USACO DEC12 Problem 'runaway' Analysis

by Travis Hance and Richard Peng

The problem asks us to find, given a rooted tree T, for each node v, the number of descendants within a certain distance L. If we assign to each node v its depth (that is, the distance of that node from the root), then for a node v of depth D, we just want to count the number of descendants of depth at most D+L.

The simplest and most concise way of doing this problem is to 'flip over' what we need to compute. For each vertex, we find how far up the tree is within a distance of L from it. This can be done either by doing a DFS traversal of the tree and binary searching on the search stack, or by building 'jump' pointers of distance 2^i upwards in the tree.

Once this information is found, all we need to do is convert it back into information about descendants. This could be done by marking the first position up the tree that's at a distance of more than L from it. The answer at a vertex can then be computed by summing over the answers for its descendants, and subtracting away the number of times the current vertex is marked.

A very concise and well written solution by Roman Rubanenko is below. His code computed tables that move up the tree by a distance of 2^i, and made clever use of the fact that the parent of node i is at a smaller index

```
Var ans,d:array[0..200333]of int64;
    i,n,j,v:longint;
    len:int64;
    p:array[0..200333,0..19]of longint;
  begin
    assign(input,'runaway.in');reset(input);
    assign(output,'runaway.out');rewrite(output);
    read(n,len);
    ans[1]:=1;
    for i:=2 to n do
      begin
        read(p[i,0]);
        read(d[i]);
        d[i]:=d[i]+d[p[i,0]];
        for j:=1 to 18 do
          p[i,j]:=p[p[i,j-1],j-1];
        v:=i;
        for j:=18 downto 0 do
          if d[i]-d[p[v,j]]<=len then v:=p[v,j];
        inc(ans[i]);
        dec(ans[p[v,0]]);
      end;
    for i:=n downto 1 do
      ans[p[i,0]]:=ans[p[i,0]]+ans[i];
    for i:=1 to n do
      Writeln(ans[i]);
  end.
```

A more complicated solution would be to count the number of descendants of each node directly. For a node v, we will say the set of such descendants is S(v).

To find the size of S(v) for each v, we try at first to simply find the entire set S(v). We can do this recursively: to find S(v), first take the union of S(v') for all children v' of v, and then insert v itself. This set can only be too large, since if a descendant of v is within distance L of v, then it is within distance L of one of v's children. Then, we just need to remove nodes from this set which are too far away.

To make this solution feasible, we need an efficient data structure to store the sets S(v). This data structure needs to be able to support the removal of objects of high distance, so the natural choice would be a priority queue, where we insert nodes keyed by depth. Then we need to worry about taking the union of sets. Many efficient priority queues do support a union operation (commonly called "meld" or "merge"), such as Fibonacci heaps, Binomial heaps, and Pairing heaps. However, most programming language implementations of priority queues (e.g., the C++ std::priority_queue, or Java's PriorityQueue) do not support this operation.

We could implement a priority queue from scratch, but the following trick is probably easier: we can two priority queues of sizes m and n simply by popping, one-by-one, elements of the first priority queue, and inserting them into the second. This takes O(m log n) time. Thus if we are careful to always merge the smaller heap into the larger one (i.e., m < n) rather than the other way around, one can show that, summed over all nodes of the tree, this will only take O(N log^2 N) time.

An alternate approach to the problem is to start by making an in-order traversal of the tree and writing out the nodes visited, in order. Then, any subtree of the tree corresponds to a contiguous range in this list. Then, we need to answer queries of the form "how many elements in a given range have value at most c?" We can answer these queries with any range-query data structure (e.g., a Fenwick tree or a range tree) in increasing order of c, by marking elements in the list as they qualify.

Here is a solution by Travis Hance, using the first approach:

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

#define NMAX 200005

struct entry {
    long long weight;
    long long depth;
    bool operator<(entry const& o) const {
        return depth < o.depth;
    }
};

struct node {
    vector<node*> children;
    int sizeInNodes;
    long long answer;
    priority_queue<entry>* q;
    long long weight;
    long long qweight;
    long long dist;
};
node nodes[NMAX];

long long l;

void dfs(node* v, long long depth) {
    v->sizeInNodes = 1;
    node* largestChild = NULL;
    for (int i = 0; i < v->children.size(); i++) {
        node* c = v->children[i];
        dfs(c, depth + c->dist);
        v->sizeInNodes += c->sizeInNodes;
        if (largestChild == NULL
               || c->sizeInNodes > largestChild->sizeInNodes) {
            largestChild = c;
        }
    }

    if (largestChild == NULL) {
        v->q = new priority_queue<entry>();
        v->qweight = 0;
    } else {
        v->q = largestChild->q;
        v->qweight = largestChild->qweight;
        while (v->q->size() > 0 && v->q->top().depth > depth + l) {
            v->qweight -= v->q->top().weight;
            v->q->pop();
        }
    }

    for (int i = 0; i < v->children.size(); i++) {
        node* c = v->children[i];
        if (c != largestChild) {
            while (c->q->size() > 0) {
                entry e = c->q->top();
                c->q->pop();
                if (e.depth <= depth + l) {
                    v->q->push(e);
                    v->qweight += e.weight;
                }
            }
        }
    }

    entry e;
    e.weight = v->weight;
    e.depth = depth;
    v->q->push(e);
    v->qweight += e.weight;

    v->answer = v->qweight;
}
```

```
int main() {
    freopen("runaway.in","r",stdin);
    freopen("runaway.out","w",stdout);

    int n;
    scanf("%d", &n);
    scanf("%lld", &l);
    nodes[0].weight = 1;
    nodes[0].dist = 0;
    for (int i = 1; i < n; i++) {
        int parent;
        scanf("%d", &parent);
        parent--;
        nodes[parent].children.push_back(&nodes[i]);
        nodes[i].weight = 1;
        scanf("%lld", &nodes[i].dist);
    }

    dfs(&nodes[0], 0);

    for (int i = 0; i < n; i++) {
        printf("%lld\n", nodes[i].answer);
    }
}
```

Here is a solution by Bruce Merry using the second approach:

```
#include <fstream>
#include <algorithm>
#include <vector>

using namespace std;

typedef long long ll;

struct node
{
    node *parent;
    vector<node *> children;
    ll depth;
    int last;
    int label;

    node() : parent(NULL), depth(0), last(-1) {}

    void make_labels(int &pool)
    {
        label = pool++;
        for (size_t i = 0; i < children.size(); i++)
            children[i]->make_labels(pool);
        if (children.empty())
            last = label;
        else
            last = children.back()->last;
    }
};
```

```
struct event
{
    int A, B;
    ll l;
    int idx;

    bool operator <(const event &b) const
    {
        if (l != b.l)
            return l < b.l;
        else
            return A < b.A;
    }
};

static void bit_add(vector<int> &bit, int p, int v)
{
    p++;
    while (p < int(bit.size()))
    {
        bit[p] += v;
        p += p & ~(p - 1);
    }
}

static int bit_get(const vector<int> &bit, int p)
{
    int ans = 0;
    p++;
    while (p > 0)
    {
        ans += bit[p];
        p &= p - 1;
    }
    return ans;
}

int main()
{
    ifstream in("runaway.in");
    ofstream out("runaway.out");

    int N;
    ll L;
    in >> N >> L;
    node *nodes = new node[N];

    nodes[0].last = 0;
    for (int i = 1; i < N; i++)
    {
        int p;
        ll l;
        in >> p >> l;
        p--;
        node *pr = nodes + p;
        nodes[i].parent = pr;
        nodes[i].depth = pr->depth + l;
        pr->children.push_back(nodes + i);
    }
    int label_pool = 0;
    nodes[0].make_labels(label_pool);

    vector<event> events;
    events.reserve(2 * N);
    for (int i = 0; i < N; i++)
    {
        event add;
        add.A = -1;
        add.B = -1;
        add.l = nodes[i].depth;
        add.idx = nodes[i].label;
        events.push_back(add);

        event q;
        q.A = nodes[i].label;
        q.B = nodes[i].last + 1;
        q.l = nodes[i].depth + L;
        q.idx = i;
        events.push_back(q);
    }

    vector<int> ans(N);
    vector<int> bit(N + 1);
    sort(events.begin(), events.end());
    for (int i = 0; i < int(events.size()); i++)
    {
        const event &e = events[i];
        if (e.A == -1)
        {
            // add
            bit_add(bit, e.idx, 1);
        }
        else
        {
            // query
            ans[e.idx] = bit_get(bit, e.B - 1) - bit_get(bit, e.A - 1);
        }
    }

    for (int i = 0; i < N; i++)
        out << ans[i] << '\n';
    delete[] nodes;
}
```