2002 Sun Microsystems and TopCoder Collegiate Challenge – Problem Analysis

PlanetX:

This discussion assumes that you have already read the problem statement thoroughly.

The PlanetX problem as originally stated is a cellular autonoma. Each cell (a unit from the lunar surface) that is empty becomes 'X' if any of the 4 cardinal neighbors is 'X', 'O' if any of the cardinal neighbors is 'O' but none of them are 'X', and stays empty otherwise.

A simple implementation of the cellular autonoma just scans the entire array for each step, but there can be up to 9,999 steps required, each of which is on a board of 10,000 x 10,000 cells. This algorithm has running time O(N^3), which cannot be completed in 8 seconds for N = 10,000 (it's not even close).

SKIPPING TO THE END

Let us define distance on the lunar surface as "Manhattan distance", where the distance between a point A and a point B as abs(A.x - B.x) + abs(A.y - B.y). This corresponds to the number of steps taken to move from A to B moving only horizontally or vertically at each step (think of the distance a taxi must drive in Manhattan to get from one place to another).

For each cell there is a minimum distance D to any of the initial probes, and there are one or more of the initial probes that are exactly D away from the cell. The cellular autonoma will assign 'X' or 'O' to the cell after exactly D steps. If any of the closest probes is from planet X the cell will be marked 'X', otherwise it will be marked 'O'. This means we can calculate the final state of a cell directly without running the simulation. An algorithm based on this must examine all 10,000 x 10,000 cells and compute the distance to up to 20 probes (running time O(M*N^2) where M is the number of probes).

I implemented the O(M*N^2) algorithm in both C++ and Java and ran these solutions on two test cases. The first test case has a maximal size lunar surface with a single X probe and a single O probe in opposite corners. The second test case has a maximal size lunar surface with 10 X probes and 10 O probes scattered randomly. Additionally I compiled the C++ version with both no optimization (as during a TopCoder contest) and with maximal optimization. Numerous hand-optimizations were employed to speed the Java and non-opimized C++ code (some of them to the detrement of the optimized C++ code).

	2 probes	20 probes
Non-optimized C++	10.3 s	90 s
Optimized C++ *	4.2 s	46 s
Java	10.8 s	103 s

* - not available in the TopCoder environment

This is still more than an order of magnitude too slow.

DIVIDE AND CONQUER

The fundamental problem with the above two approaches is that they need to iterate over all 100,000,000 cells. Divide and conquer (in particular quadtree recursion) provides a way around this by considering progressively smaller rectangular areas. We can do this because if all four corners of a rectangle on the lunar surface (a rectangle lined up with the underlying cells) are closest to the same probe (ties broken in some consistent fashion), then all of the points in the rectangle are also closest to the same original probe.

The recursion works by computing the closest probe to each of the four corners. If all four are closest to the same probe, then the recursion stops and the solution count is either 0 or H*W depending on whether the closest probe is from planet X or planet Y. If the corners are not all closest to the same probe, then the rectangle is split into 4 smaller rectangles and the process is repeated for each of them. The execution time for a simple Java implementation of the above algorithm follows. (I did not implement it in C++ or attempt to minimze function calls, but there is no object allocation during the recursion.):

	2 probes	20 probes
Java divide and conquer	0.06 s	1.4 s

I don't know the exact worst case running time of this algorithm, but I suspect it is O(N * M * sqrt(M)).

WHY DOES IT WORK?

Consider a single row or column of the lunar surface. Instead of just marking cells 'X' or 'O' identity the individual probe that is closest, breaking ties by preferring the lower numbered probe (and preferring all X probes to O probes). The row or column will be composed of a contiguous ranges of cells closest to a single original probe. For example if 1, 2, and 3 are the initial probes then a row just beneath row 3 might look like

```
 1

        2
            3
1111112222222222233333333
```

Can a slice look like A???B??A (non-contiguous ranges of ancestor probes)?. One of the shortest paths from the original probe A to one of the probes marked A in the example travels through the cell marked B (because in Manhattan distance you can go to the row first and then traverse along it to get a shortest path). This would violate the triangle inequality (the distance from from P to R must be less than or equal to the distance from P to Q + the distance from Q to R), so no, non-contiguous ranges are not possible.

The continuity of the areas closest to an initial probe means that if you check two points in a row (or column) and find that they are both closest to the same probe, then each point on the line between them is also closest to the same probe. This logic can be extended to the four corners of a rectangle as well by "filling" in the top and bottom row, then "filling in" the columns. If each of the

four corners of a rectangle is closest to the same original probe, then each point in the rectangle is also closest to the same probe.

IS THIS THE ONLY WAY TO SOLVE IT?

Another way to solve the problem would be to take pairs of probes and consider the end-state if just those two probes were involved. There are only a few possible patterns, so it would be possible to represent the boundaries in a vector fashion, then do some sort of polygonal intersection among all of the boundaries to determine the shape of the regions closest to each initial probe. The running time of this approach is only related to the number of initial probes, not to the size of the lunar surface.

Computing the boundaries directly becomes much easier if only a single row (or column) is considered at a time, because instead of having to compute the intersection of polygons one must only compute the intersection of intervals on the line. For each of M initial probes there are M-1 intervals to consider (for the M-1 other initial probes). The intersection of two intervals on a line can be computed in constant time, so each row can be processed in O(M^2). There are N rows, so a straightforward implementation of this approach has running time O(M^2 *N). This running time would easily suffice for N = 10,000 and M = 10. (Thanks to dmwright for suggesting this approach.)

For the mathematically inclined, this problem can be thought of as the computation of the Voronoi diagram using Manhattan distance, but the fact that it is on a quantized surface (individual cells instead of a smooth continuum) adds complications.

Rectangular areas don't do a very good job of handling the diagonal boundary that occurs if probes are in opposite corners, so the divide and conquer could be enhanced to work on triangles which could then be chosen intelligently. This would allow the divide and conquer algorithm to have a running time independent of the size of the lunar surface.

CODE

```
import java.util.*;
public class PlanetX
{
//  all probes are stored in the same array, probes[i][2]  is
//  1 if from planet X, 0 if from planet O
int[][]  probes;
int n;

public int colonize(int width, int height,
                    String[] xprobes, String[] oprobes) {
  n = xprobes.length + oprobes.length;
  probes = new int[n][];
  for (int i = 0; i < xprobes.length; ++i) {
    StringTokenizer tok = new StringTokenizer(xprobes[i]);
    int x = Integer.parseInt(tok.nextToken());
    int y = Integer.parseInt(tok.nextToken());
     probes[i] = new int[]{  x, y, 1 };
  }
```

```
  for (int i = 0; i < oprobes.length; ++i) {
    StringTokenizer tok = new StringTokenizer(oprobes[i]);
    int x = Integer.parseInt(tok.nextToken());
    int y = Integer.parseInt(tok.nextToken());
    probes[i + xprobes.length] = new int[]{ x, y, 0 };
  }
  return count(0, 0, width - 1, height - 1);
}

// x1 and y1 are inclusive
int count(int x0, int y0, int x1, int y1) {
  if (x0 > x1 || y0 > y1) return 0;

  int a = closest(x0, y0);
  int b = closest(x0, y1);
  int c = closest(x1, y0);
  int d = closest(x1, y1);
  if (a == b && a == c && a == d) {
    return probes[a][2] == 1 ? (x1 - x0 + 1) * (y1 - y0 + 1) : 0;
  }
  else {
    int xm = (x1 + x0) / 2;
    int ym = (y1 + y0) / 2;
    return count(x0, y0, xm, ym) +
      count(xm + 1, y0, x1, ym) +
      count(x0, ym + 1, xm, y1) +
      count(xm + 1, ym + 1, x1, y1);
  }
}

// returns the index of the closest probe, preferring
// lower indices
int closest(int x, int y) {
  int bestD = Integer.MAX_VALUE;
  int bestI = -1;
  for (int i = 0; i < n; ++i) {
    int d = Math.abs(probes[i][0] - x) + Math.abs(probes[i][1] - y);
    if (d < bestD) {
      bestD = d;
      bestI = i;
    }
  }
  return bestI;
}
}
```