Prüfer code

Table of Contents

In this article we will look at the so-called **Prüfer code** (or Prüfer sequence), which is a way of encoding a labeled tree into a sequence of numbers in a unique way.

With the help of the Prüfer code we will prove **Cayley's formula** (which specified the number of spanning trees in a complete graph). Also we show the solution to the problem of counting the number of ways of adding edges to a graph to make it connected.

Note, we will not consider trees consisting of a single vertex - this is a special case in which multiple statements clash.

Prüfer code

The Prüfer code is a way of encoding a labeled tree with n vertices using a sequence of $n-2$ integers in the interval $[0; n-1]$. This encoding also acts as a

bijection between all spanning trees of a complete graph and the numerical sequences.

Although using the Prüfer code for storing and operating on tree is impractical due the specification of the representation, the Prüfer codes are used frequently: mostly in solving combinatorial problems.

The inventor - Heinz Prüfer - proposed this code in 1918 as a proof for Cayley's formula.

Building the Prüfer code for a given tree

The Prüfer code is constructed as follows. We will repeat the following procedure $n-2$ times: we select the leaf of the tree with the smallest number, remove it from the tree, and write down the number of the vertex that was connected to it. After $n-2$ iterations there will only remain 2 vertices, and the algorithm ends.

Thus the Prüfer code for a given tree is a sequence of $n-2$ numbers, where each number is the number of the connected vertex, i.e. this number is in the interval $[0, n-1]$.

The algorithm for computing the Prüfer code can be implemented easily with $O(n \log n)$ time complexity, simply by using a data structure to extract the minimum (for instance `set` or `priority_queue` in C++), which contains a list of all the current leafs.

```
vector<vector<int>> adj;

vector<int> pruefer_code() {
    int n = adj.size();
    set<int> leafs;
    vector<int> degree(n);
    vector<bool> killed(n, false);
    for (int i = 0; i < n; i++) {
```

```
            degree[i] = adj[i].size();
            if (degree[i] == 1)
                leafs.insert(i);
        }

        vector<int> code(n - 2);
        for (int i = 0; i < n - 2; i++) {
            int leaf = *leafs.begin();
            leafs.erase(leafs.begin());
            killed[leaf] = true;

            int v;
            for (int u : adj[leaf]) {
                if (!killed[u])
                    v = u;
            }

            code[i] = v;
            if (--degree[v] == 1)
                leafs.insert(v);
        }

        return code;
    }
```

However the construction can also be implemented in linear time. Such an approach is described in the next section.

Building the Prüfer code for a given tree in linear time

The essence of the algorithm is to use a **moving pointer**, which will always point to the current leaf vertex that we want to remove.

At first glance this seems impossible, because during the process of constructing the Prüfer code the leaf number can increase and decrease. However after a closer look,

this is actually not true. The number of leafs will not increase. Either the number decreases by one (we remove one leaf vertex and don't gain a new one), or it stay the same (we remove one leaf vertex and gain another one). In the first case there is no other way than searching for the next smallest leaf vertex. In the second case, however, we can decide in $O(1)$ time, if we can continue using the vertex that became a new leaf vertex, or if we have to search for the next smallest leaf vertex. And in quite a lot of times we can continue with the new leaf vertex.

To do this we will use a variable ptr, which will indicate that in the set of vertices between 0 and ptr is at most one leaf vertex, namely the current one. All other vertices in that range are either already removed from the tree, or have still more than one adjacent vertices. At the same time we say, that we haven't removed any leaf vertices bigger than ptr yet.

This variable is already very helpful in the first case. After removing the current leaf node, we know that there cannot be a leaf node between 0 and ptr, therefore we can start the search for the next one directly at $\mathrm{ptr}+1$, and we don't have to start the search back at vertex 0. And in the second case, we can further distinguish two cases: Either the newly gained leaf vertex is smaller than ptr, then this must be the next leaf vertex, since we know that there are no other vertices smaller than ptr. Or the newly gained leaf vertex is bigger. But then we also know that it has to be bigger than ptr, and can start the search again at $\mathrm{ptr}+1$.

Even though we might have to perform multiple linear searches for the next leaf vertex, the pointer ptr only increases and therefore the time complexity in total is $O(n)$.

```cpp
vector<vector<int>> adj;
vector<int> parent;

void dfs(int v) {
    for (int u : adj[v]) {
        if (u != parent[v]) {
            parent[u] = v;
            dfs(u);
        }
    }
}

vector<int> pruefer_code() {
    int n = adj.size();
    parent.resize(n);
    parent[n-1] = -1;
    dfs(n-1);

    int ptr = -1;
    vector<int> degree(n);
    for (int i = 0; i < n; i++) {
        degree[i] = adj[i].size();
        if (degree[i] == 1 && ptr == -1)
            ptr = i;
    }

    vector<int> code(n - 2);
    int leaf = ptr;
    for (int i = 0; i < n - 2; i++) {
        int next = parent[leaf];
        code[i] = next;
        if (--degree[next] == 1 && next < ptr)
            leaf = next;
        } else {
            ptr++;
            while (degree[ptr] != 1)
                ptr++;
            leaf = ptr;
        }
```

```
    }

    return code;
}
```

In the code we first find for each its ancestor `parent[i]`, i.e. the ancestor that this vertex will have once we remove it from the tree. We can find this ancestor by rooting the tree at the vertex $n-1$. This is possible because the vertex $n-1$ will never be removed from the tree. We also compute the degree for each vertex. `ptr` is the pointer that indicates the minimum size of the remaining leaf vertices (except the current one `leaf`). We will either assign the current leaf vertex with `next`, if this one is also a leaf vertex and it is smaller than `ptr`, or we start a linear search for the smallest leaf vertex by increasing the pointer.

It can be easily seen, that this code has the complexity $O(n)$.

Some properties of the Prüfer code

- After constructing the Prüfer code two vertices will remain. One of them is the highest vertex $n-1$, but nothing else can be said about the other one.
- Each vertex appears in the Prüfer code exactly a fixed number of times - its degree minus one. This can be easily checked, since the degree will get smaller every time we record its label in the code, and we remove it once the degree is 1. For the two remaining vertices this fact is also true.

Restoring the tree using the Prüfer code

To restore the tree it suffice to only focus on the property discussed in the last section. We already know the degree of all the vertices in the desired tree. Therefore

we can find all leaf vertices, and also the first leaf that was removed in the first step (it has to be the smallest leaf). This leaf vertex was connected to the vertex corresponding to the number in the first cell of the Prüfer code.

Thus we found the first edge removed by when then the Prüfer code was generated. We can add this edge to the answer and reduce the degrees at both ends of the edge.

We will repeat this operation until we have used all numbers of the Prüfer code: we look for the minimum vertex with degree equal to 1, connect it with the next vertex from the Prüfer code, and reduce the degree.

In the end we only have two vertices left with degree equal to 1. These are the vertices that didn't got removed by the Prüfer code process. We connect them to get the last edge of the tree. One of them will always be the vertex $n - 1$.

This algorithm can be **implemented** easily in $O(n \log n)$: we use a data structure that supports extracting the minimum (for example `set<>` or `priority_queue<>` in C++) to store all the leaf vertices.

The following implementation returns the list of edges corresponding to the tree.

```
vector<pair<int, int>> pruefer_decode(vector<i
    int n = code.size() + 2;
    vector<int> degree(n, 1);
    for (int i : code)
        degree[i]++;

    set<int> leaves;
    for (int i = 0; i < n; i++) {
        if (degree[i] == 1)
```

```
            leaves.insert(i);
    }

    vector<pair<int, int>> edges;
    for (int v : code) {
        int leaf = *leaves.begin();
        leaves.erase(leaves.begin());

        edges.emplace_back(leaf, v);
        if (--degree[v] == 1)
            leaves.insert(v);
    }
    edges.emplace_back(*leaves.begin(), n-1);
    return edges;
}
```

Restoring the tree using the Prüfer code in linear time

To obtain the tree in linear time we can apply the same technique used to obtain the Prüfer code in linear time.

We don't need a data structure to extract the minimum. Instead we can notice that, after processing the current edge, only one vertex becomes a leaf. Therefore we can either continue with this vertex, or we find a smaller one with a linear search by moving a pointer.

```
vector<pair<int, int>> pruefer_decode(vector<i
    int n = code.size() + 2;
    vector<int> degree(n, 1);
    for (int i : code)
        degree[i]++;

    int ptr = 0;
    while (degree[ptr] != 1)
        ptr++;
    int leaf = ptr;
```

```
    vector<pair<int, int>> edges;
    for (int v : code) {
        edges.emplace_back(leaf, v);
        if (--degree[v] == 1 && v < ptr) {
            leaf = v;
        } else {
            ptr++;
            while (degree[ptr] != 1)
                ptr++;
            leaf = ptr;
        }
    }
    edges.emplace_back(leaf, n-1);
    return edges;
}
```

Bijection between trees and Prüfer codes

For each tree there exists a Prüfer code corresponding to it. And for each Prüfer code we can restore the original tree.

It follows that also every Prüfer code (i.e. a sequence of $n-2$ numbers in the range $[0; n-1]$) corresponds to a tree.

Therefore all trees and all Prüfer codes form a bijection (a **one-to-one correspondence**).

Cayley's formula

Cayley's formula states that the **number of spanning trees in a complete labeled graph** with n vertices is equal to:

$$n^{n-2}$$

There are multiple proofs for this formula. Using the Prüfer code concept this statement comes without any surprise.

In fact any Prüfer code with $n-2$ numbers from the interval $[0; n-1]$ corresponds to some tree with n vertices. So we have n^{n-2} different such Prüfer codes. Since each such tree is a spanning tree of a complete graph with n vertices, the number of such spanning trees is also n^{n-2}.

Number of ways to make a graph connected

The concept of Prüfer codes are even more powerful. It allows to create a lot more general formulas than Cayley's formula.

In this problem we are given a graph with n vertices and m edges. The graph currently has k components. We want to compute the number of ways of adding $k-1$ edges so that the graph becomes connected (obviously $k-1$ is the minimum number necessary to make the graph connected).

Let us derive a formula for solving this problem.

We use $s_1, \dots, s_k$ for the sizes of the connected components in the graph. We cannot add edges within a connected component. Therefore it turns out that this problem is very similar to the search for the number of spanning trees of a complete graph with k vertices. The only difference is that each vertex has actually the size s_i: each edge connecting the vertex i, actually multiplies the answer by s_i.

Thus in order to calculate the number of possible ways it is important to count how often each of the k vertices is

used in the connecting tree. To obtain a formula for the problem it is necessary to sum the answer over all possible degrees.

Let $d_1, \ldots, d_k$ be the degrees of the vertices in the tree after connecting the vertices. The sum of the degrees is twice the number of edges:

$$\sum_{i=1}^{k} d_i = 2k - 2$$

If the vertex i has degree d_i, then it appears $d_i - 1$ times in the Prüfer code. The Prüfer code for a tree with k vertices has length $k - 2$. So the number of ways to choose a code with $k - 2$ numbers where the number i appears exactly $d_i - 1$ times is equal to the **multinomial coefficient**

$$\binom{k-2}{d_1-1, d_2-1, \ldots, d_k-1} = \frac{(k-2)!}{(d_1-1)!(d_2-1)!\cdots(d_k}$$

The fact that each edge adjacent to the vertex i multiplies the answer by s_i we receive the answer, assuming that the degrees of the vertices are $d_1, \ldots, d_k$:

$$s_1^{d_1} \cdot s_2^{d_2} \cdots s_k^{d_k} \cdot \binom{k-2}{d_1-1, d_2-1, \ldots, d_k-1}$$

To get the final answer we need to sum this for all possible ways to choose the degrees:

$$\sum_{\substack{d_i \geq 1 \\ \sum_{i=1}^{k} d_i = 2k-2}} s_1^{d_1} \cdot s_2^{d_2} \cdots s_k^{d_k} \cdot \binom{k-2}{d_1-1, d_2-1, \ldots, d_k-1}$$

Currently this looks like a really horrible answer, however we can use the **multinomial theorem**, which says:

$$(x_1 + \cdots + x_m)^p = \sum_{\substack{c_i \geq 0 \\ \sum_{i=1}^{m} c_i = p}} x_1^{c_1} \cdot x_2^{c_2} \cdots x_m^{c_m} \cdot \binom{p}{c_1, c_2, \ldots c}$$

This look already pretty similar. To use it we only need to substitute with $e_i = d_i - 1$:

$$\sum_{\substack{e_i \geq 0 \\ \sum_{i=1}^{k} e_i = k-2}} s_1^{e_1+1} \cdot s_2^{e_2+1} \cdots s_k^{e_k+1} \cdot \binom{k-2}{e_1, e_2, \ldots, e_k}$$

After applying the multinomial theorem we get the **answer to the problem**:

$$s_1 \cdot s_2 \cdots s_k \cdot (s_1 + s_2 + \cdots + s_k)^{k-2} = s_1 \cdot s_2 \cdots s_k \cdot n^{k-2}$$

By accident this formula also holds for $k = 1$.

Practice problems

- UVA #10843 - Anne's game
- Timus #1069 - Prufer Code
- Codeforces - Clues
- Topcoder - TheCitiesAndRoadsDivTwo