Sardar Vallabhbhai National Institute of Technology, Surat

Algorithms & Computational Complexity

Name - Dilip Puri
ID - P18CO008

Assignment 4

Abstract

In Computer Graphics, when we have multiple layers(2 or more) while rendering these layers on 2D display so which layer will be rendered and in what order? in this fashion often we can compute things faster. This is one the basic problem of computer graphics. Similarly, in our case, there is a set of 2D lines and we'll draw these lines on 2D a plane. If we see these lines from top of plane so which lines will be visible from top view1(look down from $y = \infty$)?

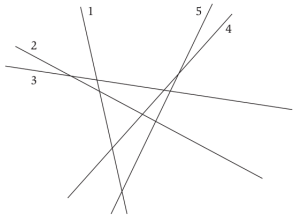


Figure 1: A set of lines

Problem Statement

Hidden surface removal is a problem in computer graphics that scarcely needs an introduction: When Woody is standing in front of Buzz, you should be able to see Woody but not Buzz; When Buzz is standing in front of Woody ... well, you get the idea.
The magic of hidden surface removal is that you-can often compute things faster than your intuition suggests. Here's a clean geometric example to illustrate a basic speed-up that can be achieved. You are given n non vertical lines in the plane, labelled $L_1, L_2, \ldots, L_n$ with the i^{th} line specified by the equation $y = a_i.x + b_i$. We will make the assumption that no three lines all meet at a single point. We say line L_i is uppermost at a given x-coordinate x_0 if its y coordinate at x_0 is greater that the y coordinates of all the other lines at $x_0 : a_i.x_0 + b_i < a_j.x_0 + b_j$ for all $j \neq i$. We say line L_i is visible if there is some x coordinates at which it is uppermost intuitively, some portion of it can be seen if you look down from $y = \infty$.
Give an algorithm that takes n lines as input and in $O(nlogn)$ time returns all of the ones that are visible.

Assumptions

- Lines are non vertical($slope \neq \infty$).
- No three lines all meet at a single point.

Base cases

0.1 Single line

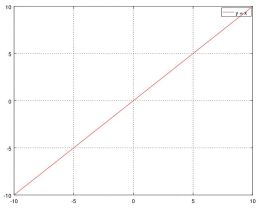
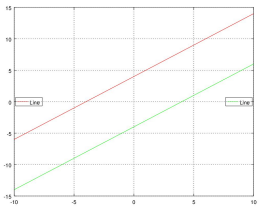
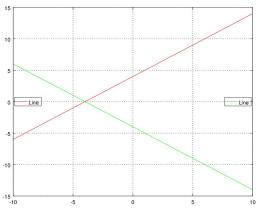


Figure 2: Single line(visible)

0.2 Two lines

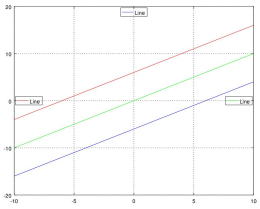


(a) Both lines are parallel
(visible: red line)

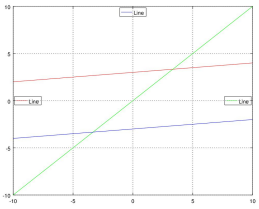


(b) Lines with different slopes
(visible: both)

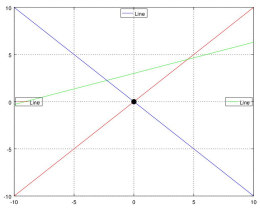
0.3 Three lines



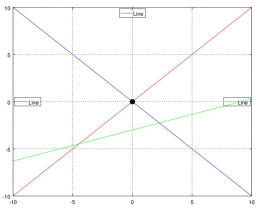
(a) All three lines are parallel
(visible: red)



(b) Two lines parallel with one difference slope
(visible: red, green)



(a) Different slope and line above to point
(visible: all)



(b) Different slope line is below to point
(visible: red, blue)

Observations

Before going to conventional approach we need to understand some important observations.

- If we start by sorting the lines in order of increasing slope. Notice that the first and last lines in this order will always be visible.
- If we have two lines with different slope then both lines are visible, lower slope line visible to left with respect to intersection point and higher slope line visible to right region.
- If we have two visible line and we add 3^{rd} line to check whether it is visible or not? So we will find intersection point and check where it is lying in the left region or in the right.

Conventional Approach

First, we have a set of multiple lines now we will check which which lines are visible so we'll check one-by-one and add line to visible set if it is visible.

Algorithm 0: Conventional Approach

```
   input  : A set of line(s)
   output : visibleSet(lines, intersection points)
 1 visibleSet=empty;
 2 def line():
 3 |   label;
 4 |   slope;
 5 |   intercept;
 6 def point():
 7 |   x-coordinate;
 8 |   y-coordinate;
 9 Function addLine(l: line):
10 |   return line added or not;
11 foreach i in the input set do
12 |   if i==1 then
13 |   |   visibleSet.addLine(line);
14 |   else if i==2 then
15 |   |   visibleSet.addLine(line1, line2);
16 |   |   visibleSet.addPoint(line1, line2);
17 |   else if i==3 then
18 |   |   visibleSet.addLine(line1, line2, line3);
19 |   |   visibleSet.point(line1, line2, line3);
20 |   else
21 |   |   visible.addLine(line);
```

Analysis

1. taking a set of lines ..... $O(1)$
2. adding line by line ... $O(n)$
3. for each line in visible set check that new line is visible or not $O(n)$ ..... $O(n^2)$
4. Time complexity = $O(1) + O(n^2) = O(n^2)$

DnC Approach

In conventional approach we are adding line once at a time but if we observe our base case that is for 3 lines so we can check 3 lines at max at a time. We can divide our line set into two subset and check for visibility. Further we'll get two visible set then we'll merge these two sets. When we combine two sub-solutions, consider each intersection point and consider which sub-solution has the uppermost line at this point.

Algorithm 1: Divide and Conquer

```
   input     : A set of line(s)
   output    : visibleSet(lines, intersection points)
 1 visibleSet=empty, n=no. of lines in set;
 2 if n==1 then
 3 |   visibleSet.addLine(line);
 4 else if n==2 then
 5 |   visibleSet.addLine(line1, line2);
 6 |   visibleSet.addPoint(line1, line2);
 7 else if n==3 then
 8 |   visibleSet.addLine(line1, line2, line3);
 9 |   visibleSet.point(line1, line2, line3);
10 else
11 |   mid = n/2;
12 |   v1=visible(0, mid);
13 |   v2=visible(n-mid, n);
14 |   merge(v1,v2);
15 Function merge(set: visibleSet1, set: visibleSet2):
16 |   return a set of visible lines, a set of intersection points;
```

Analysis and Proof

When we divide our problem into sub-problems so finally those problems will divided in our base cases. At time of merging we need to consider slope and intersection point.

1. Suppose our time complexity is $T(n)$
2. time complexity for sub-solution is $T(n/2)$
3. in the algo we're dividing problem to 2 sub-problems which is $T(n/2)+T(n/2) = 2T(n/2)$
4. when we merge sub-problems which is $O(n)$
5. so finally our recurrence relation is $T(n) = 2 * T(n/2) + O(n)$
6. Time complexity $T(n) = 2 * T(n/2) + O(n)$ which is $O(nlog_2n)$

Performance Evaluation