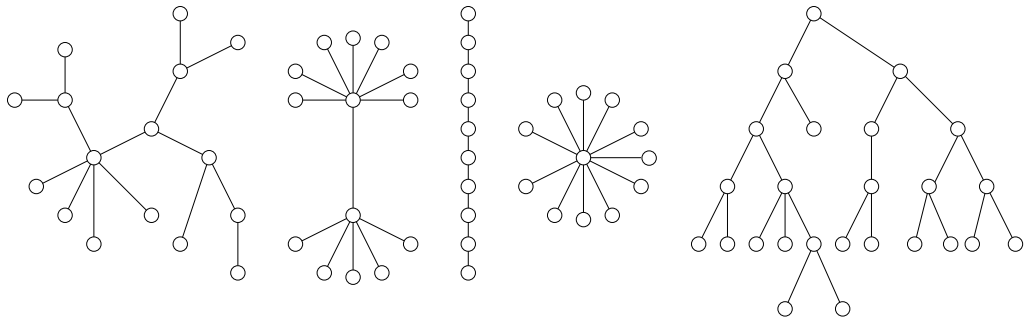

Figure 21: Five trees

10 Trees

We have met trees when studying enumeration problems; now we have a look at them as graphs. A graph $G = (V, E)$ is called a *tree* if it is connected and contains no cycle as a subgraph. The simplest tree has one node and no edges. The second simplest tree consists of two nodes connected by an edge. Figure 21 shows a variety of other trees.

Note that the two properties defining trees work in the opposite direction: connectedness means that the graph cannot have "too few" edges, while the exclusion of cycles means that it cannot have "too many". To be more precise, if a graph is connected, then adding a new edge to it, it remains connected (while deleting an edge, it may or may not stay connected). If a graph contains no cycle, then deleting any edge, the remaining graph will not contain a cycle either (while adding a new edge may or may ot create a cycle). The following theorem shows that trees can be characterized as "minimally connected" graphs as well as "maximally cycle-free" graphs.

Theorem 10.1 *(a) A graph G is a tree if and only if it is connected, but deleting any of its edges results in a disconnected graph.*

(b) A graph G is a tree if and only if it contains no cycles, but adding any new edge creates a cycle.

We prove part (a) of this theorem; the proof of part (b) is left as an exercise.

First, we have to prove that if G is a tree then it satisfies the condition given in the theorem. It is clear that G is connected (by the definition of a tree). We want to prove that deleting any edge, it cannot remain connected. The proof is indirect: assume that deleting the edge uv from a tree G, the remaining graph G' is connected. Then G' contains a path P connecting u and v. But then, if we put the edge uv back, the path P and the edge uv will form a cycle in G, which contradicts the definition of trees.

Second, we have to prove that if G satisfies the condition given in the theorem, then it is a tree. It is clear that G is connected, so we only have to argue that G does not contain a cycle. Again by an indirect argument, assume that G does contain a cycle C. Then deleting any edge of C, we obtain a connected graph (exercise 9.2). But this contradicts the condition in the theorem.

10.1 Prove part (b) of theorem 10.1.

10.2 Prove that connecting two nodes u and v in a graph G by a new edge creates a new cycle if and only if u and v are in the same connected component of G.

10.3 Prove that in a tree, every two nodes can be connected by a *unique* path. Conversely, prove that if a graph G has the property that every two nodes can be connected by a path, and there is only one connecting path for each pair, then the graph is a tree.

10.1 How to grow a tree?

The following is one of the most important properties of trees.

Theorem 10.2 *Every tree with at least two nodes has at least two nodes of degree 1.*

Let G be a tree with at least two nodes. We prove that G has a node of degree 1, and leave it to the reader as an exercise to prove that it has at least one more. (A path has only two such nodes, so this is the best possible we can claim.)

Let us start from any node v_0 of the tree and take a walk (climb?) on the tree. Let's say we never want to turn back from a node on the edge through which we entered it; this is possible unless we get to a node of degree 1, in which case we stop and the proof is finished.

So let's argue that this must happen sooner or later. If not, then eventually we must return to a node we have already visited; but then the nodes and edges we have traversed between the two visits form a cycle. This contradicts our assumption that G is a tree and hence contains no cycle.

10.4 Apply the argument above to find a second node of degree 1.

A real tree grows by developing a new twig again and again. We show that graph-trees can be grown in the same way. To be more precise, consider the following procedure, which we call the *Tree-growing Procedure*:

— *Start with a single node.*

— *Repeat the following any number of times: if you have any graph G, create a new node and connect it by a new edge to any node of G.*

Theorem 10.3 *Every graph obtained by the Tree-growing Procedure is a tree, and every tree can be obtained this way.*

The proof of this is again rather straightforward, but let us go through it, if only to gain practice in arguing about graphs.

First, consider any graph that can be obtained by this procedure. The starting graph is certainly a tree, so it suffices to argue that we never create a non-tree; in other words, if G is a tree, and G' is obtained from G by creating a new node v and connecting it to a node u of G, then G' is a tree. This is straightforward: G' is connected, since any two "old" nodes can be connected by a path in G, while v can be connected to any other node w by first going to u and then connecting u to w. Moreover, G cannot contain a cycle: v has degree 1 and so no cycle can go through v, but a cycle that does not go through v would be a cycle in the old graph, which is supposed to be a tree.

Second, let's argue that every tree can be constructed this way. We prove this by induction on the number of nodes.[4] If the number of nodes is 1, then the tree arises by the

[4]The first part of the proof is also an induction argument, even though it was not phrased as such.

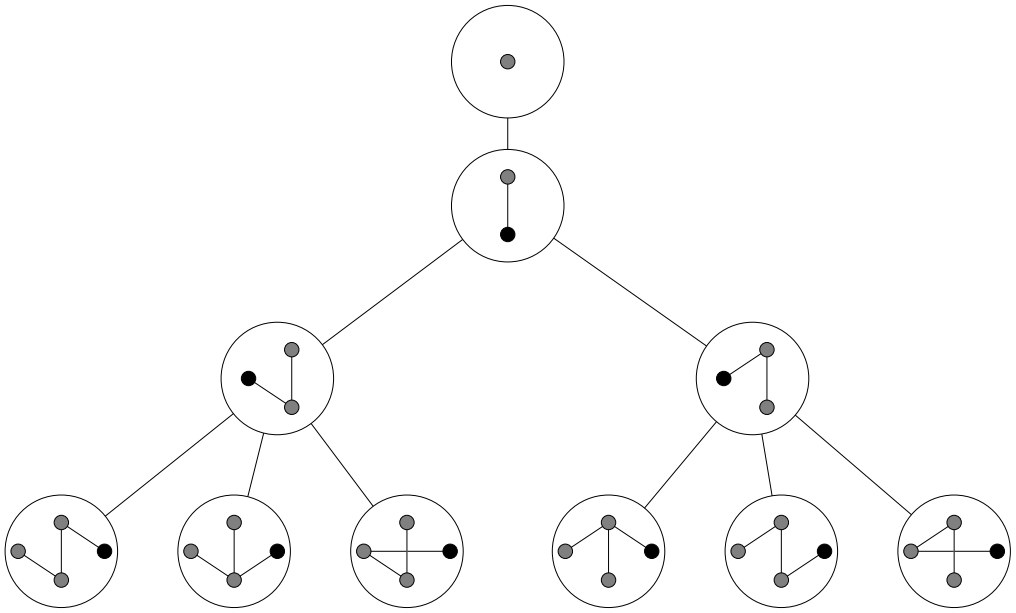

Figure 22: The descendence tree of trees

construction, since this is the way we start. Assume that G is a tree with at least 2 nodes. Then by theorem 10.2, G has a node of degree 1 (at least two nodes, in fact). Let v be a node with degree 1. Delete v from G, together with the edge with endpoint v, to get a graph G'.

We claim that G' is a tree. Indeed, G' is connected: any two nodes of G' can be connected by a path in G, and this path cannot go through v as v has degree 1. So this path is also a path in G'. Furthermore, G' does not contain a cycle as G does not.

By the induction hypothesis, every tree with fewer nodes than G arises by the construction; in particular, G' does. But then G arises from G' by one more iteration of the second step. This completes the proof of Theorem 10.3.

Figure 22 shows how trees with up to 4 nodes arise by this construction. Note that there is a "tree of trees" here. The fact that the logical structure of this construction is a tree does not have anything to do with the fact that we are constructing trees: any iterative construction with free choices at each step results in a similar "descendence tree".

The Tree-growing Procedure can be used to establish a number of properties of trees. Perhaps most important of these concerns the number of edges. How many edges does a tree have? Of course, this depends on the number of nodes; but surprisingly, it depends *only* on the number of nodes:

Theorem 10.4 *Every tree on n nodes has $n-1$ edges.*

Indeed, we start with one more node (1) than edge (0), and at each step, one new node and one new edge is added, so this difference of 1 is maintained.

10.5 Let G be a tree, which we consider as the network of roads in a medieval country, with castles as nodes. The King lives at node r. On a certain day, the lord of each

castle sets out to visit the King. Argue carefully that soon after they leave their castles, there will be exactly one lord on each edge. Give a proof of Theorem 10.4 based on this.

10.6 If we delete a node v from a tree (together with all edges that end there), we get a graph whose connected components are trees. We call these connected components the *branches* at node v. Prove that every tree has a node such that every branch at this node contains at most half the nodes of the tree.

10.2 Rooted trees

Often we use trees that have a special node, which we call the *root*. For example, trees that occured in counting subsets or permutations were built starting with a given node.

We can take any tree, select any of its nodes, and call it a *root*. A tree with a specified root is called a *rooted tree*.

Let G be a rooted tree with root r. Given any node v different from r, we know that the tree contains a unique path connecting v to r. The node on this path next to v is called the *father* of v. The other neighbors of v are called the *sons* of v. The root r does not have a father, but all its neighbors are called its sons.

Now a basic geneological assertion: *every node is the father of its sons*. Indeed, let v be any node and let u be one of its sons. Consider the unique path P connecting v to r. The node cannot lie on P: it cannot be the first node after v, since then it wold be the father of v, and not its son; and it cannot be a later node, since then going from v to u on the path P and then back to v on the edge uv we would traverse a cycle. But this implies that adding the node u and the edge uv to P we get a path connecting u to r. Since v is the forst node on this path after u, it follows that v is the father of u. (Is this argument valid when $v = r$? Check!)

We have seen that every node different from the root has exactly one father. A node can have any number of sons, including zero. A node with no sons is called a *leaf*. In other words, a leaf is a node with degree 1, different from r. (To be precise, if the tree consists of a single node r, then this is a leaf.)

10.3 How many trees are there?

We have counted all sorts of things in the first part of this book; now that we are familiar with trees, it is natural to ask: *how many trees are there on n nodes?*

Before attempting to answer this question, we have to clarify an important issue: when do we consider two trees different? There are more than one reasonable answers to this question. Consider the trees in Figure 23. Are they the same? One could say that they are; but then, if the nodes are, say, towns, and the edges represent roads to be built between them, then clearly the inhabitants of the towns will consider the two plans very different.

So we have to define carefully when we consider two trees the same. The following are two possibilities:

— We fix the set of nodes, and consider two trees the same if the same pairs of nodes are connected in each. (This is the position the twon people would take when they consider road construction plans.) In this case, it is advisable to give names to the nodes, so that

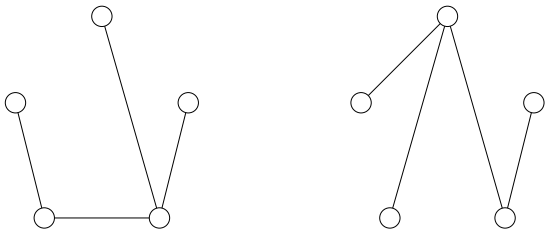

Figure 23: Are these trees the same?

we can distinguish them. It is convenient to use the numbers $0, 1, 2, \ldots, n-1$ as names (if the tree has n nodes). We express this by saying that the vertices of the tree are labeled by $0, 1, 2, \ldots n-1$. Figure 24 shows a labelled tree. Interchanging the labels 2 and 4 (say) would yield a different labelled tree.

— We don't give names to the nodes, and consider two trees the same if we can rearrange the nodes of one so that we get the other tree. More exactly, we consider two trees the same (the mathematical term for this is *isomorphic*) if there exists a one-to-one correpondence between the nodes of the first tree and the nodes of the second tree so that two nodes in the first tree that are connected by an edge correspond to nodes in the second tree that are connected by an edge, and vice versa. If we speak about *unlabelled trees*, we mean that we don't distinguish isomorphic trees from each other. For example, all paths on n nodes are the same as unlabelled trees.

So we can ask two questions: how many labelled trees are there on n nodes? and how many unlabelled trees are there on n nodes? These are really two different questions, and we have to consider them separately.

10.7 Find all unlabelled trees on $2, 3, 4$ and 5 nodes. How many labelled trees do you get from each? Use this to find the number of labelled trees on $2, 3, 4$ and 5 nodes.

10.8 How many labelled trees on n nodes are stars? How many are paths?

The number of labelled trees. For the case of labelled trees, there is a very nice solution.

Theorem 10.5 (Cayley's Theorem) *The number of labeled trees on n nodes is n^{n-2}.*

The formula is elegant, but the surprising fact about it is that it is quite difficult to prove! It is substantially deeper than any of the previous formulas for the number of this and that. There are various ways to prove it, but each uses some deeper tool from mathematics or a deeper idea. We'll give a proof that is perhaps best understood by first discussing a quite different question in computer science: how to store a tree?

10.4 How to store a tree?

Suppose that you want to store a labelled tree, say the tree in Figure 24, in a computer. How would you do this? Of course, the answer depends on what you need to store the tree for, what information about it you want to retrieve and how often, etc. Right now, we are

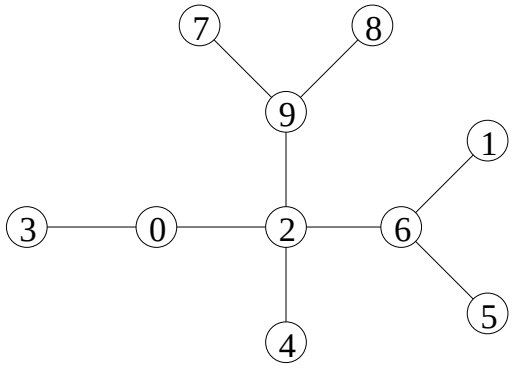

Figure 24: A labelled tree

only concerned with the amount of memory we need. We want to store the tree so that it occupies the least amount of memory.

Let's try some simple solutions.

(a) Suppose that we have a tree G with n nodes. One thing that comes to mind is to make a big table, with n rows and n columns, and put (say) the number 1 in the j-th position of the i-th row if nodes i and j are connected by an edge, and the number 0, if they are not:

$$\begin{array}{cccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array} \tag{20}$$

We need one bit to store each entry of this table, so this takes n^2 bits. We can save a little by noticing that it is enough to store the part below the diagonal, since the diagonal is always 0 and the other half of the table is just the reflection of the half below the diagonal. But this is still $(n^2 - n)/2$ bits.

This method of storing the tree can, of course, be used for any graph. It is often very useful, but, at least for trees, it is very wasteful.

(b) We fare better if we specify each tree by listing all its edges. We can specify each edge by its two endpoints. It will be convenient to arrange this list in an array whose columns correspond to the edges. For example, the tree in Figure 24 can be encoded by

$$\begin{array}{ccccccccc}
7 & 8 & 9 & 6 & 3 & 0 & 2 & 6 & 6 \\
9 & 9 & 2 & 2 & 0 & 2 & 4 & 1 & 5
\end{array}$$

Instead of a table with n rows, we get a table just with two rows. We pay a little for this: instead of just 0 and 1, the table wil contain integers between 0 and $n - 1$. But this is

certainly worth it: even if we count bits, to write down the label of a node takes $\log_2 n$ bits, so the whole table occupies only $2n \log_2 n$ bits, which is much less than $(n^2 - n)/2$ if n is large.

There is still a lot of free choice here, which means that the same tree may be encoded in different ways: we have freedom in choosing the order of the edges, and also in choosing the order in which the two endpoints of an edge are listed. We could agree on some arbitrary conventions to make the code well defined (say, listing the two endnodes of an edge in increasing order, and then the edges in increasing order of their first endpoints, breaking ties according to the second endpoints); but it will be more useful to do this in a way that also allows us to save more memory.

(c) **The father code.** From now on, the node with label 0 will play a special role; we'll consider it the "root" of the tree. Then we can list the two endnodes of an edge by listing the endpoint further from the root first, and then the endpoint nearer to the root second. So for every edge, the node written below is the father of the node written above. For the order in which we list the edges, let us take the order of their first nodes. For the tree in Figure 24, we get the table

1	2	3	4	5	6	7	8	9
6	0	0	2	6	2	9	9	2

Do you notice anything special about this table? The first row consists of the numbers $1, 2, 3, 4, 5, 6, 7, 8, 9$, in this order. Is this a coincidence? Well, the order is certainly not (we ordered the edges by the increasing order of their first endpoints), but why do we get every number exactly once? After a little reflection, this should be also clear: if a node occurs in the first row, then its father occurs below it. Since a node has only one father, it can occur only once. Since every node other than the root has a father, every node other than the root occurs in the first row.

Thus we know in advance that if we have a tree on n nodes, and write up the array using this method, then the first row will consist of $1, 2, 3, \ldots, n-1$. So we may as well suppress the first row without loosing any information; it suffices to store the second. Thus we can specify the tree by a sequence of $n-1$ numbers, each between 0 and $n-1$. This takes $(n-1)\lceil \log_2 n \rceil$ bits.

This coding is not optimal, in the sense that not every "code" gives a tree (see exercise 10.4). But we'll see that this method is already nearly optimal.

10.9 Consider the following "codes": (0, 1, 2, 3, 4, 5, 6, 7); (7, 6, 5, 4, 3, 2, 1, 0); $(0,0,0,0,0,0,0,0)$; $(2,3,1,2,3,1,2,3)$. Which of these are "father codes" of trees?

10.10 Prove, based on the "father code" method of storing trees, that the number of labelled trees on n nodes is at most n^{n-1}.

(d) Now we describe a procedure, the so-called *Prüfer code*, that will assign to any n-point labeled tree a sequence of length $n-2$, not $n-1$, consisting of the numbers $0, \ldots, n-1$. The gain is little, but important: we'll show that every such sequence corresponds to a tree. Thus we will establish a *bijection*, a one-to-one correspondence between labelled trees on n nodes and sequences of length $n-2$, consisting of numbers $0, 1, \ldots, n-1$. Since the number of such sequences is n^{n-2}, this will also prove Cayley's Theorem.

The Prüfer code can be considered as a refinement of method (c). We still consider 0 as the root, we still order the two endpoints of an edge so that the father comes first, but we order the edges (the columns of the array) not by the magnitude of their first endpoint but a little differently, more closely related to the tree itself.

So again, we construct a table with two rows, whose columns correspond to the edges, and each edge is listed so that the node farther from 0 is on the top, its father on the bottom. The issue is the order in which we list the edges.

Here is the rule for this order: we look for a node of degree 1, different from 0, with the smallest label, and write down the edge with this endnode. In our example, this means that we write down $\binom{1}{6}$. Then we delete this node and edge from the tree, and repeat: we look for the endnode with smallest label, different from 0, and write down the edge incident with it. In our case, this means adding a column $\binom{3}{0}$ to the table. Then we delete this node and edge etc. We go until all edges are listed. The array we get is called the *extended Prüfer code* of the tree (we call it extended because, as we'll see, we only need a part of it as the "real" Prüfer code). The extended Prüfer code of the tree in Figure 24 is:

$$\begin{array}{ccccccccc} 1 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 2 \\ 6 & 0 & 2 & 6 & 2 & 9 & 9 & 2 & 0 \end{array}$$

Why is this any better than the "father code"? One little observation is that the last entry in the second row is now always 0, since it comes from the last edge and since we never touched the node 0, this last edge must be incident with it. But we have payed a lot for this, it seems: it is not clear any more that the first row is superfluous; it still consists of the numbers $1, 2, \ldots, n-1$, but now they are not in increasing order any more.

The key lemma is that the first row is determined by the second:

Lemma 10.1 *The second row of an extended Prüfer code determines the first.*

Let us illustrate the proof of the lemma by an example. Suppose that somebody gives us the second row of an extended Prüfer code of a labelled tree on 8 nodes; say, 2403310 (we have one fewer edges than nodes, so the second row consists of 7 numbers, and as we have seen, it must end with a 0). Let us figure out what the first row must have been.

How does the first row start? Remember that this is the node that we delete in the first step; by the rule of cinstructing the Prüfer code, this is the node with degree 1 with smallest label. Could this node be the node 1? tree? No, because then we would have to delete it in the first step, and it could not occur any more, but it does. By the same token, no number occuring in the second row could be a leaf of the tree at the beginning. This rules out 2, 3 and 4.

What about 5? It does not occur in the second row; does this mean that it is a leaf of the original tree? The answer is yes; else, 5 would have been the father of some other node, and it would have been written in the second row when this other node was deleted. Thus 5 was its leaf with smallest label, and the first row of the extended Prüfer code must start with 5.

Let's try to figure out the next entry in the first row, which is, as we know, the leaf with smallest label of the tree after 5 was deleted. The node 1 is still ruled out, since it occurs later in the second row later; but 2 does not occur any more, which means (by the

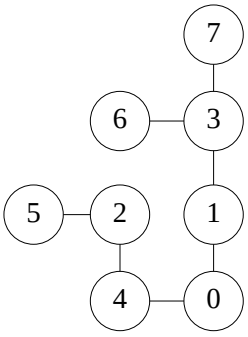

Figure 25: A tree reconstructed from its Prüfer code

same argument as before), that 2 was the leaf with smallest label after deleting 5. Thus the second entry in the first row is 2.

Similarly, the third entry must be 4, since all the smaller numbers either occur later or have been used already. Going on in a similar fashion, we get that the full array must have been:

$$\begin{array}{ccccccc} 5 & 2 & 4 & 6 & 7 & 3 & 1 \\ 2 & 4 & 0 & 3 & 3 & 1 & 0 \end{array}$$

This corresponds to the tree in Figure 25

The considerations above are completely general, and can be summed up as follows:

each entry in the first row of the extended Prüfer code is the smallest integer that does not occur in the first row before it, nor in the second row below or after it.

Indeed, when this entry (say, the k-th entry in the first row) was recorded, then the nodes before it in the first row were deleted (together with the edges corresponding to the first $k-1$ columns). The remaining entries in the second row are exactly those nodes that are fathers at this time, which means that they are not leaves.

This describes how the first row can be reconstructed from the second. So we don't need the full extended Prüfer code to store the tree; it suffices to store the second row. In fact, we know that the last entry in the second row is n, so we don't have to store this either. The sequence consisting of the first $n-2$ entries of the second row is called the *Prüfer code* of the tree. Thus the Prüfer code is a sequence of length $n-2$, each entry of which is a number between 0 and $n-1$.

This is similar to the father code, just one shorter; not much gain here for all the work. But the beauty of the Prüfer code is that it is optimal, in the sense that

every sequence of numbers between 0 *and* $n-1$*, of length* $n-2$*, is a Prüfer code of some tree on* n *nodes.*

This can be proved in two steps. First, we extend this sequence to a table with two rows: we add an n at the end, and then write above each entry in the first row the smallest integer that does not occur in the first row before it, nor in the second row below or after it (note that it is always possible to find such an integer: the condition excludes at most $n-1$ values out of n).

Now this table with two rows is the Prüfer code of a tree. The proof of this fact, which is not difficult any more, is left to the reader as an exercise.

10.11 Complete the proof.

Let us sum up what the Prüfer code gives. First, it proves Cayley's Theorem. Second, it provides a theoretically most efficient way of encoding trees. Each Prüfer code can be considered as a natural number written in the base n number system; in this way, we order a "serial number" between 0 and n^{n-2} to the n-point labeled trees. Expressing these serial numbers in base two, we get a code by $0-1$ sequences in of length at most $\lceil (n-2)\log n \rceil$.

As a third use of the Prüfer code, let's suppose that we want to write a program that generates a random labeled tree on n nodes in such a way that all trees occur with the same probability. This is not easy from scratch; but the Prüfer code gives an efficient solution. We just have to generate $n-2$ independent random integers between 0 and $n-1$ (most programming languages have a statement for this) and then "decode" this sequence as a tree, as described.

The number of unlabelled trees. The number of unlabelled trees on n nodes, usually denoted by T_n, is even more difficult to handle. No simple formula like Cayley's theorem is known for this number. Our goal is to get a rough idea about how large this number is.

There is only one unlabelled tree on $1, 2$ or 3 nodes; there are two on 4 nodes (the path and the star). There are 3 on 5 nodes (the star, the path, and the tree in Figure 23. These numbers are much smaller than the number of labelled trees with these numbers of nodes, which are $1, 1, 3, 16$ and 125 by Cayley's Theorem.

It is of course clear that the number of unlabelled trees is less than the number of labelled trees; every unlabelled tree can be labelled in many ways. How many ways? If we draw an unlabelled tree, we can label its nodes in $n!$ ways. The labelled trees we get this way are not necessarily all different: for example, if the tree is a "star", i.e., it $n-2$ leaves and one more node connected to all leaves, then no matter how we permute the labels of the leaves, we get the same labelled tree. So an unlabelled star yields n labelled stars.

But at least we know that each labeled tree can be labeled in at most $n!$ ways. Since the number of labelled trees is n^{n-2}, it follows that the number of unlabeled trees is at least $n^{n-2}/n!$. Using Stirling's formula (Theorem 2.5), we see that this number is about $e^n/n^{5/2}\sqrt{2\pi}$.

This number is musch smaller than the number of labelled trees, n^{n-2}, but of course it is only a *lower bound* on the number of unlabelled trees. How can we obtain an *upper bound* on this number? If we think in terms of storage, the issue is: can we store an unlabelled tree more economically than labelling its nodes and then storing it as a labelled tree? Very informally, how should we describe a tree, if we only want the "shape" of it, and don't care which node gets which label?

Take an n-point tree G, and specify one of its leaves as its "root". Next, draw G in the plane without crossing edges; this can always be done, and we almost always draw trees this way.

Now we imagine that the edges of the tree are walls, perpendicular to the plane. Starting at the root, walk around this system of walls, keeping the wall always to your right. We'll call walking along an edge a "step". Since there are $n-1$ edges, and each edge has two sides, we'll make $2(n-1)$ steps before returning to the root (Figure 26).

Each time we make a step *away* from the root (i.e., a step from a father to one of its sons), we write down a 1; each time we make a step *toward* the root, we write down a 0.

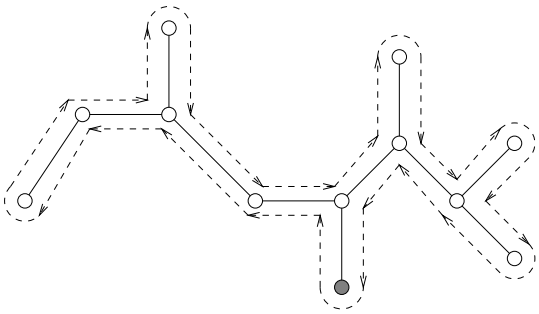

Figure 26: Walking around a tree

This way we end up with a sequence of length $2(n-1)$, consisting of 0's and 1's. We call this sequence the *planar code* of the (unlabelled) tree. The planar code of the tree in figure 26 is 1110011010100100.

Now this name already indicates that the planar code has the following important property:

Every unlabelled tree is uniquely determined by its planar code.

Let us illuminate the proof of this by assuming that the tree is covered by snow, and we only have its code. We ask a friend of ours to walk around the tree just like above, and uncover the walls, and we look at the code in the meanwhile. What do we see? Any time we see a 1, he walks along a wall away from the root, and he cleans it from the snow. We see this as growing a new twig. Any time we see a 0, he walks back, along an edge already uncovered, toward to root.

Now this describes a perfectly good way to draw the tree: we look at the bits of the code one by one, while keeping the pen on the paper. Any time we see a 1, we draw a new edge to a new node (and move the pen to the new node). Any time we see a 0, we move the pen back by one edge toward the root. Thus the tree is indeed determined by its planar code.

Since the number of possible planar codes is at most $2^{2(n-1)} = 4^{n-1}$, we get that the number of unlabelled trees is at most this large. Summing up:

Theorem 10.6 *The number T_n of unlabelled trees with n nodes satisfies*

$$\frac{n^{n-2}}{n!} < T_n < 4^{n-1}.$$

The exact form of this lower bound does not matter much; we can conclude, just to have a statement simpler to remember, that the number of unlabelled trees on n nodes is larger than 2^n if n is large enough ($n > 30$ if you work it out). So we get, at least for $n \geq 30$, the following bounds that are easier to remember:

$$2^n \leq T_n \leq 4^n.$$

The planar code is far from optimal; every unlabelled tree has many different codes (depending on how we draw it in the plane and how we choose the root), and not every 0-1

sequence of length $2(n-1)$ is a code of a tree (for example, it must start with a 1 and have the same number of 0's as 1's). Still, the planar code is quite an efficient way of encoding unlabelled trees: it uses less than $2n$ bits for trees with n nodes. Since there are more than 2^n unlabelled trees (at least for $n > 30$), we could not possibly get by with codes of length n: there are just not enough of them.

Unlike for labelled trees, we don't know a simple formula for the number of unlabelled trees on n nodes, and probably none exists. According to a difficult result of George Pólya, the number of unlabelled trees on n nodes is asymptotically $an^{3/2}b^n$, where a and b are real numbers defined in a complicated way.

10.12 Does there exist an unlabelled tree with planar code (a) 1111111100000000; (b) 1010101010101010; (c) 1100011100?