COMP 251 Assignment 2

Jeremie Poisson - McGill ID : 260627104

McGill School of Computer Science – October 15th 2015

1 Exercise 5-5

Given n linear equations in a plane, a *visible* line is defined as follows :

We say that a line, say L_i, is visible if there is some x-coordinate at which L_i is uppermost. Given the fact that we're only considering lines in a 2D space, L_i is *uppermost* at a x_0 if its y-coordinate is greater than the y-coordinates of all other lines at x_0.

We need to find an efficient way to, given n lines as input, return all lines that are visibles within that 2D space.

The **brute-force** solution for this problem would be to calculate the y-coordinate of each n line equations at any given x positions. To determine whether a line is *visible* or not, we would simply store each *uppermost* line (the line that has the greater y-coordinate) at any x. An implementation of this solution is $O(n^2)$ and is inefficient as it requires testing *every* single lines at *every* position x.

However, it is possible to consider a better approach to this problem using a divide-and-conquer algorithm.

Firstly, we split up the x-axis of the 2D plane in multiple sections (say X sections). We then after calculate the intersection of every couple of two lines (order independent) to check whether or not there is an intersection in a given section based on the x coordinates. Given n lines as the input, that is $n/2$ couples of functions $f(x)$ and $g(x)$. If there is no intersection within the x-coordinates of that section, we can safely consider the function having the greater y-coordinate (once evaluated to a x of that section) to be set as visible. If, however, there is some intersection(s) within that section, we can consider visible both intersecting functions $f_i(x)$ and $g_i(x)$ having the greatest y-coordinate intersection in the current section.

```
let $sections$[] be the splitted sections of the $x$ axis
let $visibles$ be the set of visible lines
For Each ( $sections$ )
    let $x_0$ and $x_f$ be the x-coordinates of the current section
    let $intersections$[] be the list of intersection(s) for every couple
    let $max_f$ be the function with the greatest y-coord. within that section
    For Each ( couples of two (2) lines (order-independent) )
        let $f(x)$ and $g(x)$ be these two lines
        evaluate $f(x_0)$ and $g(x_0)$ and put the greatest one in $max_f$
        let $y_i$ be the y-coordinate of the intersection of $f(x)$ and $g(x)$
        let $x_i$ be the x-coordinate of the intersection of $f(x)$ and $g(x)$
        If ( $x_i$ is within the $[x_0, x_f]$ range )
            add ($f(x)$, $g(x)$, $y_i$) to the intersections list
    If ( $intersections$ is empty )
        add $max_f$ to the $visible$ set
    Else ( $intersections$ is not empty )
        let $f_i(x)$ and $g_i(x)$ be the fn. having the greater y-coordinates $y_i$
        add $f_i(x)$ and $g_i(x)$ to the $visible$ set
```

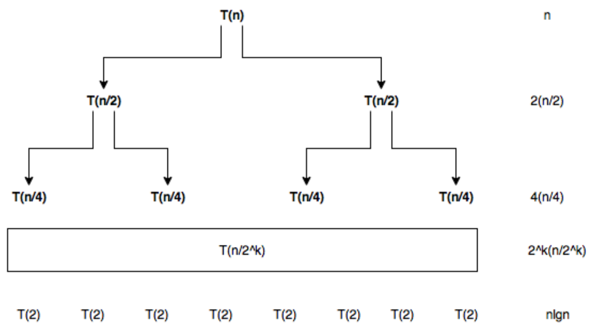
1.1 Recurrence relation

It is possible to define the recurrence relation being an upper-bound of the worst-case running time of such algorithm.

$$T(n) = \text{worst case running time}$$

$$T(n) \leq \begin{cases} 0, & \text{if } n = 1. \\ [T(n/2)] + [T(n/2)] + 2n, & \text{otherwise.} \end{cases} \tag{1}$$

Both the $T(n/2)$ are for solving every $n/2$ couples of equations in both equally splited-up halves of the x-axis. The $2n$ is for finding the best intersection of both halves in worst case.



Therefore, it follows that the algorithm is $O(n \cdot lgn)$.

2 Exercise 5-3

We basically want to test whether there is a subset of $\geq n/2$ elements using an algorithm that requires $O(n \cdot lgn)$ usage of the equivalence tester machine in a recursive fashion.

```
card_test ( set of cards $C$ )
    if ( *there is only 2 cards in $C$* )
        test using equivalency tester; return a card if equivalent
    else if ( *there is only 1 card in $C$* )
        return that card
    else ( *there is more than 2 cards in $C$* )
        let $first\_half$ be the first half of $C$ (length $n/2$)
        let $sec\_half$ be the second and last half of $C$ (length $n/2$)
            if ( recursive call card_test($first\_half$) returns a $eq$ card )
                *test eq over all $C$ using equivalency tester*
            else if ( recursive call card_test($sec\_half$) returns a $eq$ card )
                *test eq over all $C$ using equivalency tester*
            *return the card eq for which there is $> n/2$ equivalent card*
```

If there exist such a set where there are more than $n/2$ cards that are compromised within a set C, splitting up the set C in two halves guarantees us to find, at least, a pair of compromised (equivalent) cards in one of the two halves. Once we have found such a pair, we simply iterate over the whole set C finding all of the equivalent cards within the subset of size $> n/2$.

This is also a divide-and-conquer algorithm. Hence, it is possible to find an upper-bound to the worst case running time :

$$T(n) = \text{worst case running time}$$

$$T(n) \leq \begin{cases} 0, & \text{if } n = 1, 2. \\ [T(n/2)] + [T(n/2)] + 2n, & \text{otherwi se.} \end{cases} \tag{2}$$

The two $T(n/2)$ are for the recursive calls over a set of $n/2$ elements. The $2n$ if for searching (iterating) over all of the subset C once we have found an equivalent pair of cards ($2n$ being the worst case).

Therefore, as seen in exercise 1, this algorithm is $O(n \cdot lgn)$.

3 Recurrences solving - Pan

3.1 Recurrences solving

In order to solve the following recurrences, we might use the **Master Theorem** to find a bound on the worst case running time.

$$T(n) = aT(n/b) + f(n)$$

$$\textbf{Case 1 :}\ f(n) \text{ is } \theta(n^L) \text{ for cste L } < \log_b a \ \Rightarrow T(n) \text{ is } \theta(n^{\log_b a})$$

$$\textbf{Case 2 :}\ f(n) \text{ is } \theta(n^{\log_b a} \log^k n) \text{ for } k \ \geq 0 \ \Rightarrow T(n) \text{ is } \theta(n^{\log_b a} \cdot \log^{k+1} n)$$

$$\textbf{Case 3 :}\ f(n) \text{ is } \Omega(n^L) \text{ for cste L } < \log_b a$$

and $f(n)$ satisfies the regularity condition $a \cdot f(n/b) \leq c \cdot f(n)$ for some $c < 1$ and all large n

$$\Rightarrow T(n) \text{ is } \theta(f(n))$$

(a) $T(n) = 25T(n/5) + n$

We use **Case 1** of the Master Theorem.

$$1 < \log_5 25 = 1 < 2$$

Therefore, the solution for this recurrence is :

$$T(n) \text{ is } \theta(n^2)$$

(b) $T(n) = 25T(n/5) + \log n$

We use **Case 2** of the Master Theorem.

$$k = 1 \geq 0$$

Therefore, the solution for this recurrence is :

$$T(n) \text{ is } \theta((n \cdot \log n)^2)$$

(c) $T(n) = 25T(n/5) + n^2$

(by induction on $T(n)$) $T(n)$ is $\theta(n^2)$

Base Case :

$$T(1) = \theta(1) \leq c(1)^2$$

for all constant $c \geq 1$

Induction Step : Assuming $T(k) \leq c \cdot k^2$ for all $k \leq n$

$$
\begin{aligned}
T(n) &= 25T(n/5) + n^2 \\
&\leq 25 \cdot c(\frac{n}{5})^2 + n^2 \\
&\leq 25 \cdot c \cdot (\frac{n^2}{25}) + n^2 \\
&\leq c \cdot n^2 + n^2
\end{aligned}
$$

where n^2 is the residual

$$\leq c \cdot n^2$$

if residual ≥ 0; which is true for all n (positive fn)

Hence, $T(n)$ is $O(n^2)$

(d) $T(n) = 25T(n/5) + n^3$

We use **Case 3** of the Master Theorem.

$$
\begin{gathered}
3 > 2 \\
25 \cdot (\frac{n}{5})^3 \leq c \cdot n^3 \\
\frac{n^3}{5} \leq c \cdot n^3
\end{gathered}
$$

true for $c = 1/5 < 1$ and all large n

Therefore, $T(n) = \theta(n^3)$

3.2 Pan - Matrix Multiplication

At the time, Pan demonstrated that it was possible to achieve a $\theta(n^{log_{70} 143640})$ when multiplying 70x70 matrices.

In 1980, it has been shown that it is possible to multiply any matrices with a $O(n^{2.374})$. To get a sense of how many scalar multiplications are needed to multiply 70x70 matrices using the newer method, we simply need to calculate the worst-case running time on a 70 base with $O(n^{2.374})$. Since scalar multiplication is constant time, we can simply assume each multiplication on a unitary basis.

$$\log_{70} 143640 = 2.795122$$

$$\log_{70} n = 2.374 \Rightarrow 70^{2.374} \approx 24003$$

That is about 24003 scalar multiplications. Which is a considerable improvement of 119637 fewer operations.

4 Exercise 3-11

An efficient solution to this problem would be to represent the trace data in a graph where each node is a computer and an edge is whenever a communication between two computer has been established. Thereafter, to deduce whether or not a computer C_b has been infected after a time t_k, we simply have to verify that the computer C_b is in the **connected components set** of C_a and make sure that the communication has been established prior to a certain time constraint.

4.1 Constructing the graph

Firstly, we need to construct an undirected graph using the trace data of the situation. Here are the important facts to consider when building this graph:

1. As the trace data is given ordered by the time at which the communication has been established, we could build up a graph of such data with the useful property that the established time of the communications increases as we traverse each subsequent outer layers of the graph.
2. We don't have to worry about multiple subsequent communication between each pairs. *As stated by the problem set*, each pair of computers communicates once.
3. We define the communication of two computers C_i and C_j if either one of the following pair is in the trace data : (C_i, C_j, t_k) or (C_j, C_i, $t_{k'}$).
4. Again, as we are presented the trace data in a time ordered fashion, we only have to record the first trace entry of these two computers and the lowest time (in this case it will be the first entry) at which they have communicated.

Here is a general algorithm to build up such a graph :

```
Initialize the graph $G$ to an empty graph
For each ( trace data triple $T(C_a, C_b, t_k)$ )
    If ( $C_a$ is not present in the graph )
        insert $C_a$ in the graph
    If ( $C_b$ is not present in the graph )
        insert $C_b$ in the graph
    If ( $C_a$ does not have an edge joining $C_b$ )
        create an edge joining $C_a$ and $C_b$ with the $t_k$ being its time value
```

Applying the above algorithm would create a graph that will be used later to determine the connected set of an infected computer C_i. Following this algorithm, the resulting graph have all of the desired properties all listed above.

4.2 Determining whether a computer have been infected

Question : Given a collection of trace data (uniquely paired and time-ordered), if a virus was introduced from a computer C_a, could it have infected a computer C_b by time y.

In order to determine the connected set component of a node C_i in a graph, we can trivially apply a breath-first-search from node C_a. Once the *complete* connected set has been established, we simply have to check the following :

1. is C_b in the connected set of C_a
2. is there a path from C_a to C_b where each edges have a time value $\leq y$.

```
let $G$ denote the graph constructed from the trace data collection
let $BFS$ be the breath-first-search tree of G from node $C_a$
let $R$ be the complete connected component of $BFS$
If ( $C_b$ is not in $R$ )
    return False
For each ( $edge\ e\ in\ the\ path\ from\ C_a\ to\ C_b\ in\ BFS$ )
    If ( $e.time\_value > y$ )
        return False
return True
```

The worst case running time of this algorithm would be $O(m_1+n_1+n_2+m_2) = O(2m+2n)$ which is effectively $O(m+n)$ at the end.

m_1 : is the actual height of the constructed graph G from the breath-first-search tree construction execution;

n_1 : is the number of element n of the trace data collection from the complete connected set execution;

n_2 : is the number of element n of the trace data collection required to search if C_b is part of the connected set;

m_2 : is the actual height of the constructed graph G from the breath-first-search tree traversal to make sure that a path to C_b is satisfying the time constraints.

5 Exercise 3-7

The claim is true.

Proof. (by contradiction)

Lets assume that X is a graph of n nodes that is not connected.
Since the graph is un-connected, it must be that there is ≥ 2 nodes.
Let G be its smallest component. Therefore, $|G| \leq n/2$.

For any node v in G, all neighboors of v must be within that same component G.
Hence,

$$\deg(u) < |G| \leq n/2 - 1 < n/2$$

Which contradicts the assumption that all nodes have a degree $\geq n/2$.

□