Treasure Hunting for ξ_{it}

Section 1: Designing a New Algorithm

It is clear that most of the time from our dcg is spent on the column generation algorithm. Our magic number for the mcg approach is 43 because so many iterations of sp cov are necessary. This document describes a better way to solve sp cov.

Observation 1: Very rarely is $\xi_{it} > 0$.

We want our reduced cost to dip below zero, which relies on supremely negative ξ_{it} because C^p is so positive. But if we investigate a few cases, we see that there are very few values for which this is actually true. It is not hard to see that any route of negative cost corresponds to the xi's being positive, and we have to hit at least one 'treasure' (i,t) node for which $\xi_{it} > 0$; otherwise, the reduced cost, formed by $C^p + \beta - \sum \xi_{it}$ is surely positive. For instance, in n = 22, there are only 11 such eggs; for n = 25, this is 10, and for n = 28, there are 9 eggs.

We can rewrite our algorithm as follows.

Based on job routes, we know exactly when each vehicle does a job when.

Because of a conceptual simplification I created that coverage vehicles can only enter or leave their spots at specified times, making the coverage path problem a "multi-time-window revisit-spot-allowed vehicle routing problem", we will build off this.

Specifically, instead of calculating arrival and depart times allowed, and iterating first through arrival times and then through depart time, we *iterate through intervals*.

The number of intervals governing any particular spot isn't very big (maybe 10 per spot). In other words, calculate all arrival times (at's) (done by function [6.12] When to Enter a Spot) and calculate all departure times (dt's) (done by function [6.13] When to Leave a Spot). Then, loop through all at's and dt's, creating intervals. This will be on the order of O(|at's| * |dt's|).

The intervals is a convenient shorthand for at, dt pairs. My plan is to enumerate all routes which include at least one easter-egg interval. and then use DP or some other technique to "build" the rest of the route around it.

Also, the integrity of when_to_enter and when_to_leave must be absolutely secure. If there is a hole in the function, our efforts will be doomed.

An example: Suppose our spot supports two jobs whose windows (it is based on the scenario: usually we'd like it to be work windows, not time windows, this is an example) are: [5, 7] and [9, 14].

Then the appropriate at's are 5 and 9, and dt's are 7 and 14. That's it. This yields three intervals: [5, 7], [5, 14], and [9, 14] ([9, 7] is invalid).

Another example: Let's say the pertinent windows are
[1, 4], [3, 5], [6, 10], [6, 14], [10, 16].

Then the appropriate at's are 1 and 6 (not allowed to enter while a job is being done, because another coverage vehicle will have covered and you will be wasting distance) while the appropriate dt's are 5 and 16.

So the intervals are: [1, 5], [1, 16], and [6, 16] ([6, 5] is invalid).

Observe that at's are a subset of window start times, while dt's are a subset of window leave times. This is intentional. As a bijection: if you wanted to leave a bit after a job has ended, why didn't you leave when the job ended? (Bijection is also the reason we can use pruning.)

when_to_enter is written as follows:

Given time windows. For each starting time of a time window, check that for all other time windows, we are not both > its start time and <= its end time. For example, if we propose 4 as an arrival time, then all other time windows cannot contain 4, or if they do, 4 must be the start time. In other words, if t > window1 and t <= window2, then t cannot be an arrival time. Otherwise, if this sandwich is not true for all other jobs, then t is an arrival time.

when_to_leave is written similarly:

Given time windows. For each ending time of a time window, check that for all other time windows, we are not both < its end time and >= its start time. For example, if we propose 4 as a depart time, all other time windows cannot contain 4, or if they do, 4 must be their end time. In other words, if t >= window1 and t < window2, then t cannot be a departure time. Otherwise, if the sandwich is not true for all other jobs, then t is a departure time.

We get a rather fast algorithm that isn't too bad compared to the job vehicles. But we can optimize further. Notice that when we're building our cov paths, we have to loop through all sets of at's and dt's, **and a lot of routes won't even incorporate a single golden egg!**

What if we tried a system that wasn't based on Bellman-Ford (or we can try to incorporate some of its components), but instead used a golden egg approach? In other words, given lots of intervals and locations, with some of them golden (indicating $\xi_{it} > 0$ somewhere in that interval), generate ALL paths with at least one golden interval and then calculate their reduced cost.

I'm willing to do this because the number of golden paths (def.: at least one golden interval inside) is far inferior to the number of paths overall, and the benefit of route pruning might be overwhelmed by the sheer low number of golden paths compared to the number of paths overall...maybe we can still implement BF, we'll see.

Writing the Golden Egg Path Algorithm (GEPA)

The main challenge with the gepa is to make sure it does not become exponential. Here is my first algorithm proposal.

Observe the following things:

1. We only want the single best route with the most negative reduced cost. Therefore, never pick any interval that isn't golden! (We will just be adding unnecessary cost without gaining any negative xi. And conceptually, note that if we eventually have to cover one of these normal intervals to reduce our cost, they eventually will become golden themselves!)
2. Among intervals of the same spot which have the same golden cost, pick only the smallest interval. This is for the following purposes:
    1. to reduce time complexity - if [4, 6], [4, 8], and [4, 9] all have the same golden xi reduced cost, just pick [4, 6]. Otherwise we waste time going over many routes with the same golden cost and it doesn't make a difference—why stay longer at the same node when you could stay shorter?
    2. to more efficiently identify the best routes - if we only have the shortest intervals, we can just go to the depot if there's no other golden intervals. BUT if we have the longer intervals, we might accidentally miss another golden interval whose distance cost isn't enough to overtake the negative xi that we could've hit had we used shorter intervals.
    3. it mathematically makes sense and won't cause us to miss good paths later on. Again, if we eventually do need to cover some spots, they will turn golden.

3. Suppose we have golden intervals with the same gold nuggets (nodes with positive xi). Note that there must exist a unique golden interval with the unique smallest timespan that is a subset of all other golden intervals with that golden nugget. This is proven through a bit of sets.
    1. Practically we can determine smallest interval and unique identity by looking at either golden nugget ID (i.e. what node gives that xi) or reduced cost. But there is a pathological case in which multiple golden nuggets have the same reduced cost!
    2. As a corollary to point 3, if two distinct golden nuggets happen to share the same reduced cost, their respective intervals must be entirely disjoint.

This suggests an algorithm that is close to the integer knapsack problem (time is like our limit on weight), but not really.

Gather All Valid Golden Intervals

Function: golden_intervals(xi, job_windows)

1. For each j:
    1. Obtain a list of all nuggets ξ_{it} where $i \in$ neighborhood(j).
    2. Obtain a list of allowed arrival and departure times for that j.
    3. Borrow the helper function nugget_interval(times, arrivals, departs) which does the following: Given a set of times (most of which will just contain one), produces the smallest interval [arrival, depart] which contains all those times. The times correspond to the times of the nuggets. (This is easy: Pick the latest arrival that is before or equal to the min of the times, and the earliest departure that is after or equal to the max of the times.)
    4. For all nonempty subsets of the power set of nuggets for that j, find the nugget_interval. These are exactly that valid golden intervals I reasoned through.
        1. *Note:* With G nuggets, we will have at most $2^{|G|} - 1$ intervals; it is possible that due to the lack of sufficiently 'granulated' arrival and departure times, some intervals intended to have only a smaller subset of nuggets will accidentally also contain more nuggets.
    5. All of these nugget_intervals should be stored into a list. Now use Julia's unique! to kill duplicates. (See 1.4.1 note for explanation.)
    6. For all these intervals, calculate the xi reduced cost.

Return: List of intervals for each spot. Each interval consists of: [[at, dt], rc].

The method I used above is far more efficient than listing all intervals from j and then seeing if they are golden. That requires us to loop over a lot of the possibilities, because they will just be normal (0 reduced cost; no nuggets). This just starts straight from where the golden intervals are and extracting the best ones.

After we have a list of golden intervals, we can run them through Bellman-Ford. The algorithm looks very similar to what we had before, but the number of possibilities is VASTLY reduced. We cannot use two for loops to simulate at and dt, however, because what if our valid intervals are [4, 8], [6, 10], and [4, 10], but NOT [6, 8]? I'd rather not waste time through these cases, and there's a lot of time to be wasted there. Instead, I'll iterate through intervals, get the at and dt associated, and do it that way. So this is actually a tiny tweak from the BF we already created.

Inefficiencies
Do not get the power set of nuggets. Instead, take down all times of nuggets, then find the O(G^2) possible single-elements or tuples of times. This is because a lot of the power set time intervals are going to be contained in bigger ones.

Accuracy Analysis
The treasure-hunting algorithm produces far better answers than the previous mcg algorithm using entirely Bellman-Ford on coverage vehicles.

Time Analysis
The treasure-hunting algorithm is also superior in time. Based on n_jobs = 50, we take a grand total of 238 seconds (3m 58s), with 174 seconds (2m 54s) to process the dcg step and 54 seconds on the jobs-only column generation.

The former method took 353 seconds (5m 53s).