Manhattan Skyline Problem

Daniel Braithwaite

August 25, 2015

1 Partitioning The Problem

There are two ways of partitioning the Manhattan Skyline problem, one is by partitioning the list of buildings e.g. breaking the list down the middle. The other option is to partition the x-axis.

1.1 Partitioning By Buildings

When dividing the problem by the list of buildings, splitting the problem up is easy. You keep dividing the problem in half until you only have one building, from here calculating the skyline for one building is trivial. However recombining the skylines from two different sets of buildings is where it gets tricky.

1.2 Partitioning By x-axis

Partitioning this way would make combining solutions easier as by the time you reach the base case you would have a solution for some x value and combining would be trivial. The hard part would be the splitting, you would want to split between buildings and try to avoid splitting in the middle of the building as then you would have that building in two lists.

2 Algorithm

The algorithm I designed splits the problem down by breaking up the list of buildings, halving it each time, until a list of size 1 is reached, at which point constructing the skyline for a single building is trivial. Merging the skylines back together is a bit more complicated, it works by taking the two partial skylines and moving through them at each point taking the one with the smallest x, Say we choose the next element from the left skyline then we also need to look at the last added element from the right skyline so we could tell if the left skyline was dropping below the right.

When designing my algorithm I could of separated the main loop into two so that if we reached the end of one skyline we could just append the the rest of the remaining list to the output skyline. However this wouldn't improve the efficiency of the algorithm. I also felt it was more elegant to have it all in one loop.

```
1: procedure recConstructSkyline(buildings, start, end)
2:     if end − start = 1 then
3:         building ← buildings[start]
4:         return skyline for building
5:     middle ← (end − start)/2
6:     return mergeSkylines(recConstructSkyline(buildings, start, middle+
   start), recConstructSkyline(buildings, middle + start, end))
7:
8: procedure mergeSkylines(leftSkyline, rightSkyline)
9:     skyline ← emptylist
10:
11:    left ← 0, right ← 0
12:
13:    while left < leftSkyline.size or right < rightSkyline.size do
14:        if left >= leftSkyline.size then
15:            currentPair ← rightSkyline[right]
16:            prevPair ← leftSkyline[left − 1]
17:            right + +
18:        else if right >= rightSkyline.size then
19:            currentPair ← leftSkyline[left]
20:            prevPair ← rightSkyline[right − 1]
21:            left + +
22:        else if leftSkyline[left].x < rightSkyline[right].x then
23:            currentPair ← leftSkyline[left]
24:            prevPair ← rightSkyline[right − 1] if possible
25:            left + +
26:        else if rightSkyline[right].x < leftSkyline[left].x then
27:            currentPair ← rightSkyline[right]
28:            prevPair ← leftSkyline[left − 1] if possible
29:            right + +
30:        else
31:            currentPair ← leftSkyline[left]
32:            prevPair ← rightSkyline[right]
33:            right + +
34:            left + +
35:
36:        if prevPair is unassigned then
37:            add currentPair to skyline
38:            return to top of loop
39:        else if currentPair.height > prevPair.height then
40:            skyline.add(currentPair)
41:        else
42:            change ← [currentPair.x, prePair.height]
43:            if skyline.lastElement.height! = change.height then
44:                skyline.add(change)
```

3 Proof Of Correctness

3.1 Assumption

1. The input is a list of triples of the form $(x_1, x_2, height)$, where $x_1 < x_2$

3.2 Requirements

1. Sorted in ascending order by x
2. The height at any x position is the maximum height of any building spanning that x
3. Any change in height must occur at the start or end of a building x

3.3 Direct

There is only one building as $n = 1$ say we had the building (x_1, x_2, h) the skyline is trivial to create, we get the following $(x_1, h), (x_2, 0)$. Therefor clearly the requirements hold.

1. As in the input $x_1 < x_2$ then the skyline is sorted
2. There is only one building so we can clearly see this is true as for any $x_1 <= x <= x_2$ the height is that of the building
3. By the way we constructed the skyline, part three must hold as the only times the height changes is at x_1 and x_2 and these two x values are at the start and end of the building

3.4 Divide

When we are splitting the data we don't touch the triples so the assumption still holds
$n_i < n$ as we are splitting the list into two halves from the middle so $n_i = n/2 < n$

3.5 Combine

3.5.1 Requirement 1

First we show that the output skyline is sorted, showing that part 1 of the requirements is true for the out. To show this we will use a loop invariant.

Loop Invariant

1. Skyline is sorted

Initialization: Here $skyline \leftarrow \varnothing$ and therefore is trivially sorted

Maintenance: We are choosing the next x to be added from the left and right skylines, both of which are sorted. We look at the next element in both the lists and take the one with the smaller x adding this to the skyline. so we can see if we have the skyline $x_1, ..., x_k, x_{k+1}$ then $x_1 < ..... < x_k < x_{k+1}$).

Therefore part one of the requirements holds.

3.5.2 Requirement 2

Now we need to show that at any x the height is the highest of any building at that x. We are merging two skylines, we will refer to these as the left and the right skylines.
We start by looking at how we choose the next x to be added to the skyline. When choosing this we look at the next element in the left and right skylines that are to be merged and pick the one with the smaller starting x. We set this to be currentPair.

Based on what list we choose currentPair from we set prevousPair to be the last element added to the output skyline from the other list. We are interested in this prevousPair because we want to keep track of the height in the other list so if the currentPair height drops below it then we know that the skyline is overlapping.

There are three cases

1. $currentPair.height > prevousPair.height$
2. $currentPair.height < prevousPair.height$
3. $currentPair.height = prevousPair.height$

Case 1 and **Case 3** are trivial, we just add the pair $(currentPair.x, currentPair.height)$ to the skyline

Case 2 requires some more thought, we cant just add the pair $(currentPair.x, currentPair.height)$ as the height of the other skyline is still prevousPair.height. So we add the pair $(currentPair.x, previousPair.height)$.

3.5.3 Requirement 3

Requirement part 3 holds as the next x to be added is chosen from ether the left or right skyline, we know that all the requirements hold for the left and right skylines so therefore any x taken from them must be where a building starts or finishes.

4 Complexity

4.1 Direct

The complexity of solving the direct solution is $O(1)$, when we have one building constructing the skyline is very simple its just two tuples. Constructing this is clearly is $O(1)$.

4.2 Divide

The complexity of the divide part of the algorithm is also $O(1)$ as nothing actually happens here we just recursively pass the array of buildings with a section for the function to look at.

4.3 Combining

Combining has a complexity of $O(n)$ where n is the total number of buildings represented by the left and right skylines, we see this when we look at the following. The algorithm for merging skylines takes two skylines as input and will return one representing the combination. Each of the two input skylines represents $n/2$ buildings so at worse case (all the buildings being disjoint) each of the skylines will contain $2 * n/2 = n$ tuples. Meaning the loop will iterate at most $2n$ times. Thus giving us $O(n)$

4.4 Asymptotic Cost

So we want to solve the following recurrence relation using the master method we have

$$T(n) = \begin{cases} 1 & n == 1 \\ 2T(n/2) + n + 1 & otherwise \end{cases}$$

We fit what we have to the following from the master method

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

where we know that

$$a = 2, b = 2, f(n) = n + 1$$

so we let

$$\alpha = \log_2 2 = 1$$

and we see that

$$f(n) \in \theta(n^1) = \theta(n)$$

so finally by the master theorem we see that the asymptotic cost is the following

$$\theta(n \log_2 n)$$

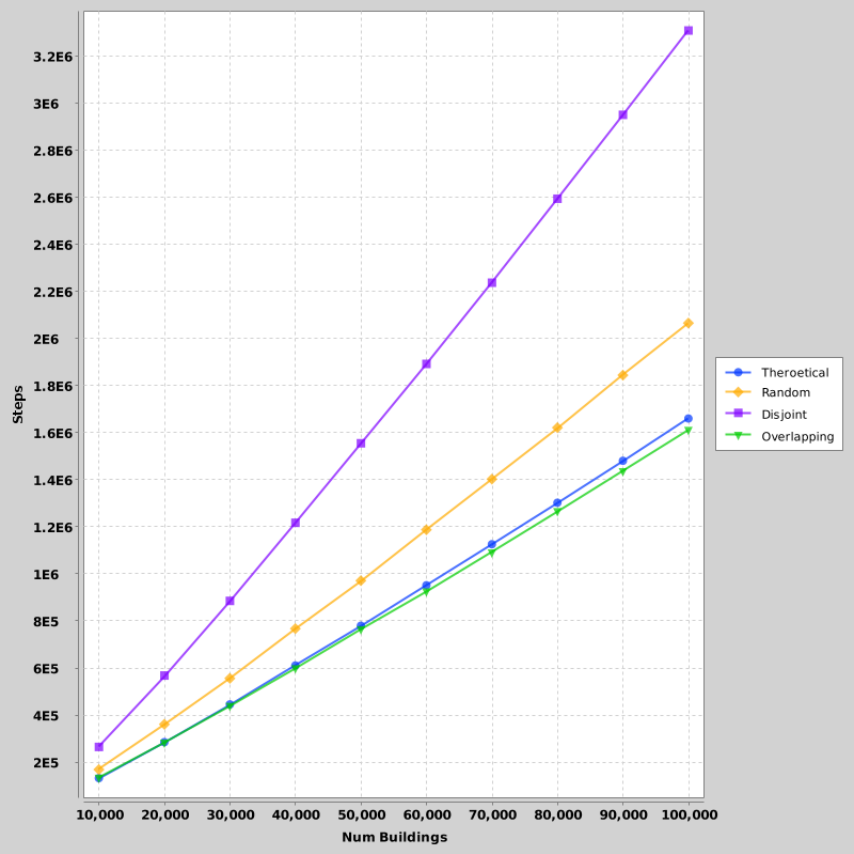
5 Implementation

When implementing the algorithm I used a list to store the buildings and to store the skylines. There can be a cost to adding to the list when the internal array gets full. We can get remove this cost as we know the maximum size of the new skyline so we can initialize the list with the maximum possible size. Using a list also meant it was very easy to divide up the buildings using indices without actually having to create any new lists.

6 Testing

I tested my algorithm on a number of randomly generated test cases from size 10000 to 100000, the results where promising as they matched up perfectly with my theoretical complexity. At each input size there where three test cases. The steps plotted on the y-axis is the number of times the loop in the merge function executes for each of the problem instances as everything inside this loop was constant time so all we need to worry about is the number of times it executes

1. **Worst Case** where all the buildings are disjoint, i.e. none of them overlap
2. **Average Case** where all the buildings are randomly distributed
3. **Best Case** where all the buildings overlap



As you can see in the graph above the theoretical complexity is bounded from above by the worst case(purple line) and bounded from below by the best case (green line). This matches up with the theoretical complexity $nlog_2(n)$