USACO FEB10 Problem 'slowdown' Analysis

by Jelle van den Hooff

From the limits of the task it appears that simply simulating the process is not good enough. It is also difficult to figure out how each cow has to walk.

Instead of that, write at each pasture in the graph the 'arrival time' of its cow. Then the problem is to count the number of smaller arrival times on the path between the barn and the pasture.

This can be done doing a depth-first search over the tree, and storing the arrival times we've already seen. A naive approach is to store the arrival times in a list, and then checking each item. This is easy to program, but also leads to quadratic runtime.

Instead of storing a list of arrival times, we store the number of times each arrival time has been seen, and then sum over all smaller arrival times. If we have seen arrival times 1, 3 and 4 that would be stored like this:

```
time 0 1 2 3 4 5
seen 0 1 0 1 1 0
```

If we now visit a pasture with arrival time 5 the answer is 0+1+0+1+1.

Why this complicated approach? Because these operations can be done very efficiently with 'segment trees'. In general, these data structures allow for efficient updating and querying over very large ranges. An explanation can be found at [1] .

A solution by Mark Gordon implements this idea:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <cstdio>

using namespace std;

#define MAXN (1 << 17)

int tree[MAXN * 2];

void add_range(int x, int xa, int xb, int a, int b, int v) {
  if(b <= xa) return;
  if(xb <= a) return;
  if(a <= xa && xb <= b) {
    tree[x] += v;
  } else {
    int x1 = 2 * x + 1;
    int x2 = x1 + 1;
    int md = xa + (xb - xa) / 2;
    tree[x1] += tree[x];
    tree[x2] += tree[x];
    tree[x] = 0;
    add_range(x1, xa, md, a, b, v);
    add_range(x2, md, xb, a, b, v);
  }
}
```

```
int get_range(int x, int xa, int xb, int a, int b) {
  if(b <= xa) return 0;
  if(xb <= a) return 0;
  if(a <= xa && xb <= b) {
    return (xb - xa) * tree[x];
  } else {
    int x1 = 2 * x + 1;
    int x2 = x1 + 1;
    int md = xa + (xb - xa) / 2;
    tree[x1] += tree[x];
    tree[x2] += tree[x];
    tree[x] = 0;
    return get_range(x1, xa, md, a, b) + get_range(x2, md, xb, a, b);
  }
}

int N;
vector<int> edge[100000];
int A[100000];
int B[100000];

void label(int x, int p, int& nxt) {
  A[x] = nxt++;
  for(int i = 0; i < edge[x].size(); i++) {
    if(edge[x][i] != p) {
      label(edge[x][i], x, nxt);
    }
  }
  B[x] = nxt;
}

int main() {
  freopen("slowdown.in", "r", stdin);
  freopen("slowdown.out", "w", stdout);
  scanf("%d", &N);
  for(int i = 1; i < N; i++) {
    int u, v;
    scanf("%d%d", &u, &v); u--; v--;
    edge[u].push_back(v);
    edge[v].push_back(u);
  }
  int nxt = 0;
  label(0, -1, nxt);
  for(int i = 0; i < N; i++) {
    int x;
    scanf("%d", &x); x--;
    printf("%d\n", get_range(0, 0, MAXN, A[x], A[x] + 1));
    add_range(0, 0, MAXN, A[x], B[x], 1);
  }
}
```

Another solution is based on 'Binary Indexed Trees'. These are explained at [2] with lots of text and pictures, but the actual implementation is very simple (so worth remembering).

For every pairs 1 <= a, b <= n the following loops take log(n) time:

```
for (int x = a; x < n; x += (x & -x)) printf("a: %d", x);
for (int x = b; x > 0; x += (x & -x)) printf("b: %d", x);
```

If a ≤ b, exactly one number is printed two times. Otherwise none at all. So if we increase seen[x] by 1 in the first loop, and sum over the seen[x] in the second loop we can efficiently calculate the number of smaller arrival times. My C++ solution:

```
#include <cstdio>
#include <cstring>
#include <vector>
using namespace std;

FILE *fin = freopen("slowdown.in", "r", stdin);
FILE *fout = freopen("slowdown.out", "w", stdout);

const int MAXN = 100000;
int n, owner[MAXN], ans[MAXN], seen[MAXN + 1];
vector<int> edges[MAXN];

void go(int x, int p) {
  ans[owner[x]] = 0;
  for (int i = owner[x]; i > 0; i -= (i&-i)) ans[owner[x]] += seen[i];

  for (int i = owner[x] + 1; i <= n; i += (i&-i)) seen[i] += 1;
  for (vector<int>::iterator i = edges[x].begin(); i != edges[x].end(); i++)
       if (*i != p) go(*i, x);
  for (int i = owner[x] + 1; i <= n; i += (i&-i)) seen[i] -= 1;
}

int main() {
  scanf("%d", &n);

  for (int i = 0; i < n - 1; i++) {
    int a, b;
    scanf("%d%d", &a, &b);
    a--; b--;
    edges[a].push_back(b);
    edges[b].push_back(a);
  }

  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    x--;
    owner[x] = i;
  }

  memset(seen, 0, sizeof(seen));
  go(0, -1);

  for (int i = 0; i < n; i++) printf("%d\n", ans[i]);
}
```

[1]: http://pegjudge.ath.cx:5050/wiki/index.php/Segment_tree
[2]: http://www.topcoder.com/tc? module=Static&d1=tutorials&d2=binaryIndexedTrees