Subproblem Label-Setting Algorithm - 1d Node List / 2d Location Map

This is going to read similarly to our previous label-setting algorithm, except with different parameters.

Recall that we are calculating $C^q - \sum_{i \in \mathcal{I}} \pi_i u_i^q - \rho - \sum_{i \in \mathcal{I}} \sum_{t \in \mathcal{T}} \mu_{it} \delta_{it}^q$ and this is the cost of an entire route. Cleverly we segment it into a shortest-paths (i.e. minimum-weight) formulation, which is Bellman-Ford, which is nearly identical to what we did before!

Parameters

#1 A matrix of distances. This is identical to the one we used for our first LSA. We will need this because this is also exactly the cost matrix (cost is distance between nodes)!

#2 A matrix of travel times. This is also identical to what we have before. Time and cost are distinct, but this will tell us whether we should get to a node, and at what time.

#3 A list of work windows. Like the previous algorithm, we need to know when work can be performed to check feasibility. Don't forget to tack on a [0, 100] to guarantee feasibility of returning back to the depot!

#4 A list of work time amounts. How long does it take to complete each job? Don't forget to tack on a 0 at the end to guarantee feasibility of returning to the depot!

#5 The numerical value of dual variable ρ. This is a single number corresponding to the dual of the equation $\sum_{q \in \mathcal{Q}} z^q \geq -K$. Each route will take a subtraction of ρ.

#6 The numerical list of dual variable π_i. This is another dual variable which is the dual of the equation $\sum_{q \in \mathcal{Q}} u_i^q z^q = 1 \ \forall i \in \mathcal{I}$. This will be coupled with u_i^q BUT that's guaranteed to be 1 (because if we even consider π_i of course $u_i^q = 1$ so no worries).

#7 The matrix of dual variable μ_{it}. Corresponds to the dual of the equation $\sum_{q \in \mathcal{Q}} z_q \delta_{it}^q - y_{it}^S + y_{it}^E \geq 0 \ \forall i \in \mathcal{I},\ t \in \mathcal{T}$. This will be coupled with δ_{it}^q; don't need $\delta_{it}^q = 1$.

Thus, #1, #5-#7 are for calculating the reduced cost. #2-#4 for feasibility.

Meaning of and Strategy on the Reduced Cost

This is where we get our clever idea. Observe that computing the above-notated calculation would be quite difficult if we used a standard algorithm and asked it to minimize over all routes. Instead, we take a graph of all routes and use label-setting to traverse it and return us a route with the most negative reduced cost. This reduced cost is calculated by running Bellman-Ford over the locations graph.

In particular, we start by initializing with the trivial route [1]. To make calculations easier, since all future routes will be based off this route, we just say it has a starting reduced cost of $-\rho$. This is mathematically correct because $C^q = 0$ (we don't go anywhere), $\sum \pi_i u_i^q$ is 0 (because π is only defined for i values equal to the job locations, not the depot), and $\sum \mu\delta$ is also 0 (because μ is only defined for i values equal to the job locations, not the depot).

Similar to our first LSA, this one will have a cost array which corresponds not to distance only, but also to the additional parameters.

Then, whenever we hit a new node (and we will have a feasibility check), we do the following:
- Increase the cost C^q by the distance from the last node to the new node
- Subtract $\pi_i u_i^q$ where i corresponds to the NEW node index ($u_i^q = 1$)
- (do nothing about the rho)
- Subtract $\mu_{it}\delta_{it}^q$ where i corresponds to the NEW node index and t corresponds to the NEW time which represents when we first hit new node i ($\delta_{it}^q = 1$)

I have a question. I believe it is correct that we would only have to subtract $\mu_{it}\delta_{it}^q$ ONCE, namely, when we enter the node i (i.e. we go from not being at i to being at i), rather than for every single time t at which we are physically at node i. For example, if we are at node number 46 for times 34, 35, 36, and 37, then I would only subtract $\mu_{46,34}\delta_{46,34}^q$ rather than $\mu_{46,34}\delta_{46,34}^q$ and ... and $\mu_{46,37}\delta_{46,37}^q$. *Is my interpretation correct?* I am trying to find a reason to convince myself either way and I'm coming up short.

The Algorithm

We initialize a set of variables which will help us. Also our first route is [1].

L is a list which keeps the last node of the current route. It turns out to be helpful. It is initialized to [1].

N is a list of lists = routes. It is initialized to [[1]].

R is a list of REDUCED costs! (Yes this was called C earlier.) It is initialized to $-\rho$.

T is a list of times. Each individual time expresses when we get to the final node so far. It is initialized to 0.

A is a list of feasibilities: is the current route still worth considering? This is only repealed if it turns out our planned route got dominated by some other route. It is initialized to true.

Like with last time, we also have a *current_state* and *total_state*.

Step 1 First, check that we still have valid paths to go. Same as before.

Step 2 Enter the for loop.

Step 2-1 Check that the current node is not already in our path.

Step 2-2 Feasibility check. This is exactly identical to what we had before. The only real feasibility check remains regarding time, so there's nothing new.

Step 2-3 Same clever tactic as last time.

First, update (or change) your new node and push to the new/existing total_state + 1 index of N.

Then, update the time and push/change to T.

Then, updated the reduced cost. The only difference is the cost will be different. Do this to list R. This time, it will be:
- ADD distances[last_node_so_far, new_node]
- SUBTRACT pi[new_node] (because u is guaranteed to be 1)
- do nothing about rho because that was already factored in earlier
- SUBTRACT mu[new_node, NEW_time_of_arrival]

Then, add "true" to list A.

And finally, update the new node to be i: update L.

Step 2-4 Run the final check to protect against n-domination. Same idea as before, no change.

After this is finished, return all variables. Specifically, return N, T, A, R.

WATCH OUT. We will need to pad pi and mu like we did for load and windows when we enter the algorithm. This is because if we actually hit i = n+2 meaning we want to go back to the depot, mu[n+2-1 = n+1] doesn't actually exist and will throw an error. So when we are changing the cost, we only do the -pi_v[i-1] if i is not equal to n+2, otherwise, we better do a zero.