Building the Label-Setting Algorithm

Parameters

***#1 A matrix of distances*.** The row represents the location you're starting FROM and the column represents the location you're going TO. The ath row and bth column equals how far it is from place a to place b.

In our specific case, we have the possibility of starting FROM any of locations 0 through n_jobs (the one depot, plus the locations for the jobs), while we can go TO locations 0 through n_jobs + 1 (because we will need to end up back at the depot post-time). We'd also like our matrix to be symmetric. Thus we will end up with a square symmetric array of size (n_jobs + 2) x (n_jobs + 2).

To create this matrix, we take the distances we've created very early on in the array and deepcopy it. Then, add (hcat) the first entire column (representing distances from anywhere into the depot at start) to the end of the deepcopied version. Because of symmetry, technically this is equivalent to hcatting the first row with 0 pushed onto it. Next, vcat a collected adjoint of the column vector composed of the first column with 0 pushed into it onto the now-asymmetric distances matrix. This will give us the desired result. Technically this is equivalent to vcatting the last column with 0 pushed into it.

Example:

0 2 3
2 0 8
3 8 0

First, hcat the first column (in this case first row is also mathematically correct):

0 2 3 0
2 0 8 2
3 8 0 3

Then, vcat the first column with 0 pushed onto it (last column also fine):
0 2 3 0
2 0 8 2
3 8 0 3
0 2 3 0

This is called *distances_label*.

The code to do this is:
```
distances_label = hcat(distances_label, distances_label[:, 1]);
distances_label = vcat(distances_label, collect(push!(distances_label[:, 1], 0)'));
```

#2 A matrix of time to travel. Same concept as above, except we measure the duration or length in TIME to get from point a to point b. But this is pretty simple: take the previous array, and divide by the maximum speed, then ceiling the whole thing.

The code to do this is:
travel_times = ceil.(distances_label / speed);

#3 A list of times to work. For each job, what's the window of time for which we can work? This will be a list with n_jobs items. Each item is a tuple. In our label-setting algorithm we will also beware the need for an additional filler element. I don't fully understand this yet, but I will once we implement the algorithm.

The code to do this is:
push!(windows_label, [0, 100])
Note that [0, 100] is a filler which will obviously be feasible at all times.

#4 A list of work time amounts. For each job, how much time will we need to spend doing it? This will also be a list with n_jobs items. Each item, unlike the previous parameter, is a single element.

The code to do this is:
push!(load_label, 0)
Note that 0 is also a filler which is always doable no matter what.

The Algorithm

We initialize a set of variables to help us with the algorithms.

Start with the trivial path [1].

The strategy is as follows: Start with the trivial path. Then, at each time, point an arrow to the current path (i.e. state) we seek to improve upon. If we can improve upon it, do so. Also include a pointer to the total number of paths.

At each iteration, we only add at most one location to the path. We go through all possible nodes each time to determine whether we should add it or not.

So when we enter the loop, we are poised to select a current state and improve upon it. We keep working on that state, unless that state already has hit the depot at the end, meaning we can't add to it, or the path itself is actually infeasible, in which case we move on. If this is the case, then we move on to the next current state, so we increase the iteration by 1.

Step 1 At any point we must check that we haven't overshot the entire array of valid paths. If this is the case, then exit the entire algorithm because we're done and there's nothing else to check. We can return our variables. Otherwise, although we incremented our current state by 1, we keep going and enter the for loop.

Step 2 Enter the for loop.

Step 2-1 Check that the node is not already in our path, because going back to visit a node number is not allowed. If the node is new, proceed.

Step 2-2 Feasibility check. Make sure that we actually have enough time to go from the last node of the current path to that next node. To do this, obtain the current end-time of the current path. Then add the amount of time it takes to do the current job. Then, add the time it takes to get from the last current node to the new node. Then, make sure the amount of time it takes to complete the new job added to this quantity is within the bounds of the job; specifically, make sure we can end by that time. The code to do this is: make sure that

current_time + travel_time + new_job_time_need + old_job_time_need
<= new_window_end_time.

If this is satisfied, do nothing and keep going. If this is not satisfied, get back to Step 2 and go to the next new node, which is done by using continue.

Step 2-3 We see if we can add any node to our current path, which is denoted by N[current_state]. Note that this will lag far behind the actual frontier of paths we have created, but this is necessary.

From a clever time-saving trick we will use later, either our array of paths is of length total_state + 1, or it is of length total_state.

2-3-1 If Possibility 1 If it is of length total_state, that means there is no invalid path sketched out in front of us, in which case we need to add a new path.

This new path will first be the same thing as a copy (deepcopy not necessary because we have immutable values) of the state we are trying to improve.

Because the new node has passed the feasibility checks (not already in list, and time-workable), add it immediately. In other words, add path to the list N.

Update Time: Either this is when we get to the node, or when the window time starts for us to do the job. Add this to the list T.

Update Cost: Add the distance traveled to the list C.

Update Last Node: Add the new node i to the list L.

Update Feasibility: Add "true" to the list A.

This concludes the first possibility.

2-3-2 If Possibility 2 There was an invalid path created before (which we intentionally don't remove to conserve time in the algorithm).

Do the same procedures as we did for 2-3-1, but change the values rather than push any values.

Step 2-4 This is the final check: We make sure that our current path is the best possible cost. This is equivalent to Bellman-Ford when we check that the distance we have on the path is actually the best distance we can achieve.

Here, we test the newly-generated path at index total_state + 1 against all old feasible paths. In particular, we see if the nodes they visit are exactly the same, regardless of order—except that they must end at the same place! If it turns out the cost of this new path is greater than some feasible path with the same nodes, then this newest path is wasted and we don't increment total_state; instead, we prepare to overwrite it. The intelligence of this method is that we avoid having to scrub or delete something, which is bad complexity-wise. Instead, just leave it there as something bad which we will overwrite.

If instead the cost of the old path is too high compared to the new one, the old path becomes useless. In this case, turn the feasibility of the old path to false and the new one is still as it is.

The name for a path which is better in cost than others with the same location is "dominated", so we'd hope our new paths dominate the old ones i.e. are dominant.

But there appears to be a problem with this approach. What happens if we end up finishing on something that gets dominated, so that we inadvertently finish with a path which is dominated by some other path? In other words, it turns out the variable NDom is false.

Finally, append etc. variables as necessary. In particular, increment the current state by one. If it turns out that the current state has now hit the total state (meaning everything we had to generate has been checked), stop the algorithm.

Keep going. Then return all variables.

We return N (the list of paths consulted), A (whether that path is optimal/feasible), C (cost of that path), and T (when the path ENTERS its last location).

Note: T is a list of ints, not a list of lists. It only tells us when we enter our LAST location. This is to save computational power. If we wanted to draw a graph of paths etc., I would go back into this algorithm and re-modify T to be a list of lists instead, showing when we enter EACH location!

Variables

L is a list which keeps the last node that is currently included in the route. When a route is being considered, only that index of L will be modified. Eventually it will equal the highest-number node (latest position) which the route will visit.

L is initialized to 1.

N is a list of lists, i.e. a list of routes, in which elements of the list are nodes. We do not guarantee that each route is feasible!

N is initialized to [1].

C is a list of costs: how much total cost is incurred when we go through this route?

C is initialized to 0.

T is a list of times: When do we GET to the final node?

T is initialized to 0.

A is a list of feasibilities: Is the current route feasible? This can only be repealed if it turns out the path got dominated.

A is initialized to true.

current_state refers to the pointer of which path number we are currently expanding upon and iterating through the nodes for. It might create several new paths, which will increment total_state, but it won't affect which arrow path we are currently looking at. If we generate a lot of new paths based off this state, then they will have be reckoned with later. For most of the time current_state will lag desperately behind total_state, until reaching the end depot starts catching up.

total_state represents how many paths have been created. All of these eventually must be verified. This quantity will race out in front against current_state.

We can mathematically prove that the number of valid paths (i.e. paths which represent a feasible combination that start from the depot and end at the depot) is exactly twice the number of total paths generated, i.e. the final value of total_state.