The Pruning Mechanism: Description and Runtime

Description

Let's start from the algorithm we already have, found in "FINAL-cg.ipynb", which stripes the ndc and represents a simple Bellman-Ford. The description of that algorithm is:

- Initialize Parameters
- Enter the While Loop
  - Check if we want to move onto the next current_state (which means: If we've already hit the depot, don't keep adding; if we aren't at the depot, keep adding)
  - Enter the For Loop: survey nodes
    - Check that the current node is not already in the path
      - → Before we add the node: if this is not a 1-element array (i.e. if we are not at the depot and have already done at least one job), check that we can actually go there and do work so we're not wasting time. If we pass the check, go down.
      - Add the new path to our list N.
      - Increment total_state by 1 (indicating number of routes added).
    - This is the end of the passed-check.
  - This is the end of the for loop.
  - Add one to current_state.
  - Check that current_state has not caught up to total state yet. Otherwise, leave the while loop.
- Exit the while loop.

Initially, we did have an ndc placed at the end of the within-while-loop which checked: If your route is still feasible and we go through the same node list, ending up at the same place (not necessarily the depot) (and implicitly starting at the same place, the depot), then we might have to replace a route if it has too low a cost.

However, we are able to do better. A new pruning mechanism is as follows: Each node itself will receive a reduced-cost label (so we need an array of size n_jobs: one number for each node). At any point, if the reduced cost of a route that would go to a certain node is not great enough to defeat the label already present, don't add that route.

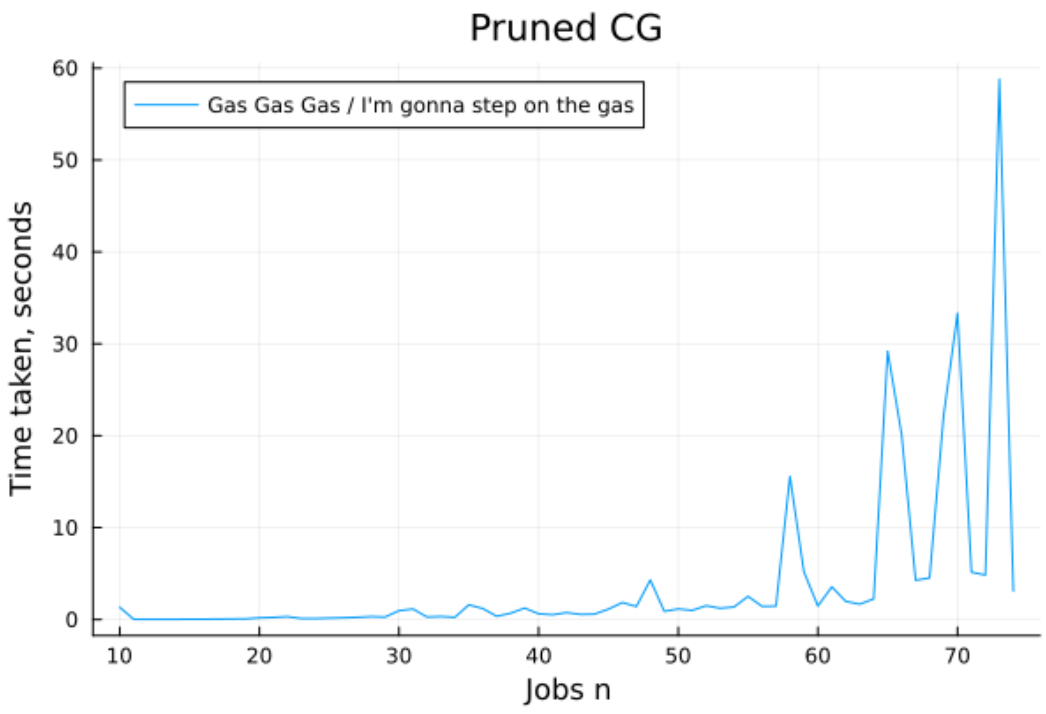
Example: Suppose node 15 were assigned -1'500. We have a route stub 0 → 13 → 20, with reduced cost -1'100, and the remaining segment 20 → 15 would take the route to a reduced cost of -1'300. But since the proposed new route stub 0 → 13 → 20 → 15 has reduced cost -1'300 > -1'500 which is the assignment, we don't follow through on this stub. Instead, we keep going and looking that way.

How would we implement this? See where I included a small arrow in the algorithm? We perform a check that is like this, but after this first check (so it is on the same "tab level"/"indentation level" as this arrowed check). Specifically, we see if R[current_state] plus the three changes (add_segdist, subtract_pi, subtract_mu) is more negative than the label at the node right now. If it is, do the update. If not, continue.

And how do we get our list of node reduced costs? First, initialize it at the top as NRC. Then, every time (and only every time) we plan to add a path, update NRC along with the other variables.

Runtime
The algorithm is devastating. It literally blazed through 50 jobs like it was nothing, at a staggering **1.466 seconds**.

In fact, the plot of time taken versus jobs is even more amazing.



A few questions remained to be answered.

1. Why are there random spikes at n = 58, 65, 70, and 73? And why are there valleys?
2. When attempting n = 75, why does Julia report an out-of-bounds error (tried locating index 4'454 when the list is only 4'453 elements long) that makes no sense (I'm currently at index 3'962)? I'll give more details soon.

The magic number right now is **74**. And if not for a mysterious error that defies reason, we'd probably be at 120 or something by now.