# Bottleneck Analysis of mcg TH

## Section 1. Time Testing Full MCG TH

Our most promising algorithm for solving jobs and coverage is the mcg TH. It solves 50 jobs in a total of 3m 51s. Let's break down the algorithm and see what improvable components we identify. Trivial things like pushing an element once per loop are not included.

1. Jobs-only algorithm (this is a black box - it has been optimized from previous work)
2. dcg algorithm
    1. Set up and optimize RMP
    2. Clip windows by when job work is done for coverage SP
    3. Clip windows by when coverage is available for job SP
    4. Perform job SP
        1. Time feasibility
        2. Prunability
        3. Push new route
    5. end job SP
    6. Perform cov SP
        1. Return to depot: add
        2. Otherwise:
            1. Get golden intervals (this should not be anything)
            2. For each interval:
                1. Check time
                2. Check prunability
                3. Push new route
            3. end interval loop
        3. end loop for spot *j*
    7. End cov SP
    8. Extract and potentially add job/cov routes
3. end dcg algorithm

We write two separate timing mechanisms to gather the following.

**Mechanism 1:** Time 1, 2.1, 2.2, 2.3, 2.4, 2.6, 2.8. This will give us a rough bottleneck spotting and allow us to immediately eliminate possibilities.

In the next mechanism, see if 2.4 or 2.6 escalate in time (these are the only ones likely to do so). If they do, test them individually.
**Mechanism 2:** Time 2.4.1, 2.4.2, 2.4.3, 2.6.1, 2.6.2.1, 2.6.2.2.1, 2.6.2.2.2, 2.6.2.2.3.

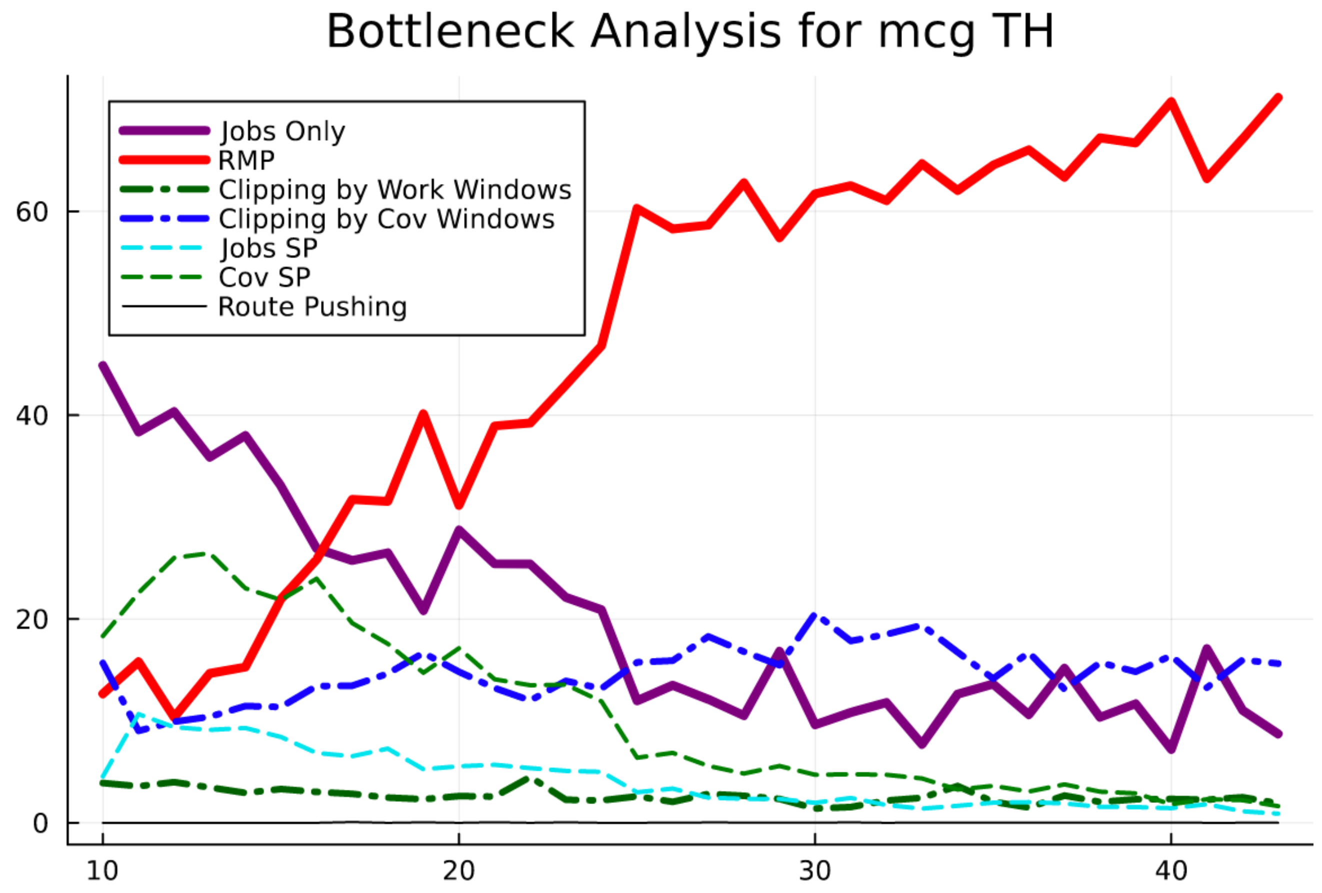
The length of 2.4.1 will give us the total number of routes considered (the new node is not within nodes we've seen). Length of 2.4.2 is routes we would've used had we not included the pruning check, and length of 2.4.3 is the number of routes that passed our checks and added.

The length of 2.6.1 is how many depot-bound routes we add. 2.6.2.1, the number of times we get golden intervals. (This should be near 0 for each one.) 2.6.2.2.1, the number of routes we will consider. 2.6.2.2.2, the number of routes we would consider without a pruning mechanism. And finally, 2.6.2.2.3, how many new routes we add.

We will also test whether the pruning mechanism should have < or < + 1e-8. We start with the former.

**Mechanism 1 Test Results**
We go from 10-43. Our time elements are:
black_box_t (jobs-only); rmp_t (RMP); clip_by_job_t (work windows);
clip_by_cov_t (clip by coverage); job_sp_t (jobs SP); cov_sp_t (cov SP); add_route_t.

Let's see which element makes up the biggest portion of time spent.



As can be seen, by far the biggest time-burner is actually not anything I've written, but the Gurobi RMP! In second place, shockingly, is the clipping by coverage windows—pruning time windows to match up with when coverage is available. And the third bottleneck is the initial time spent on jobs-only optimization. The last one is non-negotiable and there is no optimization to be done there*.

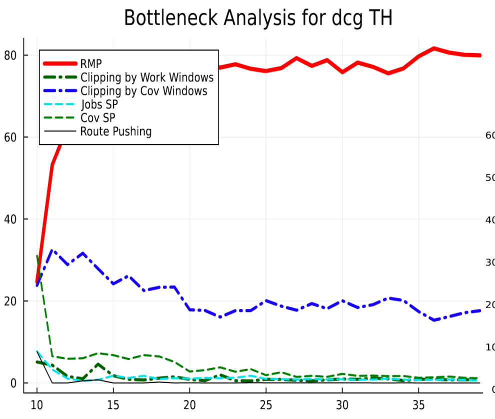
As for the SP algorithms, they are not only very well optimized, even when they are repeatedly called, they don't make up that much of the time! We consider this problem solved. Route pushing does nothing to the time, as expected.

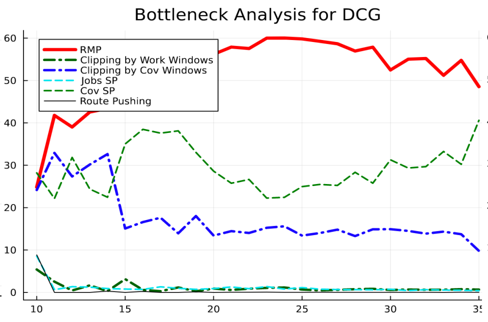
*What if, instead of running BF every single time on a dual-based shortest paths weighting system, we designed an algorithm that would get the reduced cost given sufficiently small perturbations on a graph based on the previous result? Could this happen? Might be a good theoretical CS/algorithms question but not as relevant here, especially since we have optimized to a remarkable extent.

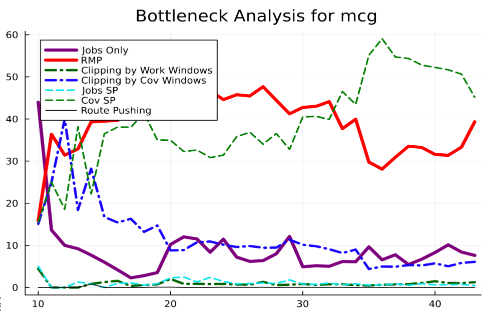
The two tasks ahead of us are:

1. **Optimize clipping of coverage windows**
2. **Reduce Gurobi's long time taken to solve the RMP**

Let's put this graph together with the other 3 algorithms' to see if we yield any additional insight. Perhaps another graph will enlighten us. In order: dcg TH; mcg; dcg.







The graphs are similar in a few respects. Gurobi, with its black box RMP, is a main burden on computation. Jobs SP, route pushing, and clipping by work windows have almost no impact on time spent. Most importantly, the contrast between TH and non-TH algorithms is testament to how powerful the treasure-hunting method is. For dcg and mcg, a whole 40% of time is burned on the coverage SP without treasure-hunting; for their TH-included methods, this drops to 2%!

All four graphs have clipping by cov. windows as a significant burden, hitting 10%-20%.

## Section 2. Time Testing Clip by Coverage

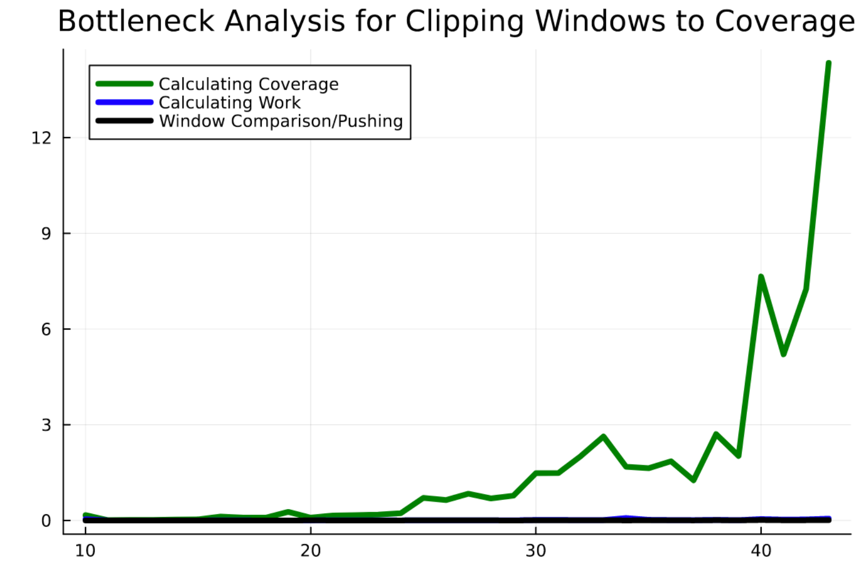
We run a bottleneck analysis specifically on clipping by coverage windows to see if we can gain any insight.

**The Clip by Coverage Algorithm**

```
1. Enter for loop
    1. For each TIME window (when work can be done, not work window):
        1. For each time:
            1. Obtain total coverage ($\sum_j \sum_p y_{jt}^p x^p$, where $j$ is in close_covs($i$))
            2. Obtain work coverage ($\sum_q z^q \delta_{it}^q$)
            3. Compare total coverage to work coverage (or if total cov >= 1), push.
        2. Push if appropriate
    2. Exit time window loop
2. End function
```

We put time clamps around 1.1.1.1, 1.1.1.2, and 1.1.1.3. The lengths of the three should be the same and will tell us how many times we entered the loop.

Here is what we get. We've run the loop from 10 to 43.



The true bottleneck is calculating coverage, which takes an excruciatingly long time.

This is the line

```
total_cov = sum(sum(y[j][t][p] * x[p] for j in close_covs(i)) for p in 1:P)
```

which is represented as $\sum_{j \in C(i)} \sum_{p} y_{jt}^{p} x^{p}$.

Let's see if we can get rid of anything here. No parameters have to be calculated; they are just brought in as parameters of the function signature (especially y, which we calculated beforehand).

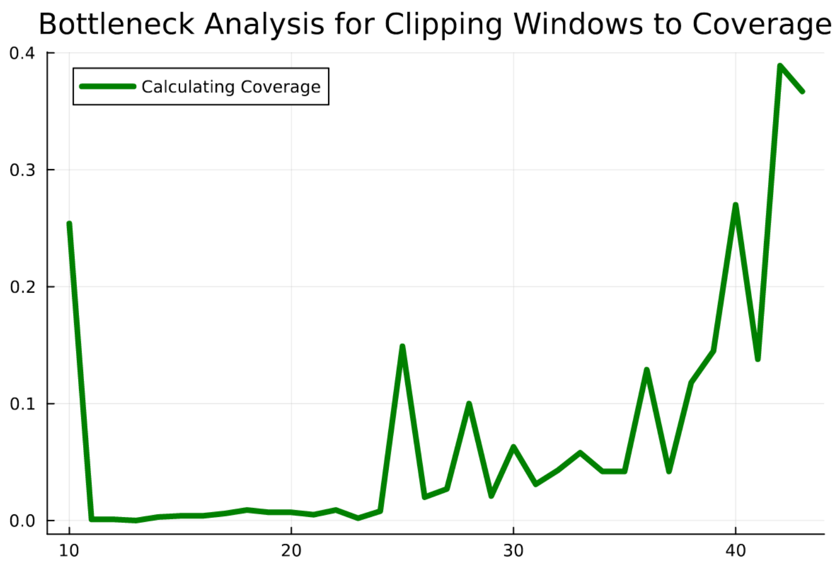
Ok, there is one thing I spot immediately.

Notice that I am forced to trudge the loop many times, basically the coverage might look like 0...1...1...0. Why should I have to sit through the middle? Let's see if we can reduce time by pointing from the start inward until coverage is 1, at which point that is our start point, and taking another pointer separately from the end back towards the beginning until coverage is 1. For example, suppose the coverage provided is [0, 0, 1, 1, 1, 1, 1, 0]. Then instead of going through all 8 elements, I only go through 5 of them because I start: index 1 is 0, index 2 is 0, index 3 is 1—there's the start index. Now, index 8 is 0, index 7 is 1—there's the end index, so my window is 3 through 7.

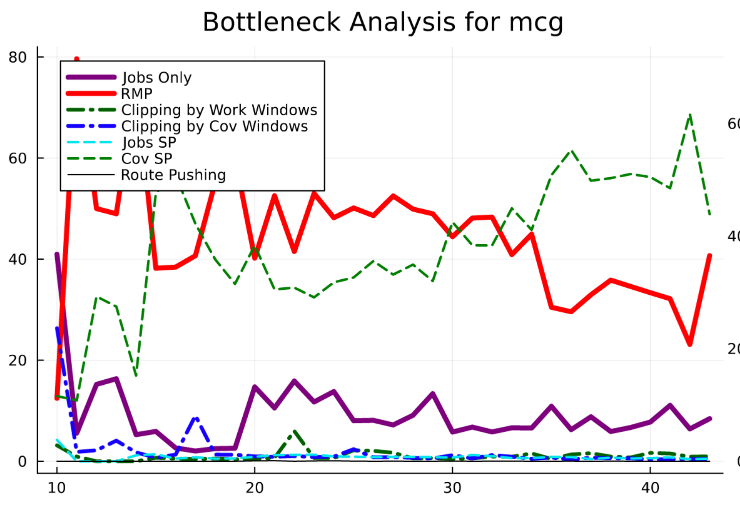
(It's not necessarily going to be 1 - that was for ease of illustration. The true requirement is that coverage >= work done OR coverage >= 1, because we might have fractional solutions in the middle.)

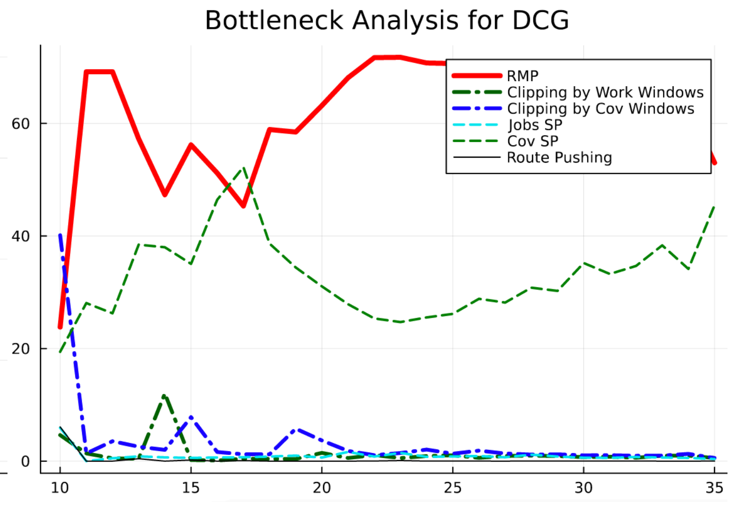
Another little thing: Don't calculate close_covs(i) every single time. When you start the loop for a certain i, calculate it right away: "cc = close_covs(i)".

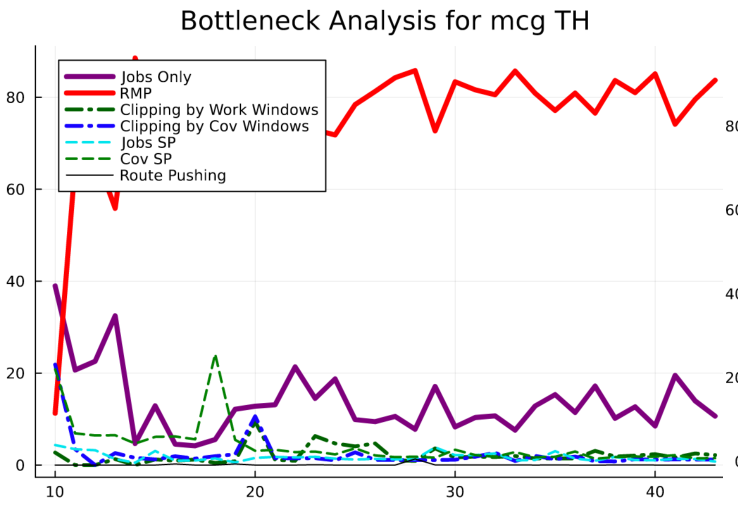
Here is a plot of the new time taken for the new coverage check. Notice how little time the new version takes. The top is 0.4 seconds, not 12 seconds. Bang! The bottleneck is GONE! We retest the 4 two-stage algorithms to find bottlenecks.

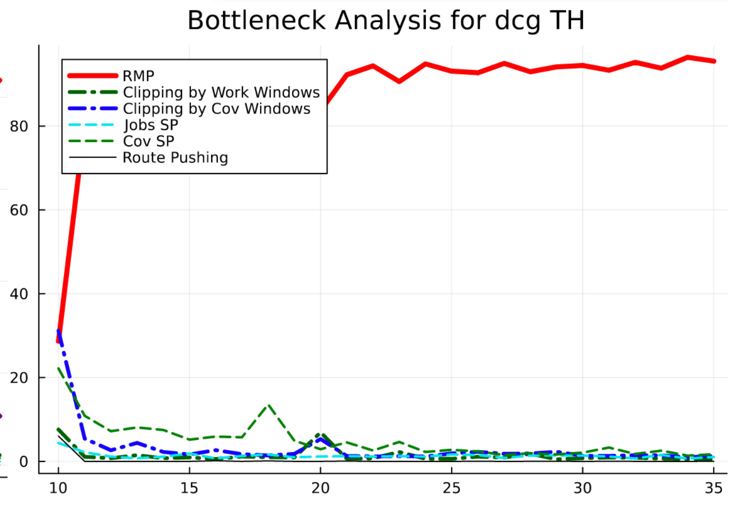


The new graphs look like this.









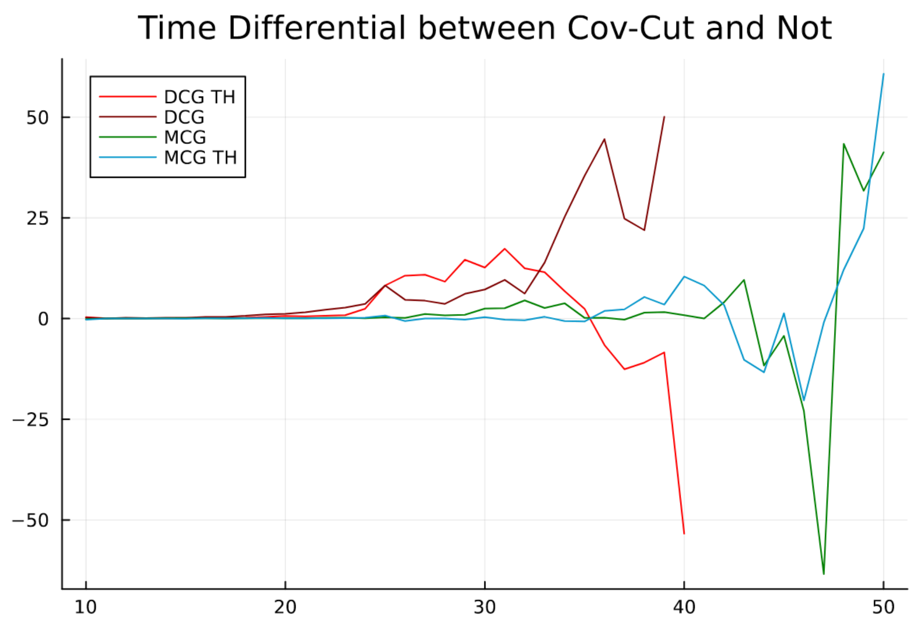
Notice the total disappearance of "Clipping by Cov Windows", dash-dot in thick blue. It is no longer an issue at all! The only bottleneck now is the RMP itself. (And once again, the difference between non-TH and TH methods proves the importance of treasure-hunting.)

## Section 3. Overall Timing and Correctness

The new method, which I will "cov-cut", is usually an improvement, but not always. We plot the differential in time (positive = good: cov-cut is more efficient than normal) and objective (positive = good: cov-cut has a better objective than normal).

Actually the objective plot is all zeroes. The objectives are exactly the same, indicating the cov-cut was written correctly.

As for time, most scenarios indicate that cov-cut does better. Its reduction in time can be as big as a minute, usually at the 20-second mark or so. But it could also just as well elongate the time necessary by a minute.



The plot analyzes all four scenarios, but the one we care about is MCG TH, because it is the fastest one. This blue line is mostly positive.

The good news is that the bottlenecks are now 'out of our control' and must be addressed at the RMP level. The only way to improve the algorithm further is to find ways to reduce the number of times we have to iterate the RMP.

Improving the RMP will involve theoretical considerations and potential snipping with mathematical proofs.