Time-Checking DF, SP, and CG *with optimality gap analysis*

This document will combine all three big approaches we use for solving the project. I will test their time complexities.

DF stands for Direct Formulation, SP stands for Set Partitioning without Column Generation, and CG stands for Set Partitioning with Column Generation.

Note: This analysis was written before I resolve what's going on with fact that *n*-best-routes CG performs better than normal CG (they should theoretically do the same thing), and update my LSA to include a better route tree prune mechanism. I did the time-checking now so I would immediately have an idea of runtime. Furthermore, since DF and SP are set in stone, getting their runtimes now would help me benchmark how my CG should perform.
Note: I have deleted n-dom check from all LSA's, but before I performed pruning.

Aspects to Check
The creation of parameters themselves that are not specific to any implementation (parameters, locations, time windows, job loads, and distance matrix) are not included in the time checks. Anything after that, including helper functions, are counted.

For DF, pre-processing is building the node network and arc system, and initializing helper functions find_node and find_arc. Model-building is the constraints etc. Optimization is hitting optimize!.

For SP, pre-processing is creating the entire route system and getting the valid ones through BF, as well as helper functions for getting parameters. Model-building is setting up the model and constraints. Optimization is hitting optimize!.

For CG, pre-processing is creating the initial set of routes and computing parameters and helper routes. Model-building is everything except the optimization that's inside the loop or part of initial constraints (this is the best way to describe it because CG, unlike the other two, needs the route results from last time to count so it can run). Optimization is hitting optimize! (Note that optimization will happen many times.)

Results and Analysis:
We vary the number of n_jobs from 10 to 39, our magic number when doing CG.
This is before we did any pruning, which I'll do. Once I do that, I'll update time trials.

Direct Formulation:
We measure pre-processing, model-building, and optimization directly. After 29, the machine took so long that I decided not to continue. It had already hit 10 minutes.

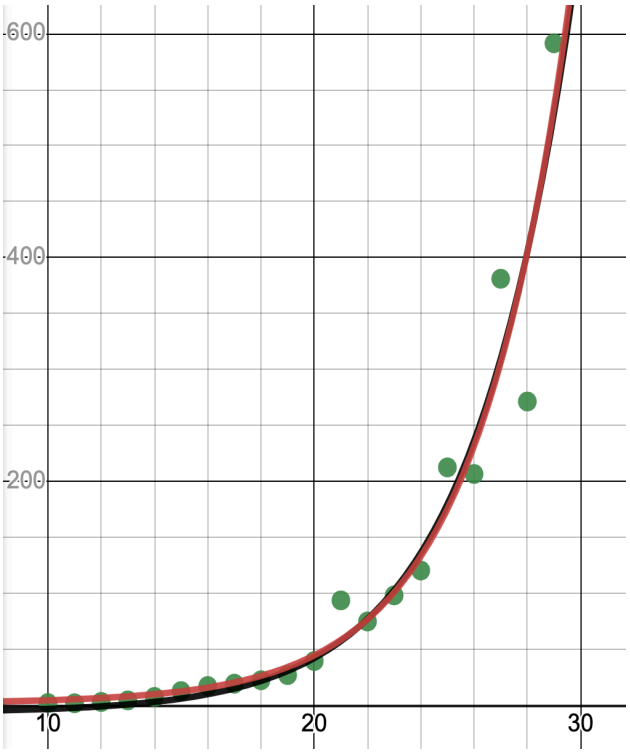
n	pp	mb	op	total	obj
10	0.144	1.718	0.541	2.403	2'327
11	0.139	1.274	0.770	2.183	2'727
12	0.148	2.002	1.233	3.383	2'795
13	0.163	2.963	1.638	4.764	3'070
14	0.226	4.996	2.634	7.856	3'124
15	0.245	8.270	4.539	13.054	2'643
16	0.248	11.906	5.443	17.597	2'677
17	0.296	14.891	4.463	19.650	3'127
18	0.306	19.777	2.645	22.728	3'135
19	0.346	24.113	2.923	27.382	3'163
20	0.400	35.694	3.814	39.908	3'156
21	0.471	45.860	47.728	94.059	3'402
22	0.553	65.123	9.524	75.200	3'512
23	0.692	77.813	20.017	98.522	3'548
24	0.744	99.951	20.044	120.739	3'601
25	0.982	137.175	74.615	212.772	3'023
26	1.138	181.870	23.881	206.889	3'043
27	1.356	210.169	169.681	381.206	3'060
28	1.444	241.422	28.721	271.587	3'135
29	1.604	331.056	258.774	591.434	3'206

Routes are stored in the appendix.

Analysis of DF Results:
Clearly DF is the most disappointing time-wise of the algorithms, but it does provide the exact correct answer. Both the model building (declaration of objectives, constraints, and

variables) and the optimizing itself take significant time and can be considered the bottlenecks of the algorithm.

The pre-processing and model-building steps appear to have an exponential (k to the x) or geometric (x to the k) relationship. (Exponential is worse.) Furthermore, the time taken in total can be approximated by either. Both models have high R-squared numbers: the exponential ($y_1 \approx c(x_1 - a)^b + d$) having 0.9325 and the geometric ($y_1 \approx cb^{(x_1 - a)} + d$) having 0.9339. This can be seen in the Desmos graph below (I used Desmos because it was faster and more accessible wrt regression).

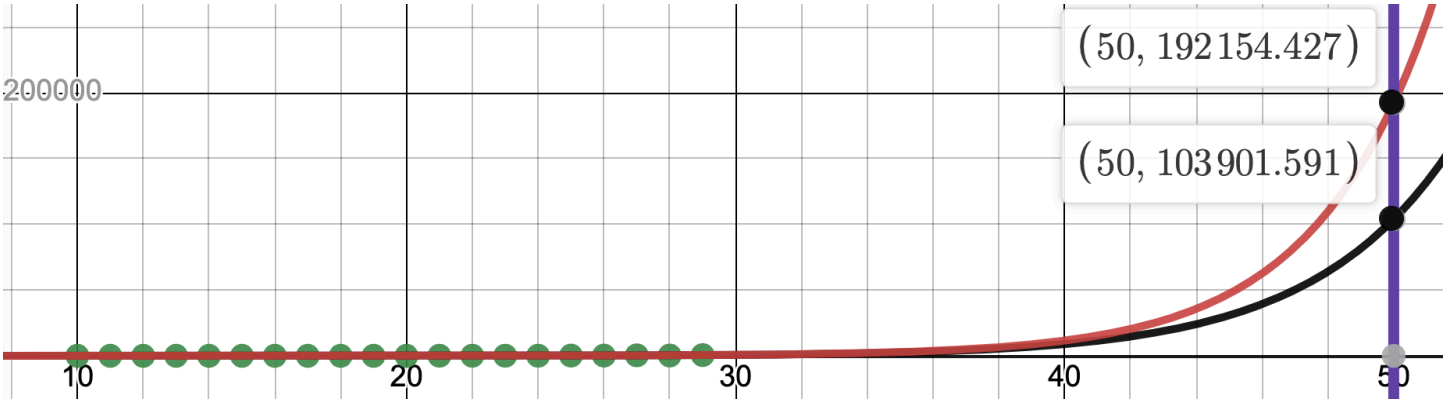


Here green dots represent the times taken, red is the exponential regression, and black is the geometric regression. Note how close they are here.

It must also be noted that changing n_vehicle_jobs (how many vehicles are available to be dispatched) does not appear to have much result on the time taken.

Each of the three relationships we analyze has its own meaning. The pre-processing and model-building steps have a consistent upward trend, whereas Gurobi might fluctuate (note 47 seconds to solve 21 but only 9 to solve 22). This is unstable and, depending on the data itself, might vary significantly. The fluctuations can occur when the addition of a window and load might cause slight variations in how exponentially/geometrically, if the pattern is such, the time taken scales itself in proportion to the number of arcs that must be built. But the pattern is easily proven to be stable.

If we take this model literally, going to 50 jobs (which took 2 hours and 22 minutes based on the initial ICT presentation sent to me back in January 2023) will burn a staggering 104k seconds by the geometric regression and 192k seconds by the exponential regression. This is displayed below. That's 29h and 53h respectively!



Set Partitioning:
We measure pre-processing, model-building, and optimization directly. The magic number this time is 35. This took a woeful 10 minutes to run, so I'm not going higher.

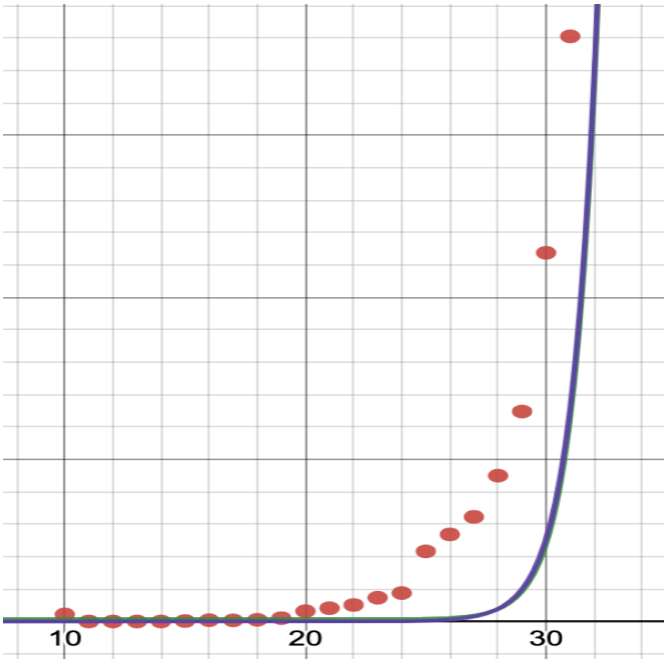
n	pp	mb	op	total	obj
10	0.268	0.171	0.005	0.444	2'262
11	0.001	0.001	0.006	0.008	2'658
12	0.001	0.002	0.007	0.01	2'717
13	0.001	0.003	0.007	0.011	2'987
14	0.001	0.003	0.009	0.013	3'035
15	0.005	0.011	0.024	0.04	2'539
16	0.008	0.045	0.040	0.093	2'566
17	0.010	0.023	0.050	0.083	3'012
18	0.011	0.031	0.073	0.115	3'014
19	0.018	0.064	0.132	0.214	3'036
20	0.070	0.197	0.381	0.648	3'028
21	0.106	0.210	0.515	0.831	3'257
22	0.084	0.322	0.625	1.031	3'364
23	0.144	0.510	0.823	1.477	3'399

24	0.136	0.633	0.994	1.763	3'445
25	0.365	1.395	2.578	4.338	2'845
26	0.645	1.742	2.994	5.381	2'856
27	0.403	2.654	3.406	6.463	2'867
28	0.600	3.570	4.834	9.004	2'933
29	0.858	5.000	7.104	12.962	3'003
30	2.078	9.152	11.523	22.753	3'282
31	2.685	13.768	19.652	36.105	3'284
32	6.215	37.212	48.343	91.770	3'292
33	8.384	44.961	65.297	118.642	3'406
34	12.810	59.892	89.183	161.885	3'649
35	35.285	180.607	336.332	582.224	3'314

Results and Analysis:
Performance is better than in DF, but still suboptimal. The magic number in this case is 35, after which it takes too long. It is worth mentioning that optimizing this algorithm appears to have great consistency in all three subprocedures, whereas the DF only exhibited consistency in the pre-processing and model-building, but not the model-solving. This may do with the fact that the problem here is far simpler, with only 'bare-bones' constraints and timing constraints already baked into which routes are being taken. Specifically, recall that our subproblem looks like the following:

DECISION VARIABLES - z_k^q (whether route q is used by vehicle k)
OBJECTIVE - MIN $\sum_k \sum_q z_k^q C^q$
CONSTRAINT - $\sum_k \sum_q z_k^q u_i^q = 1 \ \forall i$ (each job is covered by 1 route)
CONSTRAINT - $\sum_q z_k^q = 1 \ \forall k$ (each vehicle takes exactly 1 route, which may be trivial)
That's literally it. Time constraints take care of themselves.

Sadly, the models that worked so well with DF have extreme trouble adapting to the specific curve formed by these times. Geometric and exponential look like the graph below...very disappointing. Maybe we can extrapolate the DF curve at 44 (because this one is kinda like the last one delayed by 6) - that would give estimates of 7h and 11h.



Column Generation:

We measure pre-processing directly. However, since the entire loop has swallowed the model-building and optimization steps, we count the initial model parameters outside the loop as part of model-building, and everything that's inside the loop but not the actual optimize! call model-building as well. The optimization step is the only thing that goes inside optimization. Because the time involved depends on the number of times we iterate in the loop, and I don't want pushing a time array to accidentally boost up the true model runtime, I will time like this:

time_mb_outside_loop_start
model parameters initialize
time_mb_outside_loop_end
enter the loop
    time_mb_before_opt_start
        processes happen
    time_mb_before_opt_end
    time_opt_start
        optimize!
    time_opt_end
    time_mb_after_opt_start
        more processes happen
    time_mb_after_opt_end
loop ends

As can be seen, the reason I don't just time mb when I enter the loop and before it ends, and then subtract the opt time, is because I don't want the time_opt_start and time_opt_end commands, which involve pushing to arrays, to interfere with the time itself. Remember when I was timing the LSA that the number of iterations through the loops, which reached the millions, forced push!(time_check...) to literally account for 52% of the time on one check!

I will still have 3 arrays: pp, mb, and op. But I will structure the last two differently. For mb, I will push a new number after parameters initialize and before entering the loop, and then I will increment the last element in that array while we are in the loop. For op, I will also do this: push a 0 right before we enter the loop and update as necessary. This way I don't have to deal with nested arrays or different lengths.

Also, I am going to turn the column generation on n_jobs routes considered at a time instead of 1. For reasons which require investigation, using n=1 somehow terminates early and fails to get the optimal solution, which is infuriating. There must be something wrong with the colgen equation! I will go see what's wrong after I finish time trials here, and write the pruning algorithm.

n	pp	mb	op	total	obj
10	0.176	1.091	0.003	1.27	2'262
11	0.001	0.006	0.004	0.011	2'658
12	0	0.005	0.003	0.008	2'717
13	0.001	0.005	0.005	0.011	2'987
14	0	0.008	0.024	0.032	3'035
15	0.001	0.021	0.011	0.033	2'539
16	0.001	0.029	0.012	0.042	2'566
17	0	0.060	0.015	0.075	3'012
18	0	0.062	0.019	0.081	3'014
19	0.001	0.069	0.018	0.088	3'036
20	0.001	0.088	0.023	0.112	3'028
21	0.001	0.129	0.030	0.16	3'257
22	0.001	0.144	0.033	0.178	3'364
23	0.001	0.165	0.036	0.202	3'399

24	0.001	0.258	0.047	0.306	3'445
25	0.001	0.425	0.060	0.486	2'845
26	0.001	0.548	0.071	0.62	2'856
27	0.001	0.399	0.047	0.447	**2'905**
28	0.001	0.571	0.062	0.634	**2'958**
29	0.001	0.646	0.054	0.701	**3'154**
30	0.001	0.942	0.049	0.992	**3'531**
31	0.001	2.264	0.094	2.359	**3'284**
32	0.001	5.198	0.104	5.303	**3'292**
33	0.001	6.316	0.119	6.436	**3'406**
34	0.001	5.196	0.083	5.28	**3'736**
35	0.001	26.349	0.190	26.54	**3'365**
36	0.161	16.213	0.087	16.461	4'564
37	0.002	27.047	0.142	27.191	4'534
38	0.001	51.485	0.276	51.762	4'012
39	0.001	32.821	0.119	32.941	5'156
40	0.165	130.238	0.181	130.584	4'672
41	0.176	419.111	0.339	419.626	4'611

Results and Analysis:
The most concerning note is that the colgen, as it stands, cannot obtain the optimal value. This is concerning because it means something is wrong with the colgen algorithm and has to be fixed. I ran this using n_jobs as the number of new routes to take in each time, so I am very concerned. Bold red means it is not optimal (compared with SP); non-bold means that only CG was able to generate this large of a number of jobs, but it's almost guaranteed based on this pattern that it will not be optimal. (Interestingly, for n=27, 28, and 29, the upper limits of DF, CG had a better optimum than DF, though it's not known whether this pattern would persist with n=30, for instance.)

Otherwise, the results analyzed suggest that pre-processing is a painless procedure. Indeed, it barely seems to take any time at all (except the first time we do it, in which case it mysteriously needs a chunk of time longer than 1 millisecond - probably Julia's

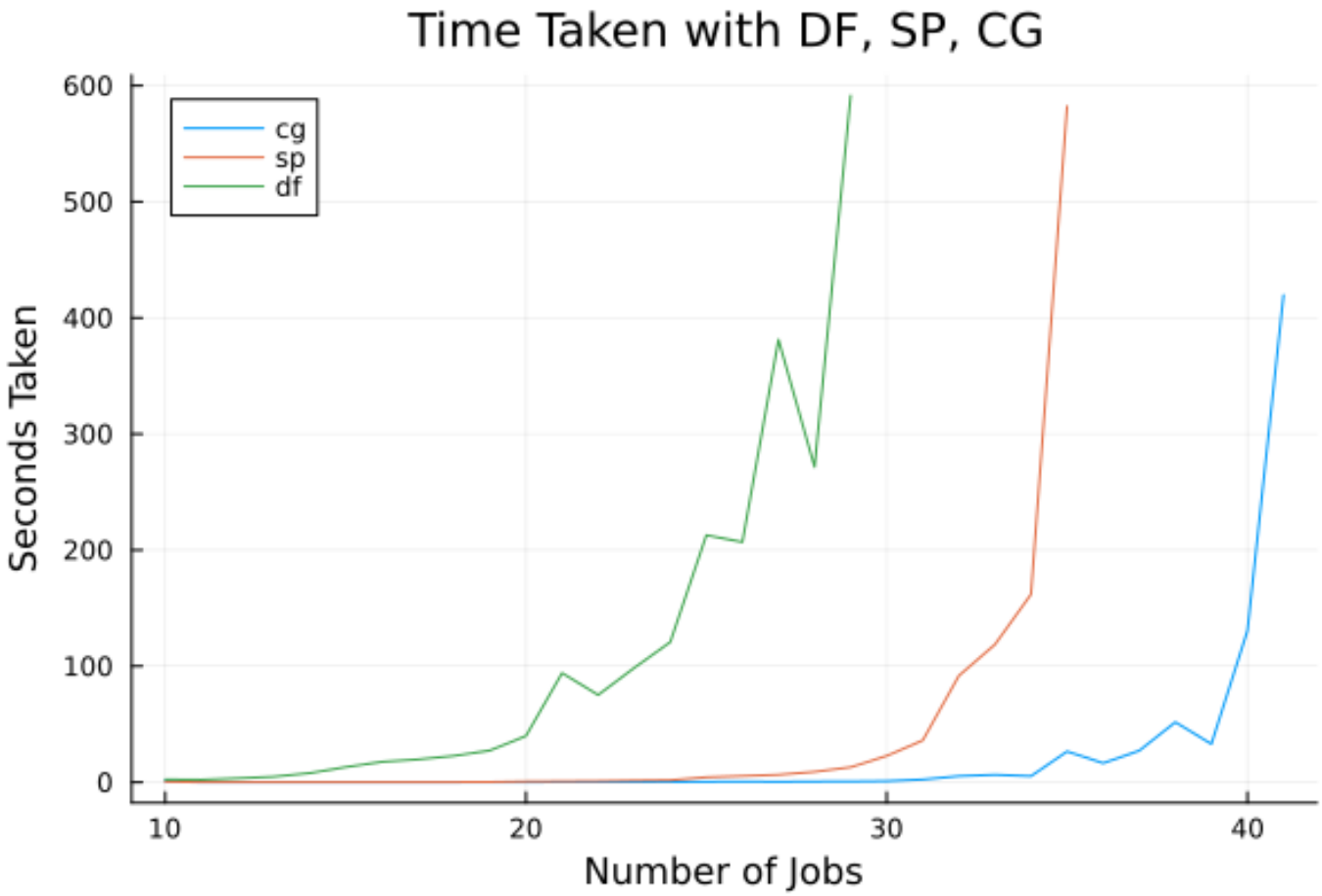
warming up or something). This is in direct contrast to the other two algorithms, for which large systems of routes or rcs must be built.

The optimization itself is mostly painless, though time does increase as we go on. After all, the number of routes is so small for now that it won't cause much harm.

The bottleneck is easily the LSA, which consumes the most time. The goal now is to find a way to prune it.

The magic number is 41, which is 6 higher than the SP stopping point of 35, but still a long way until 50.

The time taken by all 3 can be summarized in this plot.



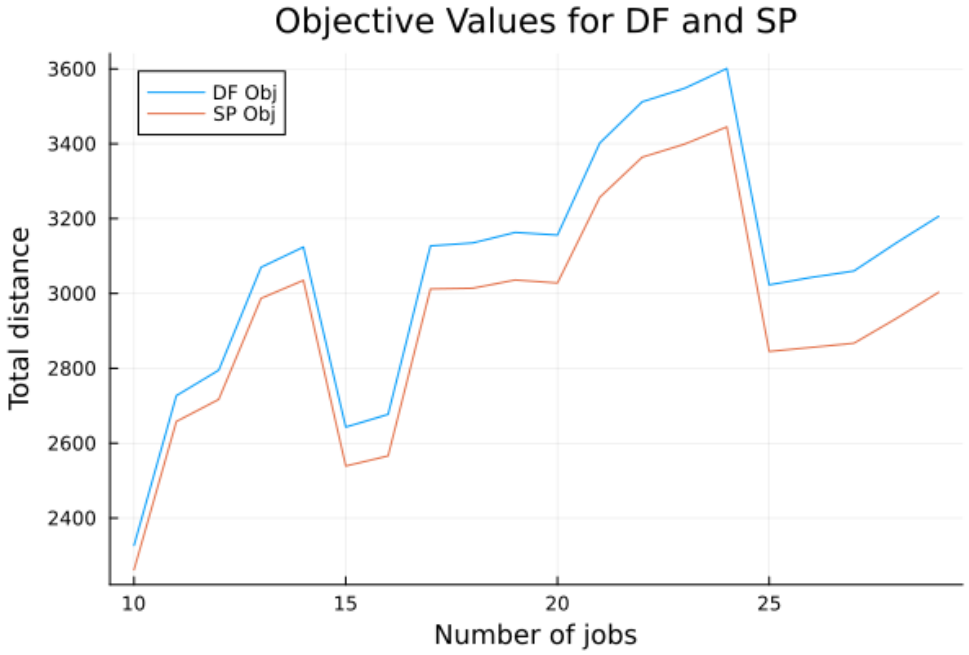
Optimality Gap Analysis

(Note: I will use the terms "optimality gap" and "objective gap" interchangeably.)

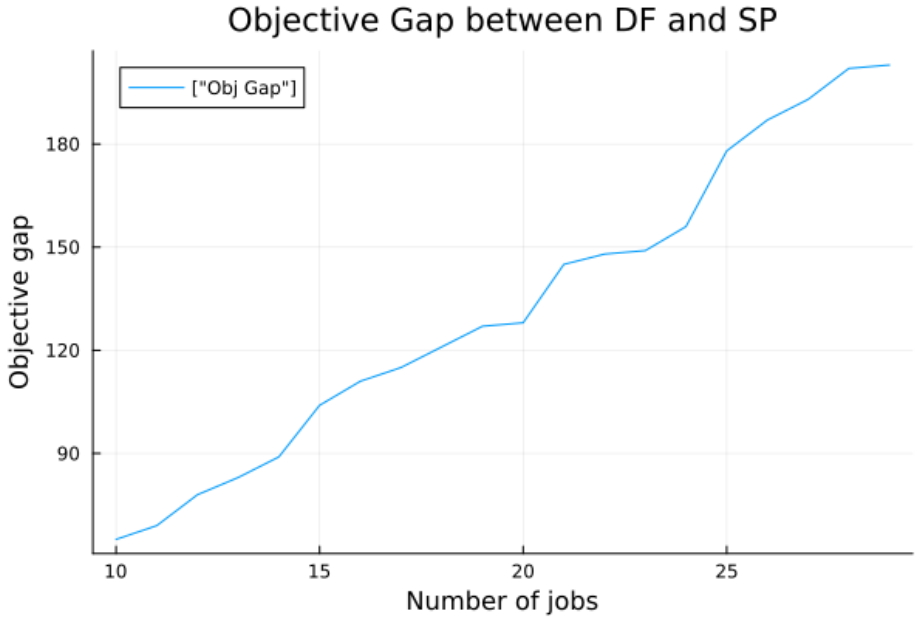
Note that the CG objective is exactly the same as the SP objective (at least until it breaks down starting at n=27), while the DF objective is usually mildly higher (which is

supposed to happen because the CG is a relaxation). It is also true that the routes produced by the DF, SP, and CG are exactly the same - this is supposed to happen.

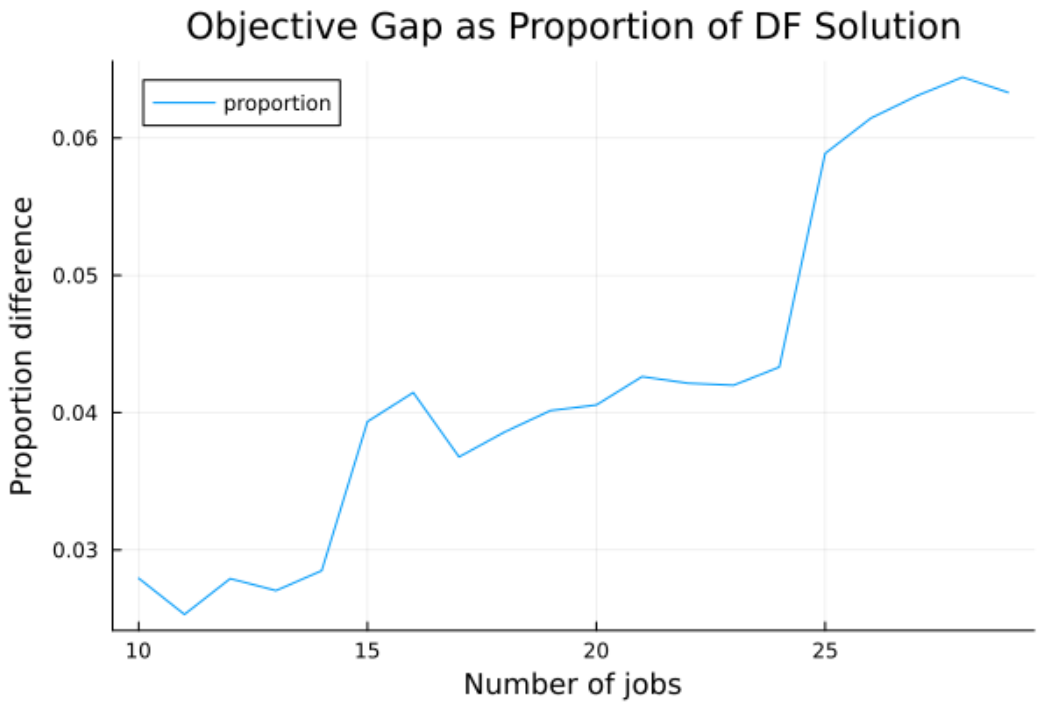
Let's see how 'bad' the optimality gap is. We plot the DF objectives from n=10 to n=29, while we plot the SP objectives from n=10 to n=29.



Also, we can plot the raw objective gap, and this appears to have a linear trend. In fact, the line y = 8x - 20 would do nicely.



This could be cause for concern as we increase the number of jobs, especially since plotting the proportional difference (% error, so to speak) gives us an even worse result. We are breaking 6% difference when we hit 29 jobs.



See the notebook "optimality-gap-plotting.ipynb" for the plots.

Meanwhile, see the below page for an appendix on routes.

Appendix

Routes:

As produced by SP and DF up to 29 inclusive. These answers are guaranteed correct.

10	7,8; 5,6; 1,2; 3,4; 9,10
11	5,6; 7,2; 9,10; 1,11,8; 3,4
12	11,12; 3,4; 5,6; 1,2; 7,8; 9,10
13	7,8; 9,10; 5,6; 13; 1,2; 3,4; 11,12
14	11,12; 13,14; 7,8; 5,6; 9,10; 1,2; 3,4
15	7,8,9; 1,2,3; 10,11,12; 13,14,15; 4,5,6
16	7,8,9; 13,14,15; 16,4,5,6; 10,11,12; 1,2,3
17	7,8,9; 10,11,12; 4,5,6; 13,14,15; 16,17; 1,2,3
18	13,14,15; 10,11,12; 1,2,3; 4,5,6; 16,17,18; 7,8,9
19	16,17,18; 13,14,15; 4,5,6; 10,11,12; 19,7,8,9; 1,2,3
20	4,19,5,20; 18,11,12,13,14; 15,16,17,6,7; 1,2,3; 8,9,10
21	21,11,12,13,14; 8,9,10; 18,19,20; 15,16,17; 4,5,6,7; 1,2,3
22	8,9,10; 21,22,3; 4,19,5,6,7; 1,2,13,14; 15,16,17; 18,11,12,20
23	21,22,23,3; 1,2,13,14; 15,16,17; 4,19,5,6,7; 18,11,12,20; 8,9,10
24	18,11,12,13,14; 15,16,17,6,7; 1,2,3; 4,19,5,20; 21,22,23,24; 8,9,10
25	16,17,18,19,13,14; 1,2,3; 24,25; 20,21,22,23; 4,5,6,7; 8,9,10; 11,12,15
26	24,25,26; 11,12,15; 20,21,22,23; 4,5,6,7; 16,17,18,19,13,14; 8,9,10; 1,2,3
27	16,17,18,19,13,14; 24,25,26,27; 1,2,3; 20,21,22,23; 11,12,15; 8,9,10; 4,5,6,7
28	20,21,22,23; 11,12,15; 24,25,26,27,28; 4,5,6,7; 1,2,3; 16,17,18,19,13,14; 8,9,10
29	16,17,18,19,13,14; 24,25,26,27,28; 1,29,2,3; 8,9,10; 20,21,22,23; 4,5,6,7; 11,12,15
30	13,14,15,16,17,18; 5,6,7,8,9; 28,29; 23,24,25,26; 1,2,3,4; 27,10,11,12,30; 19,20,21,22
31	27,10,11,12,30,31; 1,2,3,4; 23,24,25,26; 13,14,15,16,17,18; 28,29; 5,6,7,8,9; 19,20,21,22
32	5,6,7,8,9; 27,10,11,12,30,32; 13,14,15,16,17,18; 1,2,3,4; 19,20,21,22; 28,29,31; 23,24,25,26
33	27,10,11,12,30,32; 19,20,21,22; 33,28,29,31; 5,6,7,8,9; 1,2,3,4; 23,24,25,26; 13,14,15,16,17,18
34	23,5,24,25,26; 19,20,21,22; 27,10,11,12,30,32; 1,2,3,4; 28,29,31; 33,34,6,7,8,9; 13,14,15,16,17,18
35	5,6,7,8,9; 25,26,27; 10,11,12,13; 1,2,3,4; 21,22,23,24,28; 14,15,16,17,18,19,20; 29,30,31,32,33,34,35