RaoTA Final Report

Kevin An, Matthew Tran, Joe Zou

December 2019

Introduction

Our algorithm is the combination of multiple algorithms. We had a script that updated our outputs only if there was an improvement, which allowed us to easily try a variety of different ideas. Following are the three approaches that worked best and we spent the most time on.

1 Optimized TSP Algorithm

This algorithm uses the assumption that TSP run on only the houses and starting location will return a near-optimal ordering for dropping off TAs. We know that this may not be always true, but it makes some sense intuitively and it closely approximated brute force solutions on small graphs.
We start by approximating a TSP-solution through the homes and starting location to give us an order to drop-off TAs. We copy this initial ordering into an array for drop-off locations. Then we iterate over this drop-offs array and consider all other vertices in the graph as a new drop-off point, taking into account driving cost, and keeping the drop-off of minimum cost. We do this `len(drop-offs)` times because it could take that many iterations to have a change in the final drop-off point to make its way to the first drop-off point.
Later on we performed further optimizations by using an ILP reduction of TSP run on Gurobi instead of LocalSolver or Google OR-Tools. We also reran TSP through the new drop-off points to further decrease cost.

2 Integer Linear Programming (ILP) Reduction Algorithm

Taking a look at the TSP ILP reduction, we started with the binary variable e_{ij}, which denotes driving along the shortest path from vertex i to vertex j. We made it directed, so e_{ij} and e_{ji} are two different variables. Next, we added the binary variable v_{ik} which denotes dropping off TA k at vertex i. After that, we added the binary variable ind_i which denotes whether a TA is dropped off at vertex i. To improve speed, we assumed that the naive case of dropping all TAs off immediately never happens.
Next up are the constraints. We start by ensuring all TAs are dropped off and only once using $\sum_i v_{ik} = 1, \forall k$. Note that we denote k as the k^{th} TA. Next we setup each ind_i using a Gurobi "or" constraint. Next, we are forced to exit and come back to the starting location s. This means that the outdegree and indegree of s are both 1. Since we can "jump" from vertex to vertex, we don't need to "revisit" s. Algebraically, we write $\sum_i e_{is} = 1$ and $\sum_i e_{si} = 1$. Similarly, for every other vertex, we only visit it if it's a drop-off point. For these, we can write $\sum_j e_{ij} = ind_i$ and $\sum_j e_{ji} = ind_i$ for each vertex i. Last, we have to eliminate subtours which are cycles that don't pass through all drop-off vertices and the starting location. Using Gurobi, we were able to add a lazy constraint callback that for each subtour S, we add the constraint $\sum_{i,j \in S} e_{ij} \leq |S| - 1$. Of course, if there's only one "subtour", then we have found the solution of minimum cost.

3 Semitree Algorithm

This algorithm divides the graph into biconnected components and associated articulation points and creates a tree-like recursive structure using them. It depends on having a subsolver that works well on biconnected graphs and supports forced dropoffs.

Conceptually, from a starting location we look at each biconnected component it is in. For each component, we look at the articulation points included in it. If there are no TAs in the subgraph formed by that point, we don't recurse on it. If there is one TA, we replace the point with a "fake" TA. If there is more than one TA, we replace the point with a "fake" TA and force a drop-off there. We also recurse on the subgraph formed by splitting at this point. We then run the subsolver on the biconnected component with the fake TAs and forced dropoffs. After stitching together the cycle and dropoffs found by the recursive call, we have a cycle and dropoffs for the biconnected component. Finally, we stitch together these cycles and dropoffs for each biconnected component which finalizes our solution. For the sake of word count, we leave out quite a few small details.
As for the subsolvers we used, we started with Optimized TSP and added support for forced drop-offs by not letting the algorithm pick a better dropoff for a forced TA. We later tried using our ILP reduction and gave it support for forced drop-offs by adding the constraint $v_{kk} = 1$ for each forced TA k.

4 Computing Resources We Used

To create our outputs, we ran them on our 2018 Macbook Pro (i7-8850H), 2016 Macbook Pro (i7-6820HQ), and one Hive machine (i7-4770).

5 What Worked Best and Why

Overall, our Semitree algorithm worked best because of its ability to make use of provably more optimal techniques and use of subsolvers. We had two versions of it, one that used Optimized TSP and one that used the ILP reduction. Generally, the ILP version had better outputs but took way too long on some inputs.

6 How We'd Improve Current Algorithmic Performance

Not counting using more powerful computers, we would improve algorithmic performance by further refining our ILP to avoid slower constraint types and improve speed. In cases where ILP is still too slow, we'd improve Optimized TSP by doing more searches for better drop-offs and incorporating elements of this minor improvement algorithm that we also developed. As for Semitree which houses both of these algorithms as subsolvers, we'd search for more properties of graphs we can exploit in a divide and conquer fashion.