CS170 Final Report - Matthew Tang, Christianna Xu - 12/6/19

1 Final Algorithm(s)

Our final algorithm is the superscore of 2 main types of algorithms: a novel algorithm called the Footsteps algorithm and a variant of Christofides' with an additional condition to see if dropping someone off is more optimal. The best output between the algorithms will be saved.

1.1 Walking vs Driving

Note that we never want to traverse a path that is to only drop off 1 person. It would be more energy efficient to just drop that person off($d < \frac{2}{3}d + \frac{2}{3}d$). We can use a stack to detect when the car goes to drop just 1 person off and traverses the same route back.

1.2 Footsteps Algorithm (v4)

1) Find the nearest house from the current vertex that has not been already visited using Dijkstra's shortest paths

2) Repeat step 1 until all homes are visited. Keep track of the order of the homes in a list.

3) Create the path using the home order by using Dijkstra's, adding all the intermediate edges in the full path. Add a path from the last home back to the start vertex.

4) "Collapse" vertices that are traversed downward by the car and then immediately upward. This represents dropping the person off. Add all collapsed vertices to a "saturated" set.

5) Using all pairs shortest paths, decide the closest vertex on our path to drop the person off to walk. Keep track of who walks in a set.

6) If there are people who walk, re-run steps 1 and 2 with the same graph except with the homes of people removed, and this time we must visit the vertices that have become saturated, so add the saturated vertices to the homes list. Create a path from this home list like step 3 and collapse vertices like step 4, except prevent the algorithm from collapsing saturated vertices.

7) Return the path and dropoffs

1.3 Footsteps Algorithm (v2)

This is the same algorithm as v4, but just return after step 5.

1.4 Footsteps Algorithm (v3)

This is the same algorithm as v4, but don't keep a saturated set.

1.5 Footsteps Algorithm (v3.5)

This is a bugged version of v4 where the newHomes was incorrectly calculated. But somehow, some of the outputs were better so the outputs were still superscored.

1.6 Christofides' Variant (v1)

1) Create an MST M of $G(V, E)$

2) Create O, a Minimum weight perfect matching of the odd degree vertices of M.

3) Combine the edges from M and O to form H. If H does not have an eulerian path then remove edges from odd-degreed vertices from H. If after this removal, a vertex has no edges, remove that vertex from H. Keep track of the removed vertices that are homes.

4) Find an eulerian path in H

5) Insert back the removed home vertices into this path (this path will no longer be eulerian) in the following way. Insert each vertex in the place where the distance is shortest to travel from the previous vertex to this vertex to the next vertex.

6) Visit the vertices in the order generated by the path made in step 5. We return this path to SmartOutput to determine when it is more optimal for people to walk than to drive.

2 Previous Ideas

2.1 Footsteps v0

Initially, the footsteps algorithm was to use a shortest paths tree as a heuristic of what edges to use, and use shortest paths when the SPT edges were all used up. However, this gave very poor results (in the 60s for the autograder score). We also tried using an MST but this was not much better. We then just scrapped the SPT heuristic idea and just had the algorithm navigate to the closest house.

2.2 Christofides (v0)

Initially, we planned to use a very similar algorithm to Christofides using MST, perfect matching, and eulerian paths (detailed in our design doc). However, we found that an eulerian path graph was not always achievable given our inputs so we had to make adjustments detailed in our final version to make the graph eulerian.

3 Computational Resources

Our approximate solvers were very fast. We used multiprocessing to asynchronously create output files. We also created a local autograder to test results without spamming to gradescope. The solver takes about 5mins with multiprocessing and the local autograder takes about 7mins with multiprocessing and caching. The superscorer took around 1 hour on my desktop to maximize the output folders.

4 Libraries

We used the following libraries:
– NumPy (for networkx graph conversion)
– Networkx (shortest path graph methods)
– Pickle (for local autograder caching)
– Multiprocessing (for using async threads to solve inputs)
– Cs188 data structures (Stack and Counter from Cs188 website)