LSA Time Analysis

What exactly takes the machine so long to do n_jobs = 39 (55 seconds)? We'll figure it out here. From the previous iteration I ran a time analysis using 25 jobs, and back then we had the n-domination check (ndc). 99.233% of time was burned on the ndc, which I mathematically proved was unnecessary and time-consuming. Eliminating that reduced the column generation full loop time from 11 minutes to less than a second.

Back then, there were a few components we checked in the colgen loop which appeared to increase linearly with n. These are: pushing routes, checking conditions, etc. As is apparent now, somewhere along the line there are huge spikes in time with big ramifications for time complexity. It is necessary to analyze the time, especially in light of larger numbers of jobs.

It might be the case that the number of iterations itself drives an otherwise linear process. We'll figure out what happened. Code: datetime2unix(now())

Our timing will proceed as follows:

1. **Start Timing**
    1. Enter the While Loop
        1. **Time Check Start**
            1. Perform check step
        2. **Time Check End**
        3. Enter the For Loop
            1. Check current node not already included
            2. **Time Feasibility Start**
                1. Perform feasibility check
            3. **Time Feasibility End**
            4. **Time Add Start**
                1. Add path
            5. **Time Add End**
2. **End Timing**

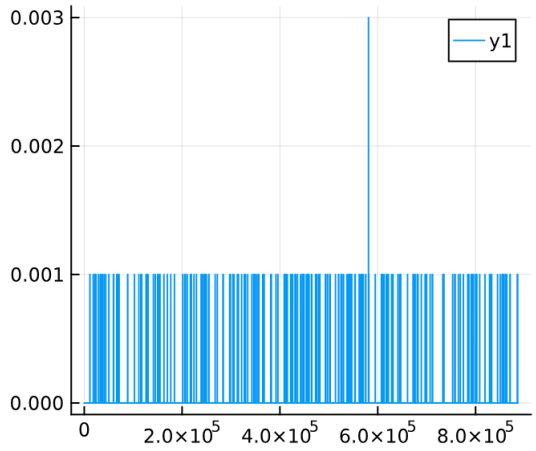
We will have 4 sets of variables to time. We will gather how many times they are iterated over, and how much time is taken by each one. Three of them are in the subfunction.

The first experiment is run on n_jobs = 35, which produced an answer in 10.984 seconds. The check to increment current_state happened 886'848 times. The feasibility check within the loop (with all the i's swirling around) happened 12'879'664 times! (This is nearly 360k iterations of the loop.) Finally, the adding took place 886'847 times.

Is there a true bottleneck? We can find out by assessing how long it took to calculate each representation.

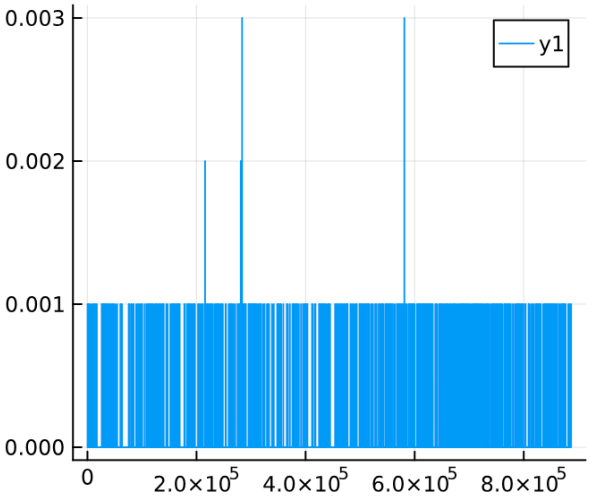
The check to see if we increment the current state took an average of 0.001 seconds each time. The total time taken was 0.204 seconds, which is quite small, but we'll test other n_jobs numbers to see how big/small it really is.

It turns out that only 0.024% of all current-state incrementations took more than 0 seconds (this might, again, have to do with Julia's lack of precision beyond milliseconds). The graph looks like the following.



Now we'll examine the feasibility check. There is so much data (12.9 million datapoints) that Julia refuses to give me the graph. But let's look at the sum which is a whopping 3.986 seconds! Once again, a tiny 0.030% of feasibility checks take more than 1e-4 time (i.e. 0.1 millisecond). But this is 3901 checks that make up a millisecond. It is the sheer number of checks that must be run that indicates we have an issue.

Finally, we examine the pushing of data. This takes a total of 1.098 seconds, which isn't bad. We remove some outliers which took 0.4 seconds and 0.01 seconds.



The weird thing is that these three methods make up 5.288 seconds, only 48% of the entire time! Where did the rest of the time go?

If we run a time check inside sp_lsa, what happens? i.e. have the whole loop inside.

Not much difference; one says 9.929 seconds and the other, 10.130 seconds.

We conclude the following:

1. **All the elementary operations we implement in the SP LSA are as simple as possible.** This is because all steps we've timed take only 0.001 second regardless of how far in the process we are. Furthermore, there is no one obvious bottleneck (unlike in the ndc). This proves that the number of iterations and the relative complexity of elementary operations will seal the fate of how our algorithm does.

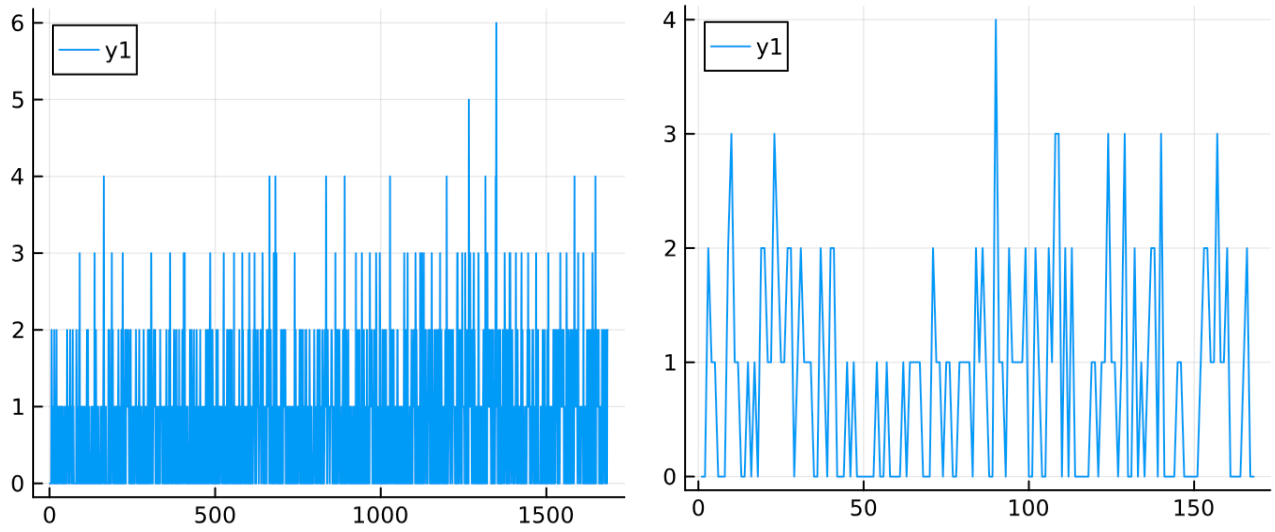
2. **The sheer number of times a code is executed is the reason it takes so long.** Although each time we call the feasibility check appears to need the same amount of time, we use it 12.9 million times.

We are wary of the following:

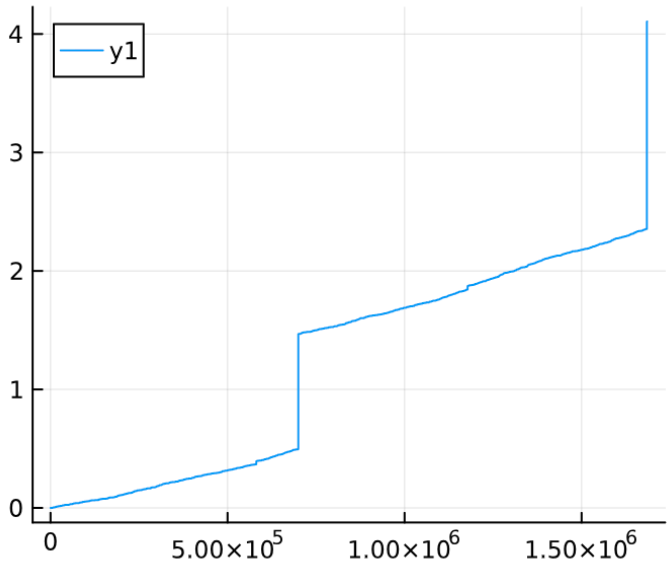
1. **Julia's timing specificity is bad—1 millisecond is not accurate enough because we have millions of times the loop is run. This means several seemingly impossible possibilities cannot be ruled out yet.** If you look at the graph above, looks like either we are at 1 millisecond or zero, but...what if the runs are actually getting progressively longer, but we don't know it because it gets "subducted" beneath true milliseconds well enough? You know what I'm saying? Maybe the 'density' skyrockets at 39 n_jobs which is why it takes so long.

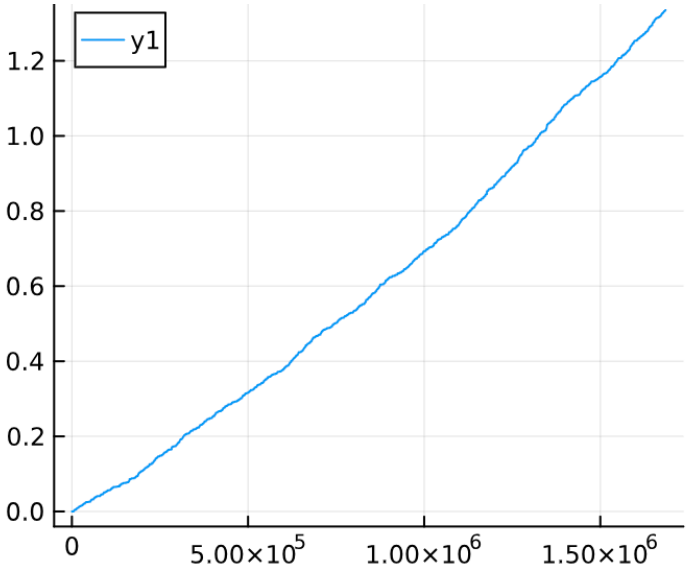
So let's actually test 39 jobs and see what happens. Here we have 1'685'279 times that a path is added. We now assess the *density* per 1000 times in the array to assess if there is such an increasing trend (which was our wary point 1). **Specifically we are pushing the "push data" step.**

First, density only. If greater than 0 (i.e. greater than 1e-5), increment.
Unfortunately there is no clear trend in density. Maybe 10k (right) instead of 1k (left)?



Same problem, 10k bin size also yields no clear pattern in terms of density.
Maybe cumulatively? (With and without outliers)





No, looks very slightly superlinear.

So I conclude that our methods for optimization are either (or a combination of):

1. **Overthrow Bellman-Ford-inspired LSA altogether and find a new algorithm.** Fundamentally, we've reached our limits with this implementation because of what I said in conclusion 1: our operations are already as simple as can be, with a language (Julia) that optimizes for simplicity ('looks like Python, speed like C').

2. **Don't overthrow Bellman-Ford, but look for existing heuristics to avoid traversing the loops so many times.** There do exist traveling salesman heuristics: for instance, if we have a "criss-cross" like shoelaces, untangling the shoelaces will shorten distance. This is a simple operation that can be done without resorting to extreme loop tactics.

3. **Fine-tune/nitpick on our existing LSA and find little things we can cut time from.** Since many of the operations within each time testing loop have the same complexity (e.g. pushing to an array), perhaps it would help us if we can reduce as much array-pushing etc. as possible. Pushing to arrays is particularly costly out of our operations; simple fetches are simple. *Testing the efficiency of individual maneuvers is conducted by visiting [4] The Playground and running an operation ad nauseam, seeing how it takes for Julia to complete.*