CS170 Project Final Report

Ruochen Liu, Xingyu Li, Mengti Sun

December 2019

1 Preprocessing - Floyd Warshall Algorithm

We first find shortest distances between every pair of vertices in the graph using Floyd Warshall Algorithm: We initialize the *dist_matrix* same as the input *adjacency_matrix*. Then we update *dist_matrix* by considering all vertices as an intermediate vertex. We one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.

2 Clustering - K-means Algorithm

We first cluster the vertices on the graph into k clusters. Initialize centroids by first shuffling the rows of the *adjacency_matrix* and then randomly selecting k rows for the centroids without replacement. Keep iterating until there is no change to the centroids or it meets the maximum iteration: Compute the distance between each vertex and all centroids. Assign each vertex to the closest cluster (centroid). Then update the centroids for the clusters by choosing the vertex with the minimal average distance to the all vertices within this cluster.

The choice of k is a trade-off between time complexity and total cost. While a large k will result in a better clustering, it will also increase the time spent on the TSP step tremendously. After some testings, we decided to initialize k as 10. Besides, we also do a sanity check. We calculate the mean and standard deviation of the weight of all paths in the graph. Then we calculate the distance between every vertex and the center of the cluster that it has been assigned to. If the distance is greater than 2.5 times standard deviation plus the mean, we remove this vertex from the cluster and make itself a new cluster. We also set a limit that no more than 4 new centroids will be added.

One last thing we need to do in this step is to make sure that the starting location is in the centroids, so that the starting location will be always in the path.

3 Finding a Hamiltonian Cycle - TSP

After the previous step, we have divided the graph into at most 14 clusters, and according to the abstraction we made in $dist\_matrix$, there exists a path between each pair of centroids, so we know that a Hamiltonian Tour exists and we want to find the minimum weight Hamiltonian Cycle. That being said, we have transformed the problem into a Traveling Salesman Problem (TSP) in this smaller new graph.

With respect to the algorithms for solving the TSP, we hesitated between the approximation method and enumeration. We tested the two methods on the same set of inputs for several times, and it turned out that although the approximation has a lower runtime, the enumeration method guarantees the travelling cost efficiency. As we restricted the number of nodes in the TSP graph, the enumeration can be handled in a relatively short time and provides a better skeleton for the actual path on the original graph. Thus we decided to go with enumeration.

4 Connecting the Clusters - Dijsktra Algorithm

Now we want to connect all the clusters by running the Dijsktra Algorithm on every two adjacent vertices on the TSP path. Since we want to traverse more TA homes along the path, we divide the weight by $\frac{1}{6}$ for every edges that points to the vertex that is a TA home.

The use of $\frac{1}{6}$ is the result of parameter tuning. We randomly picked 10 input files as the training set and swept through values in the range between $\frac{1}{3}$ and $\frac{1}{9}$. Based on the results, we decided that $\frac{1}{6}$ has the best performance.

5 Local Optimizations

We proposed to do three kinds of optimization.

- **small triangle optimization**
  For every TA home t that is not currently on the path, we find the two vertices i, j on the path that are closest to it. We calculate the driving cost directly from i to j plus the walking cost from the drop-off point to t and the driving cost of making a detour to t. If the latter is smaller, we modify the current path to take a detour to t and drop this TA at his/her home.

- **small rectangle optimization**
  Similarly, for each pair of TA homes away from the current path, we compare the current cost and the cost for taking a detour to both homes.

If the latter is more cost-efficient, we detour to drop these two TAs at their homes.

- **round trip optimization**
  We noticed that according to our current solution, sometimes the solution path makes this kind of round trip that we leaves vertex v, drive to a TA home t, drop the TA, and return to v immediately. Since our driving cost is only $\frac{2}{3}$ of the walking cost, in this case it is more optimal to drop the TA at v and let him/her walk to home. So once we detect this kind of round trip, we modify the path and drop-off mapping.