure shows the five two-dimensional points (0, 0), (1, 0), (1.2, 1), (1.8, 1), and (0.5, 1.5). The right-hand figure shows the hierarchical clustering.

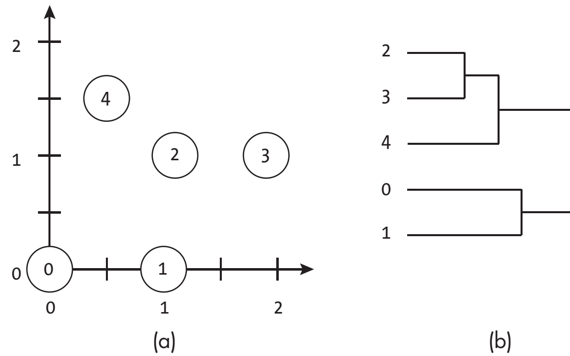


Figure 10-9: A set of two-dimensional points (left) and the corresponding single-linkage clustering (right)

We start with each point in its own cluster and create a merged cluster from the two individual points with the closest distance (points 2 and 3). Next, we merge the cluster {2, 3} with {4} because points 2 and 4 have the smallest distance of any pair of points in different clusters. This process continues as shown in the right-hand side of Figure 10-9.

The advantage of hierarchical clustering is that it provides an easily visualized and interpretable structure. We can use this structure to dynamically change the number of clusters (level of partitioning) by walking up the hierarchy until we hit a given distance threshold. Points that have been joined together before we hit the threshold are clustered together, while clusters that have not been merged remain disjoint.

The Code

To simplify the logic of the clustering code, we define two small helper classes that store information about the points and the resulting clustering links. First, to represent the two-dimensional points we are clustering, we define a `Point` class to store the coordinates and compute pairwise distances:

```
class Point:
    def __init__(self, x: float, y: float):
        self.x: float = x
        self.y: float = y

    def distance(self, b) -> float:
        diff_x: float = (self.x - b.x)
        diff_y: float = (self.y - b.y)
        dist: float = math.sqrt(diff_x*diff_x + diff_y*diff_y)
        return dist
```

Graph Algorithms the Fun Way (Sample Chapter) © 2024 by Jeremy Kubica

The distance() function computes the Euclidean distance in two-dimensional space and requires us to include import math at the start of the file in order to use the math library's square root function. (Appendix A further discusses creating graphs from spatial points, including the use of alternative distance functions.)

Second, since we are not using an explicit graph, we also define a Link data structure to hold the connection between points in the same cluster:

```
class Link:
    def __init__(self, dist: float, id1: int, id2: int):
        self.dist: float = dist
        self.id1: int = id1
        self.id2: int = id2
```

This data structure is effectively identical to an undirected graph edge. It stores a pair of identifiers for the points and the distance (weight) between them. We define it here as an independent data structure to highlight the fact that we do not need to explicitly build a graph for single-linkage clustering.

Using these two helper data structures, we can then implement the single-linkage hierarchical clustering algorithm using an approach based on Kruskal's algorithm:

```
def single_linkage_clustering(points: list) -> list:
    num_pts: int = len(points)
    djs: UnionFind = UnionFind(num_pts)
    all_links: list = []
    cluster_links: list = []

  ❶ for id1 in range(num_pts):
        for id2 in range(id1 + 1, num_pts):
            dist = points[id1].distance(points[id2])
            all_links.append(Link(dist, id1, id2))

  ❷ all_links.sort(key=lambda link: link.dist)

    for x in all_links:
      ❸ if djs.are_disjoint(x.id1, x.id2):
            cluster_links.append(x)
            djs.union_sets(x.id1, x.id2)

    return cluster_links
```

The code takes a list of Point objects (points) to cluster. The function starts by creating a series of helper data structures, including a UnionFind data structure representing the disjoint sets (djs) to determine which points already belong to the same cluster, an empty list (all_links) to hold all pairwise distances, and an empty list (cluster_links) to hold the Link objects representing each merge. The code then uses a nested pair of for loops to iterate through all pairs of points ❶. For each pair, the code computes the distance using the points' distance function and creates a Link data

structure to hold this distance information. After all the pairwise distances are computed, the code sorts the links in order of increasing distance ❷.

Next, another `for` loop iterates over each edge in the sorted list, using the `UnionFind` data structure to check whether the next pair of points is already in the same cluster ❸. If not, the program adds the link to `cluster_links`, joining the two clusters that contain those points, and merges the disjoint sets for the points.

Finally, the code returns the list of `Link` objects representing the clustering. Each `Link` represents a connection between two previously disjoint clusters. The links in `cluster_links` will be ordered by increasing distance, so the first element represents the first two points merged.

An Example

Figure 10-10 shows the steps of our clustering algorithm on the points from Figure 10-9. The left column of the figure shows the current clusters as connected graph components of the two-dimensional points. The right column of the figure shows the same clusters as merged points in the hierarchy with each cluster represented as a circle.

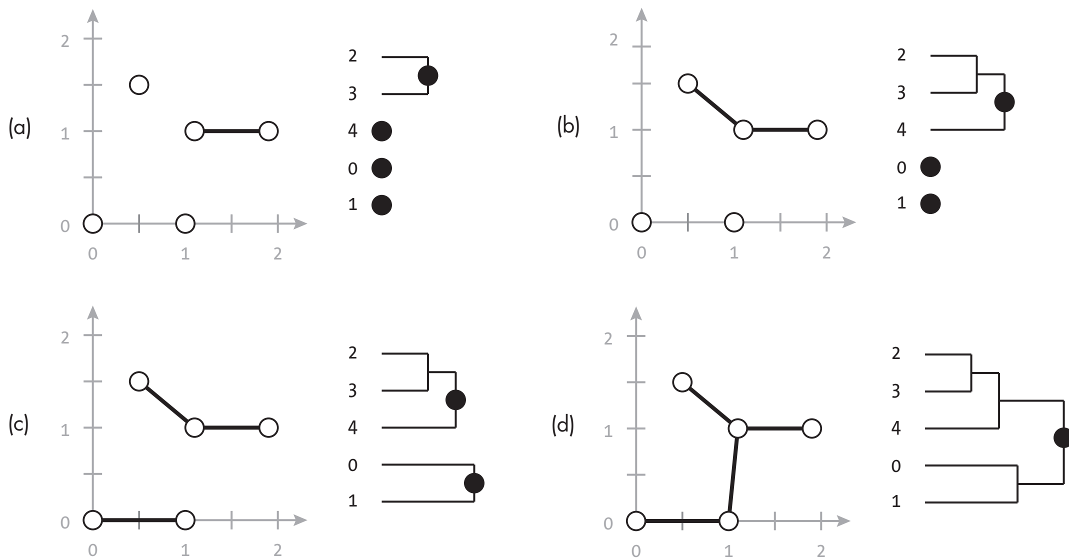


Figure 10-10: Single-linkage clustering

In Figure 10-10(a), the algorithm has joined the closest two points—those at (1.2, 1) and (1.8, 1)—into a single cluster. Using the points' labels from Figure 10-9, we call these points 2 and 3, respectively.

In the next step, in Figure 10-10(b), the algorithm joins the two clusters with the closest pair of points. At this stage, the closest points are (1.2, 1) and (0.5, 1.5) with a distance of approximately 0.86. Since (1.2, 1) is already part of a cluster, the algorithm merges the entire cluster with the one

Graph Algorithms the Fun Way (Sample Chapter) © 2024 by Jeremy Kubica

containing the single point (0.5, 1.5). The resulting cluster contains three points {2, 3, 4}.

The algorithm continues in Figure 10-10(c) by creating a new merged cluster from the two remaining individual points (0, 0) and (1, 0). The algorithm has now created two separate clusters with three and two points, respectively. During the final step, in Figure 10-10(d), these two clusters are merged by adding a link between the closest pair of points from each cluster (1.2, 1) and (1, 0).

Since single-linkage clustering grows the clusters by linking increasingly distant pairs of points, we can use this distance as a stopping threshold for the algorithm. For example, if we set the maximum distance to 0.95, we would produce the three distinct clusters shown in Figure 10-10(b).

Why This Matters

The minimum spanning tree problem allows us to solve a range of real-world optimization problems, from building roads to designing communication networks. Within the field of computer science, we can use minimum spanning trees to help solve a range of problems in networking, clustering, and analysis of biological data. For example, we can represent a communication network as a graph and find the minimum spanning tree to inform which links need to be upgraded to ensure that all nodes are reachable through the new technology.

We can also apply the same basic approach to problems we might not normally think of as graph based. Using a variation of Kruskal's algorithm, we can search for structure in real-valued datasets by building clusters of similar data points or design solvable mazes by introducing randomization into the algorithm to create novel solutions. In single-linkage clustering, we use the distances to determine which points are similar.

The next chapter expands on this discussion, introducing algorithms that help us identify the nodes and edges that are essential to maintaining connectivity.