USACO OPEN08 Problem 'nabor' Analysis

by Richard Peng

There are several ways of attacking this problem, here are three of them in increasing order of elegance (and decreasing order of code length):

Method 1. Build a planar MST on the points

It's easy to see how the rest of the solution follows since we can just take the O(n) edges on the MST whose length are shorter than C and find the connected components. Building the MST is trickier and requires the following result:

For each point p, we partition the plane into 8 pieces around it, with each partition taking up 45 degrees and one of the partition barriers being the x axis. Then we find the closest point to it in each of these regions (breaking ties by x-difference) and add edges between p and these 8 points. The MST is a subset of these edges.

This is highly non-trivial to come up with, but fairly easy to prove. The proof is based on if there is another MST edge pq in one of these regions instead of the shortest edge pp', removing pq and adding pp' and p'q will not make the tree worse.

Then this can be done using eight plane sweeps and range trees (it can be reduced to 4 by considering symmetric cases together). This runs in O(nlogn) time, but is quite involved to implement.

The next two methods rely on rotating the plane by 45 degrees under the following transform: (x,y)->(x+y,x-y). This turns the Manhattan distance into infinite norm, which is dist(p1,p2)=max(|p1.x-p2.x|,|p1.y-p2.y|).

Method 2. Divide the plane into C-by-C buckets.

The division can be done in O(N) time. Note all points in the same bucket are already connected. And points that differ by more than C in either x or y cannot be connected, so it suffices to consider adjacent boxes, which has 2 cases:

- Boxes that differ in only 1 direction. WOLOG it's in the x direction. Then we don't have to worry about y and it suffices to consider the right most in the left box and the leftmost in the right box.
- Boxes that differ in 2 directions. WOLOG, the first box is to the left-top of the second. Then if we sweep bottom-up in the second box, we add points in the first box one-by-one into consideration as they pass below a 'upper' sweepline indicating a difference of exactly C. Then considering the right most point that we've encountered in the top box and the leftmost point that's still above our sweep line suffices.

This method runs in O(nlogn) and uses nothing other than sorting. But it's still a bit involved due to the chain of steps.

Method 3. Divide conquer

We divide the points along the middle, process both halves and attempt to merge the connectivity along the middle.

Note it suffices to consider points within a distance of C of our division line, which we assume is vertical. Note we cannot have two points belonging to different components that are within C of our division line and within C of each other vertically. So we can track, for each component on one side, the 'region' to the right of the swepline that it can cover in O(n) time since these regions cannot overlap with a depth of more than 3. For each of these regions, we check all potential points on the right that are within the height range. This works in O(nlog^2n) if we sort at each stage, O(nlogn) if we apply some care and use merge sort.

Most contestants who solved this problem solved it using methods similar to method 2. A few applied ideas similar to the idea behind 1 in finding closest points in each of the quadrants. I did not notice anything submitted along the lines of method 3.

Below is Bruce Merry's solution:

```
#include <fstream>
#include <algorithm>
#include <complex>
#include <deque>
#include <cstdlib>

using namespace std;

#define MAXN 200000

typedef complex<int> pnt;
struct cow
{
    pnt loc;
    int id;
    bool operator <(const cow &b) const { return loc.real() < b.loc.real(); }
};

static cow cows[MAXN], tmp[MAXN];
static int parent[MAXN];
static int N, C;
static int ans;

static void join(int a, int b)
{
    while (parent[a] >= 0) a = parent[a];
    while (parent[b] >= 0) b = parent[b];
    if (a == b) return;
    if (parent[a] > parent[b]) swap(a, b);
    parent[a] += parent[b];
```

```
    parent[b] = a;
    ans--;
}

static void recurse(int l, int r)
{
    if (r - l <= 1) return;
    int m = (l + r) / 2;
    recurse(l, m);
    recurse(m, r);

    int lp = l;
    int rp = m;
    deque<int> ls, rs;
    int t = 0;
    while (lp < m || rp < r)
    {
        bool left = rp == r || (lp < m && cows[lp].loc.imag()
               < cows[rp].loc.imag());
        int &i = left ? lp : rp;
        deque<int> &me = left ? ls : rs;
        deque<int> &you = left ? rs : ls;

        while (!you.empty() && cows[you.front()].loc.imag()
               < cows[i].loc.imag() - C)
            you.pop_front();
        if (!you.empty() && abs(cows[you.front()].loc.real()
               - cows[i].loc.real()) <= C)
            join(cows[you.front()].id, cows[i].id);
        while (!me.empty() && ((cows[me.back()].loc.real()
               >= cows[i].loc.real()) ^ left))
            me.pop_back();
        me.push_back(i);
        tmp[t++] = cows[i++];
    }
    copy(tmp, tmp + t, cows + l);
}

int main()
{
    ifstream in("nabor.in");
    ofstream out("nabor.out");
    in >> N >> C;
    for (int i = 0; i < N; i++)
    {
        int x, y;
        in >> x >> y;
        cows[i].loc = pnt(x + y, x - y);
        cows[i].id = i;
    }
    sort(cows, cows + N);
    fill(parent, parent + N, -1);
    ans = N;
    recurse(0, N);

    out << ans << " " << -*min_element(parent, parent + N) << "\n";
}
```