CS325 - Project 2

Group #6
William Jernigan, Alexander Merrill, Sean Rettig

October 27, 2014

Correctness

Proof of Claim 3: Direct Proof

> Claim 3: If $\{z_1, z_2, ..., z_t\}$ and $\{z'_1, z'_2, ..., z'_s\}$ are two visible sets of lines (each ordered by increasing slope), then the visible subset of $\{z_1, z_2, ..., z_t\} \cup \{z'_1, z'_2, ..., z'_s\}$ is $\{z_1, ..., z_i\} \cup \{z'_j, ..., z'_s\}$ for some $i \geq 1$ and $j \leq s$.

Let $A = \{a_1, a_2, ..., a_t\}$ be the set $\{z_1, z_2, ..., z_t\}$ and let $B = \{b_1, b_2, ..., b_s\}$ be the set $\{z'_1, z'_2, ..., z'_s\}$ for improved clarity.

Prove that $\{a_1, ..., a_i\}$ is visible

Because all elements in A were defined to be visible with respect to each other, any covering line b_q must be from B.
Given that $m_{a_t} < m_{b_1}$, a covered line $a_p \in A$ has $m_p < m_q$.

Prove that for any covered line $a_p, p = i + 1$, all lines to the right of p in A are also covered:
Because a_p is defined to be invisible, then by Claim 2 in the P1 Visible Lines Handout:
$y_{a_{p-1}}(x_{a_{p-1},b_q}) > y_{a_p}(x_{a_{p-1},b_q})$, where x_{a_{p-1},b_q} is the x coordinate where a_{p-1} and b_q intersect.
Because a_{p+1} is defined to not cover a_p, we know that $y_{a_p}(x_{a_{p-1},b_q}) > y_{a_{p+1}}(x_{a_{p-1},b_q})$.
For $x < x_{a_{p-1},a_p}$, a_{p+1} is covered by a_p.
We need to show that a_{p+1} is covered for $x \geq x_{a_{p-1},a_p}$.
Because b_q is covering a_p, $y_{b_q}(x) > y_{a_q}(x)$ for $x \geq x_{a_{p-1},b_q} \geq x_{a_{p-1},a_p}$.
$\therefore p + 1$ is invisible if p is invisible.
This follows for all p.
Then let $p = i + 1$.
$\therefore \{a_1, ..., a_i\}$ is visible and $\{a_p, ..., a_t\}$ is invisible.

Prove that $\{b_j, ..., b_s\}$ is visible

Because all elements in B were defined to be visible with respect to each other, any covering line a_o must be from A.
Given that $m_{a_t} < m_{b_1}$, a covered line $b_r \in B$ has $m_o < m_r$.

Prove that for any covered line $b_r, r = s - 1$, all lines to the left of r in B are also covered:
Because a_p is defined to be invisible, then by Claim 2 in the P1 Visible Lines Handout:
$y_{b_{r+1}}(x_{b_{r+1},a_o}) > y_{b_r}(x_{b_{r+1},a_o})$, where x_{b_{r+1},a_o} is the x coordinate where b_{r+1} and a_o intersect.
Because b_{r-1} is defined to not cover b_r, we know that $y_{b_r}(x_{b_{r+1},a_o}) > y_{b_{r-1}}(x_{b_{r+1},a_o})$.
For $x < x_{b_{r+1},b_r}$, b_{r-1} is covered by b_r.
We need to show that b_{r-1} is covered for $x \geq x_{b_{r+1},b_r}$.

Because a_q is covering b_r, $y_{a_o}(x) > y_{b_o}(x)$ for $x \geq x_{b_{r+1},a_o} \geq x_{b_{r+1},b_r}$.
$\therefore r-1$ is invisible if r is invisible.
This follows for all r.
Then let $r = j-1$.
$\therefore \{b_j, ..., b_s\}$ is visible and $\{b_1, ..., b_r\}$ is invisible.

What we need to prove

Prove that for each line that mergeVisible checks, it correctly determines its visibility (see proof of Claim 1 in Project 1) Prove that each line in the rest of each list is also invisible if an invisible line is found (see proof of Claim 3) Prove that mergevisible is only passed 2 sets of visible lines, since it only works in that situation (this is proven because algorithm 4 only calls mergevisible on lists of length 1 (which are trivially visible) or on the output of mergevisible (which we just proved to be a set of visible lines)

Proof of Algorithm 4: Inductive Proof

Let Y be a list of n lines sorted in ascending order by slope. We claim that Algorithm 4 can determine which lines are visible in the set.

Base case:

If $n \leq 2$, the lines are trivially visible because no two lines exist to cover any single line.

Inductive hypothesis:

Assume for any $n \geq 3$, Algorithm 4 can split Y in half, determine visibility for each half, and then merge them.

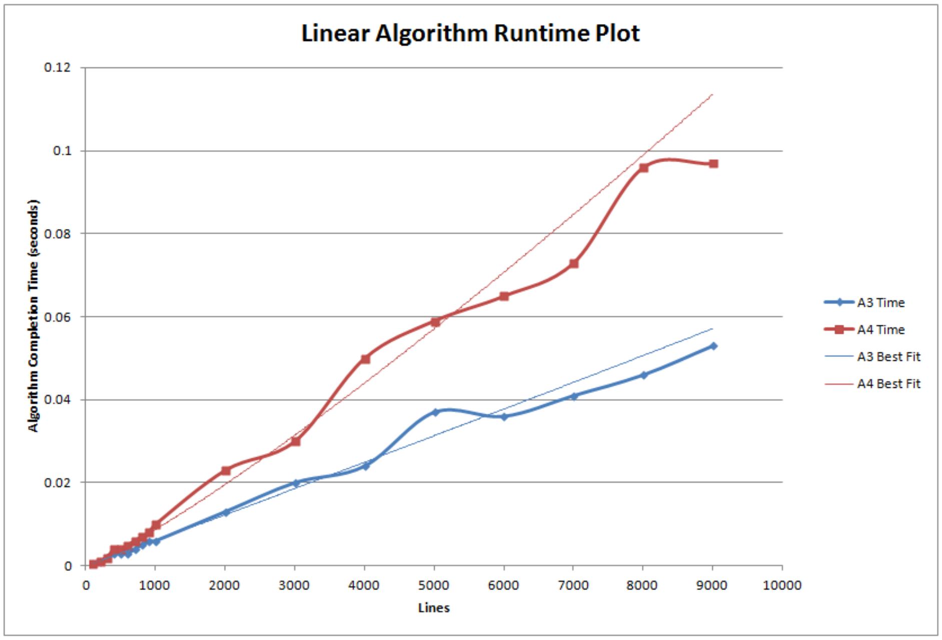
Applying the axiom of induction:

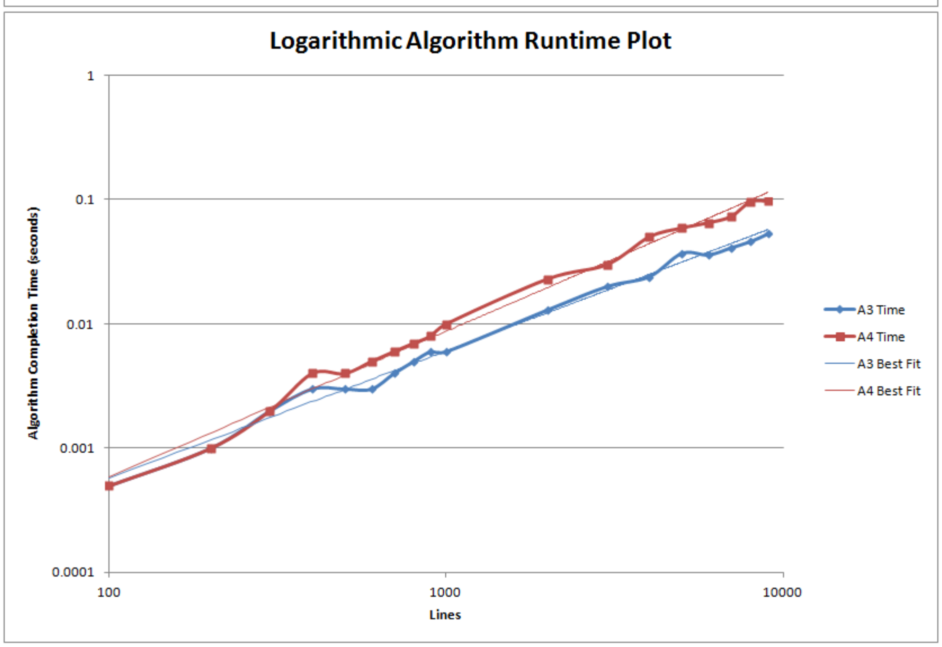
The inductive hypothesis means that "calls to" Algorithm 4 will cut input in half until the base case is reached, where $n \leq 2$ and the input is trivially all visible. Then, Algorithm 4 begins merging two sets of lines, each of which contains lines visible with respect to the lines within its own set. This hypothesis states that those sets will successfully be merged with an output of visible lines, meaning that earlier "calls" to Algorithm 4 also receive input of two sets of visible lines.

Experimental and Asymptotic Run Time Analysis

Experimental Run Time Data

Experimental Run Time Plots





Experimental Run Time Analysis

Algorithm 3: $y = 5 \times 10^{-6} x^{1.0234}$
Algorithm 4: $y = 3 \times 10^{-6} x^{1.1695}$

Given these equations, we can use both the slopes and our code to analyze the run times of the algorithms. We can also determine the biggest instance that can be solved with each algorithm in an hour.

Asymptotic Run Time Analysis

Algorithm 1: $\Theta(n^3)$
Algorithm 2: $\bigcirc(n^3)$
Algorithm 3: $\bigcirc(n^2), \Omega(n)$
Algorithm 4: $\bigcirc(nlogn), \Omega(logn)$
Algorithm 4

Extrapolation and Interpretation

Estimated Number of Lines Per Hour

Algorithm 3: $y = 5 \times 10^{-6} x^{1.0234}$
Algorithm 4: $y = 3 \times 10^{-6} x^{1.1695}$

Discrepancies

We note that the slopes from our experimentally-derived equations are within the asymptotic run-time range of $\bigcirc(n^2)$, $\Omega(n)$ and $\bigcirc(nlogn)$, $\Omega(logn)$ however Algorithm 4, which is theoretically faster than Algorithm 3 actually runs slower. This discrepancy many have many possible contributing factors, including a small sample size, a low timing resolution (particularly for Algorithm 3), how the compiler and system handle arrays and operations, and randomness in run-time present. This randomness in run-time is due to both Algorithms not performing a fixed number of operations, but rather change what operations they run depending on the input lines given. The randomness and how the compiler and system handle the our code are likely the primary causes of these differences.

Pseudocode

```
algorithm1(lines):
    for j in lines[0 ... n]:
        for i in lines[j+1 ... n]:
            for k in lines[i+1 ... n]:
                Xjk, Yjk = intersection(j, k)
                Yi = i.slope * Xjk + i.intercept
                if Yjk > Yi:
                    i.visible = False
    return lines

algorithm2(lines):
    for j in lines[0 ... n]:
        for i in lines[j+1 ... n]:
            for k in lines[i+1 ... n]:
                if i.visible:
                    Xjk, Yjk = intersection(j, k)
                    Yi = i.slope * Xjk + i.intercept
                    if Yjk > Yi:
                        i.visible = False
    return lines

algorithm3(lines):
    vlines = []
    for i in lines:
        vlines.append(i)
        removeCovered(vlines)
    return lines

algorithm4(lines):
    if len(lines) <= 1:
        return lines
    else:
        left = algorithm4(first half of lines)
        right = algorithm4(second half of lines)
    merged = mergeVisible(left, right)
    return merged

def mergeVisible(a, b):

    # Don’t check the ends for visibility because they are always visible.
    i = 1
    j = len(b)-2

    while checking A’s or checking B’s:
        if checking A’s:

            # a[i] is the next line in A.
            # b[j] is the next line in B.
            # a[i-1] is the previous line in A.
            # b[j+1] is the previous line in B.

            # Check the next line in A.
```

```
        x*, y* = intersection(a[i-1], b[j+1])
        testLineY = a[i].slope * x + a[i].intercept
        if y* > testLineY:
            stop checking A's (because the rest of them are invisible)
        else:
            i += 1 # Get ready to check the next line in A on the next iteration

            # Now we check to make sure that we didn't just cover the last line in 'b'.
            if there is a previously added line in j:
                intersectionY = a[i-1].slope * (a[i-1].intercept - b[j+2].intercept) + a[i-1].interc
                testLineY = b[j+1].slope * (a[i-1].intercept - b[j+2].intercept) + b[j+1].intercept
                if intersectionY > testLineY:
                    checkBs = False
                    j += 1

    if checkBs and j >= 0:
        intersectionY = a[i-1].slope * (a[i-1].intercept - b[j+1].intercept) + a[i-1].intercept * (b
        testLineY = b[j].slope * (a[i-1].intercept - b[j+1].intercept) + b[j].intercept * (b[j+1].sl
        if intersectionY > testLineY:
            checkBs = False
        else:
            j -= 1

            # Now we check to make sure that we didn't just cover the last line in 'a'.
            intersectionY = a[i-2].slope * (a[i-2].intercept - b[j+1].intercept) + a[i-2].intercept
            testLineY = a[i-1].slope * (a[i-2].intercept - b[j+1].intercept) + a[i-1].intercept * (b
            if intersectionY > testLineY:
                checkAs = False
                i -= 1

vlines = a[:i] + b[j+1:]
return vlines
```