USACO OPEN14 Problem 'code' Analysis

by Mark Gordon

Immediately, this problem appears to be solvable using tree dynamic programming. Given a subtree and information about the digits assigned to the previous four nodes we can compute how many legal ways digits can be assigned to nodes in that subtree. Then we can simply calculate the number of legal assignments for the whole tree and subtract this from 10^N to compute the number of illegal assignments.

However, there are 10^4 ways the previous four nodes could be assigned and there are 20000 nodes. This gives us prohibitively many states (200 million). Fortunately we can simply ignore digits that are earlier than the currently longest matching pattern. This works because any digits before the longest matching pattern are now irrelevant.

This leaves us with N+MP states (where P=5 is the pattern length). Unfortunately, for subtle reasons, this still does not give us a great runtime on some pathological test data. In some cases a node could be reachable by O(M) patterns and could have O(N) children in the tree. This, unfortunately, makes the direct implementation of this algorithm take O(NM) time on such cases.

Resolving this takes some creativity. Let tree_dp[u][S] give the number of assignments to the subtree rooted at u where S describes the up to four previous nodes (dropping digits based on the longest matching pattern). Let assigned_dp[u][S'] give the number of assignments to the subtree rooted at u where S describes the assignment of node u and up to the previous three nodes.

From S we can determine which digits can be assigned to u without directly violating a pattern. This allows us to calculate an S'_d for each digit so that we have

tree_dp[u][S] = sum assigned_dp[u][S'_d] (where d is a valid assigned digit)

A basic way of calculating assigned_dp[u][S'] then is to simply iterate over each child and chop of early digits of S' until it is a partial match on the pattern at the child. This means that assigned_dp[u][S'] is almost equal to assigned_dp[u][S' - earliest digit] except for those children that have a pattern that matches S' exactly. This gives us a hint at how to calculate assigned_dp efficiently.

One approach would be to initially set assigned_dp[u][S'] = assigned_dp[u][S' - earliest digit] and the divide out the results of all children that match S' exactly. Unfortunately, the modulus is not prime which makes division difficult. To get around this, we can calculate the value of assigned_dp[u][S'] for all S' at the same time for a given u.

To do this we recursively maintain a segtree structure. First we update the values for any children that have an exact match for the pattern S'. This allows us to query the segtree and compute assigned_dp[u][S']. Then we recurse on any more specific patterns of S'. Finally, we return any updated values in the first step to their original value.

Using all of these techniques allows for a O((N+MP)(P+D) log(N+MP)) solution. Here's my implementation

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <vector>
#include <cassert>
#include <fstream>
```

```
#include <map>
#include <set>

using namespace std;

int MOD = 1234567;

int madd(int x, int y) {
  x += y;
  if(x >= MOD) x -= MOD;
  return x;
}

int msub(int x, int y) {
  return madd(x, MOD - y);
}

int mmul(int x, int y) {
  return (1ll * x * y) % MOD;
}

#define MAXDIGIT 10
#define MAXN (1 << 15)

int P[MAXN];
set<int> partpath[MAXN];
set<int> fullpath[MAXN];
vector<pair<int, int> > TE[MAXN];

map<int, int> tree_dp[MAXN];

int SEG[MAXN*2];

void tree_update(int pos) {
  for(int i = pos >> 1; i; i = i >> 1) {
    SEG[i] = mmul(SEG[i << 1], SEG[(i << 1) + 1]);
  }
}

int assigned_dp[1 << 4 * 4];
int assigned_dp_set[1 << 4 * 4];
int assigned_tree_start[1 << 4 * 4];
vector<int> assigned_tree[1 << 4 * 4];

void solve_assigned_tree(int u, int S) {
  vector<int> prevvals;
  for(int i = assigned_tree_start[S];
      i < TE[u].size() && TE[u][i].first == S; i++) {
    int v = TE[u][i].second;
    int prev = tree_dp[v][S];
    int pos = MAXN + v;
    swap(prev, SEG[pos]);
    tree_update(pos);
    prevvals.push_back(prev);
  }

  assigned_dp[S] = SEG[1];
  for(int i = 0; i < assigned_tree[S].size(); i++) {
    solve_assigned_tree(u, assigned_tree[S][i]);
  }

  /* Now undo changes to tree. */
  for(int i = assigned_tree_start[S], j = 0;
      i < TE[u].size() && TE[u][i].first == S; i++, j++) {
    int pos = MAXN + TE[u][i].second;
    SEG[pos] = prevvals[j];
    tree_update(pos);
  }
}

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

  int N, M;
  in >> N >> M;

  P[0] = -1;
  partpath[0].insert(0);
  for(int i = 1; i < N; i++) {
    in >> P[i];
    partpath[i].insert(0);
    TE[P[i]].push_back(make_pair(0, i));
  }

  for(int i = 0; i < M; i++) {
    int u; in >> u;
    string pat; in >> pat;

    int S = 0;
    for(int j = 0; j < pat.size(); j++) {
      S = S << 4;
      S |= pat[j] - '0' + 1;
    }

    fullpath[u].insert(S);
    for(int j = pat.size() - 1, v = u; j > 0; j--, v = P[v]) {
      int SS = S & ((1 << 4 * j) - 1);
      partpath[v].insert(SS);
      TE[P[v]].push_back(make_pair(SS, v));
    }
  }
  for(int i = 0; i < N; i++) {
    sort(TE[i].begin(), TE[i].end());
    TE[i].resize(unique(TE[i].begin(), TE[i].end()) - TE[i].begin());
  }

  memset(assigned_dp_set, -1, sizeof(assigned_dp_set));
  for(int i = 0; i < 2 * MAXN; i++) {
    SEG[i] = 1;
  }
  for(int u = N - 1; u >= 0; u--) {
    /* Compute assigned_dp for node u. Static arrays are preferable but we need
     * to avoid looping over the whole array to clear everything.  Therefore
     * assigned_dp_set[S] is used to test if the assigned_* entries have been
     * initialized for the state S. */
    assigned_dp_set[0] = u;
    assigned_tree[0].clear();
    assigned_tree_start[0] = 0;
    for(int i = 0; i < TE[u].size(); i++) {
      int S = TE[u][i].first;
      if(S == 0 || (0 < i && S == TE[u][i - 1].first)) {
        continue;
      }

      assigned_dp_set[S] = u;
      assigned_tree[S].clear();
      assigned_tree_start[S] = i;
      while(assigned_dp_set[S >> 4] != u) {
        assigned_dp_set[S >> 4] = u;
        assigned_tree[S >> 4] = vector<int>(1, S);
        assigned_tree_start[S >> 4] = i;
        S = S >> 4;
      }
      assigned_tree[S >> 4].push_back(S);
    }

    /* Compute assigned_dp[S] for all of the interesting states S computed
     * above. */
    solve_assigned_tree(u, 0);

    /* Compute tree_dp for node u. */
    for(set<int>::iterator it = partpath[u].begin(); it != partpath[u].end();
        ++it) {
      int S = *it;
      while(S && S < (1 << 4 * 3)) {
        /* Pad the pattern with 0s for any unknown digits. */
        S = S << 4;
      }

      int result = 0;
      for(int d = 1; d <= MAXDIGIT; d++) {
        int NS = d << 4 * 4 | S;
        if (fullpath[u].find(NS) != fullpath[u].end()) {
          /* A pattern was matched and the digit cannot be used. */
          continue;
        }

        /* Remove digits until we reach a set value for assigned_dp. */
        for(NS = NS >> 4; assigned_dp_set[NS] != u; NS = NS >> 4);
        result = madd(result, assigned_dp[NS]);
      }
      tree_dp[u][*it] = result;
    }
  }

  int allways = 1;
  for(int i = 0; i < N; i++) {
    allways = mmul(allways, MAXDIGIT);
  }
  out << msub(allways, tree_dp[0][0]) << endl;
  return 0;
}
```