Code for Colgen on Jobs and Coverage

1 - Set Up the Parameters

Initialize B and V (number of job vehicles and number of coverage vehicles). Initialize n_jobs and m_cov (number of job locations/nodes and coverage spots; the latter will be changed when we hit step 2). Time frame (1 to T is allowable for jobs, 0 and T+1 are reserved for exclusive use of the depot), min and max duration of jobs, speed of travel, coverage radius, size of the square, and coverage mesh size.

2 - Create Job Parameters

Create job locations, job time windows, and job work load.

Also create a (n_jobs + 2) x (n_jobs + 2) matrix of distances, and one for travel times. The first and last indices both refer to the depot and have the same constitution.

3 - Create Coverage Parameters

Create coverage spots. There are two methodologies.

One - create all valid coverage spots. If a potential coverage spot, determine by the coverage mesh, is close enough to at least one job, include it.

Two - arrange a sequence of spots and only add the next spot if a previously-uncovered job would be within range.

Just like the job case, make the coverage array also include the depot start and end.

4 - Create Initial Job Routes

Make dummy routes: 1 for each job that goes from the depot to that job and back. Have the route stay there for the entire time window.

5 - Create Initial Coverage Routes

Make dummy routes: 1 for each coverage spot that goes from the depot to that spot, stays there until the last possible time, and then comes back.

6 - Write Helper Functions

1. Compute the cost of a route (gets C^q and C^p)
2. Compute u_i^q (if route q passes through job i)
3. Compute δ_{it}^q (if route q passes through job i at time t)
4. Compute y_{jt}^p (if route p is at spot j at time i)

5. Compute L_{ji} (if spot j can cover node i)
6. Compute the neighborhood of a spot j
7. Compute the valid coverage stations of a node i
8. Given route, list the nodes
9. Given route, list the times
10. Given nodes and times, produce the route

7 - Write Shortest Paths Algorithm for Job Routes

8 - Write Shortest Paths Algorithm for Coverage Routes

9 - Write Restricted Master Problem

10 - Extract Dual Variables, Update Parameters, Do 9 Again
This method should be similar for q and p.

Variable Bank
n_jobs - number of jobs.
m_cov - number of coverage spots.
B - number of job vehicles.
V - number of coverage vehicles.
T - time units. 0 and T+1 are depot only; 1 through T are for jobs/coverage.
min_duration - minimum work load for a job.
max_duration - maximum work load for a job.
speed - how fast vehicles travel.
radius - coverage radius.
size - how long the coverage square is on each side.
mesh - how fine our coverage mesh will be.

time_windows - the n_jobs tuples of when jobs can start to be done and must end.
work_load - how much time it takes to do each of n_jobs jobs.
job_locs - a n_jobs + 2 length list of where depot, n_jobs jobs, and depot are.
job_dists - a (n_jobs + 2) x (n_jobs + 2) matrix of how long, distance, it takes to get from a node to another node. Remember to add 1 to the true location to index properly.
job_travel_times - a (n_jobs + 2) x (n_jobs + 2) matrix of how long, time, it takes to get from a node to another node.

cov_locs - a m_cov + 2 length list of where depot, m_cov spots, and depot are.
cov_dists - a (m_cov + 2) x (m_cov + 2) matrix of how long, distance, it takes to get from a spot to another spot. Remember to add 1 to the true location to index properly.

cov_travel_times - a (m_cov + 2) x (m_cov + 2) matrix of how long, time, it takes to ge from a spot to another spot.

job_routes - the job routes for iteration in column generation.
cov_routes - the coverage routes for iteration in column generation.

C_jobs - route cost for jobs.
C_covs - route cost for coverage.
u - u[i][q].
delta - delta[i][t][q].
y - y[j][t][p].
L - L[j][i].

Q - length of job routes.
P - length of coverage routes.

Things to Test:
- how much of a time and answer difference the two methodologies for creating job parameters gives
- whether the following methods of adding new columns makes a difference:
  - new route q; run; new route p; run
  - new route p; run; new route q; run
  - new route q, new route p; run
  - successive new route q-run pairs
  - successive new route p-run pairs

Upgrades:
- Instead of letting T be set initially and not touched again, have a check after creating job routes: Have a list of time_windows[i][2] plus distance back. Set T to be the max value. (I know you can technically do T-1 as well.)
- Include tree-pruning for both job and coverage shortest paths subproblem routes.

Writing 7 - Shortest Paths for Job Routes
We adopt much the same algorithm as we did for PT NyVA. I'm going to make sure it works without pruning before I do it with pruning.

Take a set of distances, travel times, windows, and work loads. Include dual variables ρ, π, and ξ.

We are going to minimize $C^q + \rho - \sum_i u_i^q \pi_i + \sum_i \sum_t \delta_{it}^q \xi_{it}$. While we traverse in shortest paths, we don't care about u_i^q and δ_{it}^q because they are trivially 1.

Initialize an empty set for paths, reduced costs, and times. Start with a trivial path that goes nowhere (only a single 0), a reduced cost of rho (because all routes will build on this so don't change it), and a time of 0. Remember that we are going to measure both departure and arrival times so it becomes easier to know when we're where. In other words, job nodes will now have two node times associated with them. For example:
nodes 0 4 8 26
times 0 3 8 10 16 17
means we start at the depot at time 0, take 3 time to travel to job 4, work at job 4 for 5 hours, leave to get to job 8 between times 8 and 10, work at job 8 for 6 hours, and take 1 hour to return to depot.

Initialize also curp and endp as pointers.

Enter a while loop and indicate a for loop for how long we are traveling where.

First, decide whether the current path is time-feasible. In particular, take the last element in times (this will be our departure time from the old node), add the travel time to the new node, and add the job load at that new node. This must be less than or equal to the end of the time window at that new node. If this is true, set the arrival time at the new node to be the maximum of that summation I just mentioned and the opening of the time window (the second component is important because we don't want to arrive too early).*

Next, calculate the reduced cost. In particular, set the departure time as the arrival time plus the job load.* For all times from the arrival to departure time, add a positive xi variable.

If pruning: Test the total reduced cost against benchmark.

Then, add the route's new node, the arrival and departure times (two values), and the new reduced cost.

* In the jobs-only formulation we could get by because time was irrelevant. As long as you worked within the time window, no matter when you did it, cost was still optimized (subject to, of course, that you could actually arrive at that node). This principle allowed us to scrap the y variables and accelerate our algorithm bigly. Here, we might

have to consider time because of the pathological example I gave in my document "DF Coverage Formulation after Jobs Solved" on page 3. Also, we may have to optimize "collecting" negative (or lack of positive) xi values in our job routes! But for now, let's make the formulation work, then I'll think about this afterwards. Hopefully this "window-shifting" isn't too bad, as I think the number of cases is usually pretty limited.

Writing 8 - Shortest Paths for Coverage Routes

This is far more painful than the above because we can return to nodes, we need to account for times, etc. Nevertheless, there are opportunities for decreasing the time spent solving this.

First, I claim that we only ever depart a coverage node if there is a "job escape" in the neighborhood. In particular, if at any point, we are not leaving while a job in the neighborhood is going on. Otherwise, we are changing horses in the middle of the river and unnecessarily adding to our cost. If we leave in the middle, there will have to be two vehicles in the same location, which will obviously be unideal by the Triangle Inequality.

Note that if a job concludes at time t* and another starts at time t*, we canNOT leave at time t*! But if the gap is 1, we can leave. For example, if we see that the jobs in the neighborhoods are [2, 8], [3, 9], and [10, 12], this is NOT the same as if jobs are [2, 8], [3, 11], and [10, 12]. In the former, we are allowed to leave right after 9 so that a new vehicle can come at 10 if hypothetically necessary. But in the latter, there is no leaving at time 9 because we will switch horses in the middle of the river.

Interesting point: We can do this either on the time windows altogether, or just delta itq. The latter is more complicated, so we'll do time windows first.

To find "job escapes", we start from t=1. Consider t and t+1. If any job uses both those times, then we cannot escape; if no job uses both times, we can escape, and specifically, t will be our escape time.

After collecting all escape times, we kick off any that occur after the last job ends. For example, if the last job in the neighborhood ends at 36, and our escape times are 5, 7, 8, 14, 22, 33, 36, 37, 38, 39, ..., then only use 5, 7, 8, 14, 22, 33, 36. There is no point in wasting time covering nothing; by deleting such scenarios we do not remove viable solutions. This helps bring down complexity, though it still hurts having to consider all of these possibilities.

Also, the constraint that we can't visit a node twice is no longer viable.

Finally, consider the dual variables we have. Note that they are all nonnegative by the way we formatted the problem. But only ξ_{it} can possibly bring down the cost. We observe empirically that ξ_{it} is very sparsely populated. As a result, we should go xi-hunting. How can we turn this into a constraint? Well...I'll worry about that in a bit.

We will definitely prune this one. Can we use the same pruning mechanism? Yes, even though you can visit the same node multiple times. Why? Because if you go back to the same node after visiting at least one other and you don't reduce the total cost, a better solution will exist: this one, but without that 'loop' viewed in 2d.

So what are our arrival and departure times? Now there's no time you're constrained to arrive or depart, so if you're leaving for a node, you might as well arrive at the earliest possible time. (Make a note to check out: What if we solved the coverage shortest paths as a job shortest paths problem, but with multiple time windows and the ability to return? Could we constrain arrival to only when the next job starts?)

Also, if all jobs have already ended, don't go to a spot. That wastes time.

So we keep an array of times to indicate when all jobs have indicated at each spot so we don't call the function over and over again.

Let's see where that gets us.

Notable Debug Hunts
The difference between a vector and a matrix in .- subtraction caused coverage distances to be far higher than they should, resulting in an unsolvable problem.