Code Jam 2009 - World Finals

Analysis: Marbles

Last year we were trying to solve different instances of this problem. It took a long time to converge to this particular shape, and even after we settled on the current requirements, we were still tweaking the input limits at 2am the night before the contest to make the problem a bit more interesting. The problem nicely combines together dynamic programming, greedy and graph related notions like biconnected components and trees.

The first step is deciding if a particular configuration is or isn't solvable. If for two colors their corresponding marbles alternate it means that the two pairs of marbles need to be joined by curves on opposite sides of the horizontal line *Y=0*. We can build a graph where the nodes represent pairs of same-color marbles and form a graph with edges between pairs of marbles that alternate. We can draw the paths with no intersection if and only if this graph is bipartite.

Next, for solvable configurations we compute the minimum height. The pairs graph can have many connected components, and for each such component we can choose two ways of drawing the lines (with the first pair of marbles above the *Y=0* line or below it), so in total we would have $O(2^{components})$ configurations. This idea can solve the small case but is too time consuming for the large case.

Solving the large case requires us to use dynamic programming. Our state will be defined by *left, right, height_up* and *height_down*. For each state we compute a boolean value which tells us if the subproblem which uses the set of marbles with indexes from *left* to *right* can be solved in the vertical range *[-height_down .. height_up]*. Computing this value is a bit tricky, what we need to notice is that we can try each of the two ways of drawing the component that starts at index *left*. Then an important observation is that we can draw each path with the maximum height possible if the line is above the X axis or maximum depth possible if the line is below the X axis as long as our drawing is within the *[-height_down .. height_up]* vertical range. Using these ideas we can come up with an $O(n^5)$ algorithm.

We can improve on this solution by using the state *(left, right, height_up)* and for each state finding the smallest height_down for which the subproblem *[left .. right]* is solvable. Now we notice that we should use dynamic programming on pairs of *left* and *right* where connected components of marbles start and finish. This will make *right* uniquely defined by *left*. Thus we have reduced the state space to $O(n^2)$ states. We also notice that the connected components form a tree-like structure where we need to solve the innermost components first and then solve outer components, much like visiting the leaves of a tree first and getting closer and closer to the root. Now each connected component will be analyzed just once at an upper component level so the overall algorithm will take $O(n^2)$ time.

Here's Tomek Czajka's solution:

```
#include <algorithm>
#include <cassert>
#include <cstdio>
#include <map>
#include <string>
#include <vector>
using namespace std;
#define REP(i,n) for(int i=0;i<(n);++i)
template<class T> inline int size(const T&c) { return c.size();}
const int INF = 1000000000;
```

```
int n; // number of types of marbles
vector<vector<int> > where; // [n][2]
vector<int> marbles; // [2*n]

void readInput() {
  char buf[30];
  map<string,int> dict;
  scanf("%d", &n);
  marbles.clear(); marbles.reserve(2*n);
  where.clear(); where.resize(n);
  for(int i=0;i<2*n;++i) {
    scanf("%s",buf);
    string s = buf;
    map<string,int>::iterator it = dict.find(s);
    int m;
    if(it==dict.end()) {
      m = size(dict);
      dict[s] = m;
    } else {
      m = it->second;
    }
    marbles.push_back(m);
    where[m].push_back(i);
  }
}

struct Event {
  int x,t;
  // t=0 start top, 1 end top
  // t=2 start bot, 3 end bot
};

vector<int> vis;

bool cross(int m1,int m2) {
  return
      where[m1][0] < where[m2][0] &&
      where[m2][0] < where[m1][1] &&
      where[m1][1] < where[m2][1] ||
      where[m2][0] < where[m1][0] &&
      where[m1][0] < where[m2][1] &&
      where[m2][1] < where[m1][1];
}

void dfs(int m,int sign) {
  if(vis[m]==sign) return;
  if(vis[m]==-sign) throw 0;
  vis[m]=sign;
  REP(i,n) if(i!=m && cross(m,i)) dfs(i,-sign);
}

vector<vector<Event> > cacheCalcEvents;

const vector<Event> &calcEvents(int startx) {
  vector<Event> &res = cacheCalcEvents[startx];
  if(!res.empty()) return res;
```

```
  vis.assign(n,0);
  dfs(marbles[startx],1);
  REP(x,2*n) {
    int m = marbles[x];
    if(vis[m]==0) continue;
    int nr=0;
    if(where[m][nr] != x) ++nr;
    assert(where[m][nr]==x);
    Event e; e.x=x;
    e.t = (1-vis[m]) + nr;
    res.push_back(e);
  }
  return res;
}

vector<vector<int> > cacheCalcH2;

int calcH2(int a,int b,int h1) {
  if(h1<0) return INF;
  if(a==b) return 0;
  int &res = cacheCalcH2[a][h1];
  if(res!=-1) return res;
  const vector<Event> &events = calcEvents(a);
  res = INF;
  for(int mask = 0; mask<=2; mask+=2) {
    int top=0, bot=0;
    int h2 = 0;
    REP(i,size(events)+1) {
      int alpha = i==0 ? a : events[i-1].x + 1;
      int beta = i==size(events) ? b : events[i].x;
      h2 = max(h2, calcH2(alpha, beta, h1 - top) + bot);
      if(i!=size(events)) {
        switch(events[i].t ^ mask) {
          case 0: ++top; break;
          case 1: --top; break;
          case 2: ++bot; break;
          case 3: --bot; break;
        }
      }
    }
    res = min(res, h2);
  }
  return res;
}

int solve() {
  int res = INF;
  cacheCalcH2.assign(2*n, vector<int>(n+1,-1));
  cacheCalcEvents.clear(); cacheCalcEvents.resize(2*n);
  try {
    REP(h1,n+1) {
      res = min(res, h1 + calcH2(0,2*n,h1));
    }
    return res;
  } catch(int) { return INF; }
}
```

```
int main() {
  int ntc; scanf("%d", &ntc);
  REP(tc,ntc) {
    readInput();
    int res = solve();
    if(res==INF) res = -1;
    printf("Case #%d: %d\n", tc+1, res);
  }
}
```