added: 26 Mar 2012 1:00
Edited: 26 Mar 2012 1:00

Prüfer code. Cayley's formula. Number of ways to make the graph connected

In this article we consider the so-called **Prüfer code** , which is a way of unambiguous labeled tree by a sequence of numbers.

Use the code Priifer demonstrated proof **of Cayley's formula** (specifying the number of spanning trees in a complete graph), as well as solution to the problem of the number of ways to add to the given graph edges to turn it into a coherent.

Note . We will not consider trees consisting of a single vertex - is a special case, in which many statements degenerate.

Contents [hide]

Prüfer code

Prüfer code - a way of mutually unambiguous labeled trees with $n$vertices by a sequence $n - 2$of integers in the interval $[1; n]$. In other words, the code Priifer - is **a bijection** between all spanning trees of a complete graph and numerical sequences.

Although the use of Prüfer code for storing and manipulating trees is impractical due to the specificity of the submission, Prüfer codes are used in solving combinatorial problems.

Author - Heinz Prüfer (Heinz Prüfer) - suggested this code in 1918 as proof of Cayley's formula (see. Below).

Building code Priifer for this tree

Prüfer code is constructed as follows. Will be $n - 2$times prodelyvat procedure: choose a tree leaf with the lowest number, remove it from the tree, and add the code to the top of the Prüfer number that was associated with this list. Eventually, the tree will be only $2$the top, and this completes the algorithm (the number of these vertices explicitly in the code are not recorded).

Thus the code tree for a given Prüfer - a sequence of $n - 2$numbers, where each number - number of vertices associated with the lowest at the time the sheet - i.e. this number in the interval $[1; n]$.

Algorithm for computing Prüfer code is easy to implement with the asymptotic behavior $O(n \log n)$, simply maintaining a data structure for extracting a minimum (for example, set <>or priority_queue <>in the language C ++), containing a list of all current leaves:

```
const int MAXN = ...;
int n;
vector<int> g[MAXN];
int degree[MAXN];
bool killed[MAXN];

vector<int> prufer_code() {
        set<int> leaves;
        for (int i=0; i<n; ++i) {
                degree[i] = (int) g[i].size();
                if (degree[i] == 1)
                        leaves.insert (i);
                killed[i] = false;
        }

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

                int v;
                for (size_t i=0; i<g[leaf].size(); ++i)
                        if (!killed[g[leaf][i]])
```

```
                                v = g[leaf][i];

                result[iter] = v;
                if (--degree[v] == 1)
                        leaves.insert (v);
        }
        return result;
}
```

However, the construction of Prüfer code can be realized and in linear time, which is described in the next section.

Building code Priifer for this tree in linear time

We give here a simple algorithm, which has the asymptotic behavior $O(n)$.

The essence of the algorithm is to keep **moving the pointer** ptr , which will always be to move only in the direction of increasing numbers of vertices.

At first glance, this is not possible, because in the process of building code Prüfer numbers of leaves can both increase and **decrease** . But it is easy to note that the reduction occurs only in one case: the code by removing the current sheet its parent has a lower number (this will be the ancestor of the minimum list and removed from the tree on the very next step Prüfer code). Thus, the reduction of cases can be treated in time $O(1)$, and does not interfere with the construction of the algorithm **linear asymptotic behavior** :

```
const int MAXN = ...;
int n;
vector<int> g[MAXN];
int parent[MAXN], degree[MAXN];

void dfs (int v) {
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (to != parent[v]) {
                        parent[to] = v;
                        dfs (to);
                }
        }
}

vector<int> prufer_code() {
        parent[n-1] = -1;
        dfs (n-1);

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

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

We comment on this code. The main function here - $\mathrm{prufer_code}()$which returns Priifer wood specified in global variables n(number of vertices) and g(adjacency lists specifying the graph). Initially, we find for each vertex of its ancestor $\mathrm{parent}[i]$- ie of the ancestor, which this vertex will have at the time of removal of the tree (all that we can find in advance, using the fact that the maximum vertex $n-1$never removed from the tree). Also, we find for each vertex its degree $\mathrm{degree}[i]$. Variable ptr- is moving the pointer ("candidate" for a minimum list) that changes always in the direction of

increasing. Variable leaf- is the current sheet with a minimum number. Thus, each iteration of code Priifer is to add leaf to the answer, as well as making sure there was not $\text{parent}[\text{leaf}]$less than the current candidate ptr: if it turned out less, we simply assign $\text{leaf} = \text{parent}[\text{leaf}]$, and otherwise - move the pointer $\text{ptr}$to the next sheet.

As can be seen easily by the code, the asymptotic behavior of the algorithm is really $O(n)$a pointer to ptrundergo a $O(n)$change, and all the rest of the algorithm is obviously work in linear time.

Some properties of codes Priifer

- Upon completion of the construction of Prüfer code in the tree will remain undeleted two vertices.

  One of them will surely be the vertex with the maximum number - $n - 1$and that's about another vertex nothing definite can be said.

- Each vertex is found in the code Priifer certain number of times equal to its power minus one.

  This is easily understood if we note that the vertex is removed from the tree at a time when its power is equal to one - ie by this time all the adjacent edges, except one, were removed. (For the two remaining vertices after the construction of the code it is also true.)

Recovering a tree by its code Priifer

To restore the tree is sufficient to note in the previous paragraph, that the degrees of all vertices in the target tree, we already know (and can calculate and store in a array $degree[]$). Therefore, we can find all the leaves, and, accordingly, the smallest number plate - which was removed in the first step. This sheet was connected to the top, the number of which is recorded in the first cell of the Prüfer code.

Thus, we find the first edge, remote Priifer code. Add this edge back, then reduce power $degree[]$at both ends of the ribs.

We repeat this step until you have reviewed all the code Priifer: search with minimal vertex $degree = 1$, connect it to another vertex Prüfer code, decrease $degree[]$at both ends.

In the end we are left with only two vertices with $degree = 1$- these are the vertices that the algorithm Priifer left undeleted. Connect them by an edge.

The algorithm is complete, the desired tree is built.

Implement this algorithm easily during $O(n \log n)$: maintaining the structure of data to extract a minimum (for example, $\text{set}<>$or $\text{priority_queue}<>$in C ++) numbers of all vertices with $degree = 1$, and removing from it every time at least.

We give an implementation (where the function $prufer_decode()$returns a list of edges required tree):

```
vector < pair<int,int> > prufer_decode (const vector<int> & prufer_code) {
        int n = (int) prufer_code.size() + 2;
        vector<int> degree (n, 1);
        for (int i=0; i<n-2; ++i)
                ++degree[prufer_code[i]];

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

        vector < pair<int,int> > result;
        for (int i=0; i<n-2; ++i) {
                int leaf = *leaves.begin();
                leaves.erase (leaves.begin());

                int v = prufer_code[i];
                result.push_back (make_pair (leaf, v));
                if (--degree[v] == 1)
                        leaves.insert (v);
        }
        result.push_back (make_pair (*leaves.begin(), *--leaves.end()));
        return result;
}
```

Recovering tree code Priifer in linear time

To obtain an algorithm with linear asymptotic behavior can apply the same technique that was used to obtain a linear algorithm for computing Prüfer code.

In fact, in order to find the lowest-numbered sheet optionally start a data structure for retrieving a minimum. Instead, you

will notice that after we find and treat the current sheet, it adds to the consideration of only one new vertex. Consequently, we can do a moving pointer with a variable that holds a minimum current list:

```
vector < pair<int,int> > prufer_decode_linear (const vector<int> & prufer_code) {
        int n = (int) prufer_code.size() + 2;
        vector<int> degree (n, 1);
        for (int i=0; i<n-2; ++i)
                ++degree[prufer_code[i]];

        int ptr = 0;
        while (ptr < n && degree[ptr] != 1)
                ++ptr;
        int leaf = ptr;

        vector < pair<int,int> > result;
        for (int i=0; i<n-2; ++i) {
                int v = prufer_code[i];
                result.push_back (make_pair (leaf, v));

                --degree[leaf];
                if (--degree[v] == 1 && v < ptr)
                        leaf = v;
                else {
                        ++ptr;
                        while (ptr < n && degree[ptr] != 1)
                                ++ptr;
                        leaf = ptr;
                }
        }
        for (int v=0; v<n-1; ++v)
                if (degree[v] == 1)
                        result.push_back (make_pair (v, n-1));
        return result;
}
```

One to one correspondence between trees and Prüfer codes

On the one hand, for every tree there is exactly one code Prüfer, the corresponding (this follows from the definition of Prüfer code).

On the other hand, the correctness of the algorithm for reconstructing the tree code Priifer implies that any code Prüfer (ie, the sequence of $n-2$numbers where each number lies in the interval $[1;n]$) corresponds to a tree.

Thus, all the trees and all the codes form a Prüfer **bijection** .

Cayley's formula

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

$$n^{n-2}.$$

There are many **proofs** of this formula, but the proof using the code Priifer clearly and constructively.

In fact, any set of $n-2$numbers from the interval $[1;n]$corresponds uniquely to a tree of $n$nodes. Total of different codes Prüfer n^{n-2}. As in the case of a complete graph of the $n$vertices as the core fits any tree, then the number of spanning trees of the same n^{n-2}, as required.

Number of ways to make the graph connected

Power Prüfer codes is that they allow you to get a more general formula than formula Cayley.

So, given a graph of $n$vertices and $m$edges; let k- the number of connected components in this graph. Required to find the number of ways to add $k-1$an edge to the graph was connected (obviously $k-1$rib - the minimum number of edges to make the graph connected).

We derive a ready formula for the solution of this problem.

We denote $s_1, \ldots, s_k$the size of the connected components of the graph. Since adding edges in the connected components of smoking, it turns out that the problem is very similar to the search for the number of spanning trees in a complete graph of $k$vertices: but the difference here is that each vertex has its own "weight" s_i: each edge is adjacent to i th top, multiplies response s_i.

Thus, to count the number of ways is important, what power have all $k$vertices in the skeleton. To obtain equations for the problem it is necessary to sum the answers to all possible degrees.

Let $d_1, \dots, d_k$- degrees of the vertices in the core. Sum of the degrees of vertices is equal to twice the number of edges, so:

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

If ith vertex has degree d_i, the Prüfer code it enters $d_i - 1$again. Prüfer code for the tree of the $k$vertices has a length $k - 2$. Number of ways to choose a set of $k - 2$numbers, where the number $i$occurs exactly $d_i - 1$once, as well **multinomial coefficient** (similar to the binomial coefficient)

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

Given the fact that each edge adjacent to ith top, multiply the answer to s_i, we get the answer, provided that the degrees of the vertices are $d_1, \dots, d_k$equal:

$$s_1^{d_1} \cdot s_2^{d_2} \cdot \ldots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)!\,(d_2-1)!\ \dots\ (d_k-1)!}.$$

To get the answer to the problem, we must add this formula over all admissible sets $\{d_i\}_{i=1}^{i=k}$:

$$\sum_{\substack{d_i \geq 1, \\ \sum_{i=1}^{k} d_i = 2k-2}} s_1^{d_1} \cdot s_2^{d_2} \cdot \ldots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)!\,(d_2-1)!\ \dots\ (d_k-1)!}.$$

For coagulation of this formula we use the definition of a multinomial coefficient:

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

Comparing this formula with the previous one, we see that if we introduce the notation $e_i = d_i - 1$:

$$\sum_{\substack{e_i \geq 0, \\ \sum_{i=1}^{k} e_i = k-2}} s_1^{e_1+1} \cdot s_2^{e_2+1} \cdot \ldots \cdot s_k^{e_k+1} \cdot \frac{(k-2)!}{e_1!\ e_2!\ \dots\ e_k!},$$

after folding **answer to the problem** is:

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

(This formula is valid and $k = 1$, although formal proof of it should not have.)

Problem in online judges

Problem in online judges, which use codes Priifer:

- UVA # 10843 **"Anne's game"** [Difficulty: Easy]
- TIMUS # 1069 **"Code Priifer"** [Difficulty: Easy]
- Codeforces 110D **"Clues"** [Difficulty: Medium]
- TopCoder SRM 460 **"TheCitiesAndRoadsDivTwo"** [Difficulty: Medium]