Time Analysis of the Subproblem LSA

There are many steps we take in our LSA. Let's see which ones consume the most time.

Enter While Loop
    #1 Check Increment Current State
    Enter For Loop
        Check Node Not Already In
            #2 Check Feasibility
            #3 if... Push New State
            #4 if not... Amend New State
            #5 Check domination
        End
    End
End

5 steps might be responsible for our time. Let's see how they stack up when we measure them.

My strategy is to use the Julia function datetime2unix(now()) from the Julia Dates package. This will give me the time in seconds and allow me to take an absolute measurement of how much time has passed by measuring the difference between two calls to this function.

We will load up an array of time differences which will represent how long the computer spent calculating each of the 5 bolded sections. These are potential bottlenecks for the subproblem. The sum of the elements within each individual list will tell us how much time was burned at each step. Finally, this will be graph to determine which steps we need to optimize.

#1 Check Increment Current State
There are three possibilities.

```
if (L[current_state] == n+2) | (~A[current_state])
    current_state += 1
    if current_state > total_state
        print(current_state, " ", total_state, " nothing more to check")
        break
    else
        continue
    end
end
```

Either we enter the if loop, or we don't. If we don't, put a timer at the bottom if the if loop after the last end in the image. If we do enter the loop, either we break or continue; in either case, we will not hit the timer I just described. Put two timers: one before the break, one before the continue.

These are three timers which will never intersect.

#2 Check Feasibility
There are two possibilities.

```
if L[current_state] != 1
    cur_time = last(T[current_state])
    dist_nec = travel_time[L[current_state], i]
    old_job_time_nec = load[L[current_state]-1] #out of our parameters this one has
    new_job_time_nec = load[i-1]
    new_window_close = windows[i-1][2]

    #if invalid, don't bother
    if cur_time + dist_nec + old_job_time_nec + new_job_time_nec > new_window_close
        continue
    end
end
```

Here either we enter the if loop or we don't. If we don't, put a timer at the bottom. In fact, even if we do and don't hit the inner if, we also hit the bottom so we don't need an extra timer. The other timer comes if we hit the invalid state, in which case before the continue we insert a timer.

The two timers will never intersect, though there will be a significant difference in times when we hit the first timer because sometimes you've just calculated things and other times you don't need to. Also note that for all timers I'm not yet checking whether we entered or didn't enter some place; this should be discernible from the times themselves.

#3 If ... Push New State
There is one possibility: We enter the if part. I mean...if we don't enter then we won't actually create that timer, so no worries. We'll have longer time check arrays in the 1st, 2nd, and 5th elements as opposed to the 3rd and 4th which split. (I'm betting that the vast majority of paths created will not be dominated by another one, causing the 3rd case to be much more frequent than the 4th.)

#4 if not ... Amend New State
Same idea.

#5 Check Domination
There are two possibilities. Either we hit the break, or we don't. Similar to case 2.

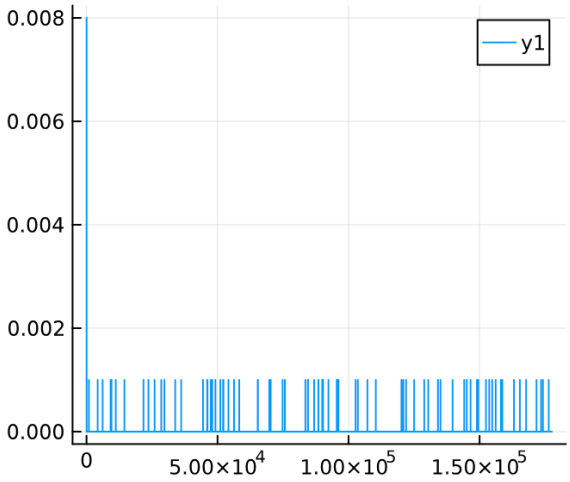
Results

The full algorithm took 71.997 seconds.

The first check (increment current state) took a sum of 0.033 seconds. It is not the bottleneck. In fact, the majority of checks required 0 time because we'd skip right by. There were 16576 such checks, and only checks number 545, 13003, 13501, 14678, and 16576 yielded checks over 1e-10 seconds. There were a total of 16576 routes generated, so only the last check took time—and only because we had a print statement!

The second check (check feasibility of new node) had far more possibilities than the others. There were 16576 routes generated, but this check was run 177614 times—10.71 times as much as 16576. The reason is because a lot of i nodes are not feasible, but we had to first verify their feasibility. But it also only took 0.088 time!

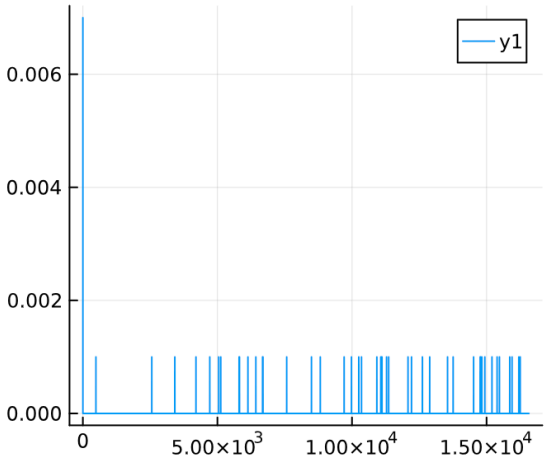
As we can see in the plot below which quantifies how long checks, feasibility checks take very little time. Indeed, most of them take zero time because the current state is 1.



(Note: units = seconds.)

The third check (which happened 16575 times) indicates: If we didn't have a bastardized route from last time (i.e. the last route was not dominated by another route), then we append some things. Let's see how bad this was!

As it turns out, this only took 0.052 seconds. We can draw a graph as well to test what happened. A similar plot emerges.



Each check took similar time to the feasibility check, and on average going through the third section took 3.14e-6 seconds.

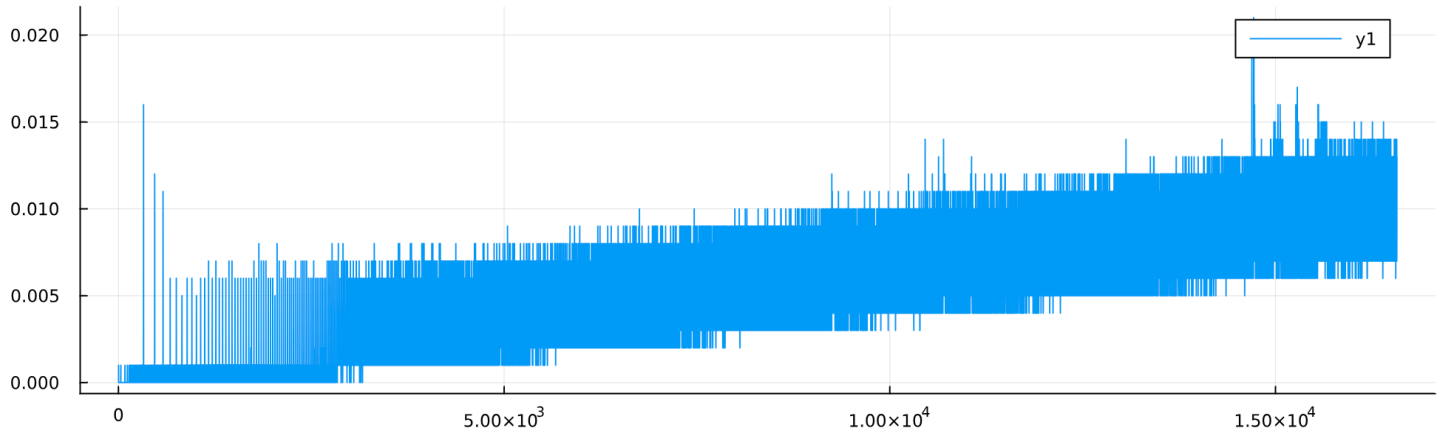
The fourth check never happened so we don't care.

Finally, the culprit is likely to be the 5th section: the n-domination check.

This is it. This thing takes 71.445 seconds, dominating the total time used, which is 71.997 seconds. So it burns **99.2% of our time**.

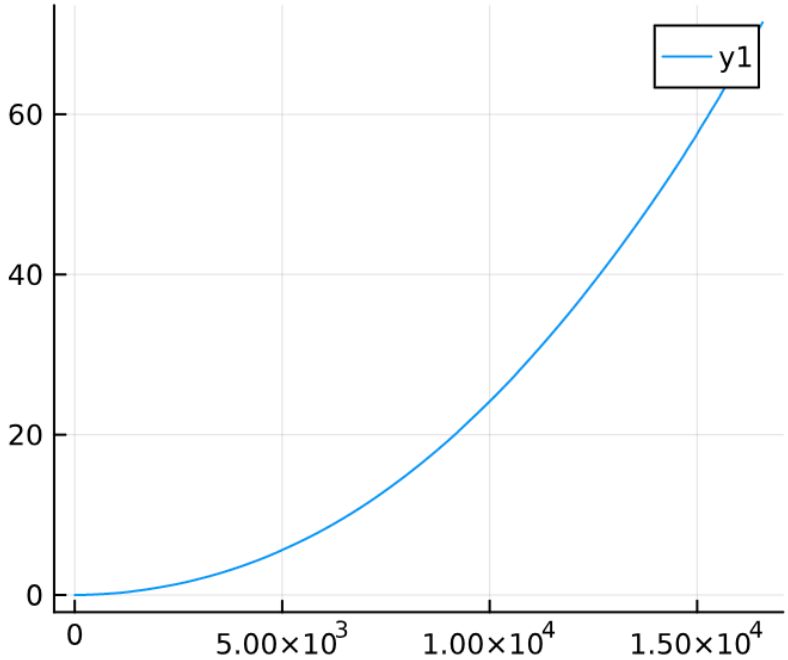
Since this domination check only happens 16575 times, each time we have this check we'd actually go through it. How long does it take? My hypothesis is that we have to run checks of increasing size, so it should be a superlinear trend upward because the latest routes we check are likely to be larger than the previous ones.

This is what it looks like.

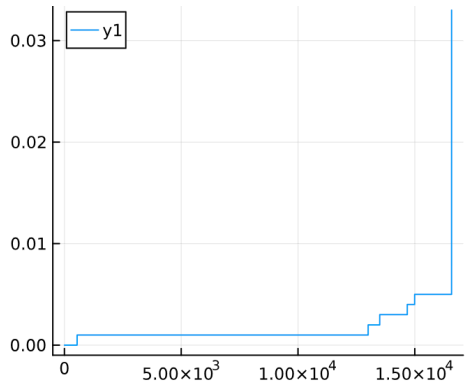


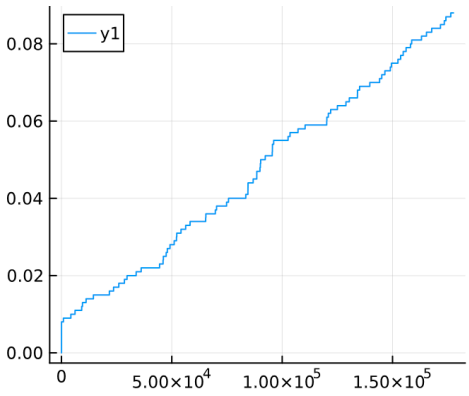
As can be seen, there's an upward trend but some checks are much longer than other ones. The if statement about whether the sets are equal takes off some time. Eventually we are forced to enter the second, deeper if loop because 50% of all routes end at the depot, so if we currently have a route that ends at the depot, a lot of them go to the deeper if loop.

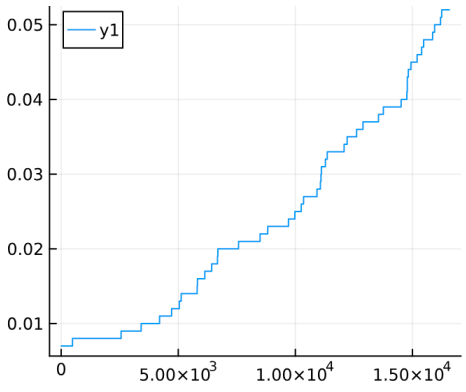
Cumulative time looks like this. Looks quadratic.



Let's also draw the other cumulative plots and see how they do. From left to right on the next page: #1 (check increment current state), #2 (feasibility check), #3 (increment and push a new state).







Huh...not too much pattern to be detected here, after all, there's barely any time burned here anyways.

In summary:

Time Burned: Total	**71.997**	**100.0%**
First Step (current state):	0.033	0.046%
Second Step (feasibility):	0.088	0.122%
Third Step (push route):	0.052	0.072%
Fourth Step (amend route):	0	0%
Fifth Step (n-domination):	71.445	99.233%
Everything Else:	0.379	0.526%

One can imagine that if we scaled up our efforts, the other steps (which account for 0.767% of the entire runtime) will scale up linearly; it appears Julia's mechanism for pushing arrays is either O(1) or barely higher (at least this is not noticeably quadratic; some time trials suddenly take longer).