CS170 Project Final Report - Fall 2019

Alex Chan
Jinwoo Park
Brian Stone

Throughout the project, we only used our own laptops to compute the outputs. We divided the input files to four equal pieces and ran our algorithm on separate laptops through multiple nights. The outline of our algorithm uses a steepest descent method. We have two fundamental functions, “descent1” and “descent2”, which changes the initial set of drop off locations in each iteration until a local minimum, the set where making a change to the set does not produce a better cost, is found. Descent1 checks if removing or adding a drop off location creates a better cost. Descent 2, in addition to methods in Descent1, checks if switching a location with another not in the set produces a better cost. In addition, we developed an extra algorithm called “descent12mix” that runs descent1 and descent2 in a row to reduce the running time but increase search depth. To account for heavy running time of descent2, we decided to use descent12mix and descent2 only for 50.in and 100.in cases and only descent12mix for 200.in.

In order to make descent algorithm run in a finite time, it is critical to reduce the time it takes for the evaluation of the cost of the given set of drop-off locations. We first used a revised Dijkstra’s algorithm to find the shortest path between every pair of vertices and saved them in a matrix. Doing so allowed us to have O(1) search time for the cost of the path, making our algorithm run faster.

We next realized that it is not difficult to estimate the cost of the program once we have a set of drop off locations. Creating an optimal traverse through every vertex in a set of vertices is also an NP-hard problem, but we used several algorithms to find a “good” traverse in a quick manner. We first begin by applying nearestNN algorithm, which begins from the starting position and iterates through the set, choosing the next visit as the one with the lowest cost edge. Next, we apply opt-2 exchange to the path which checks if flipping a certain interval of the path produces a better cost. Although we could apply more intense opt-N exchange algorithm, we decided to use opt-2 because this process of cost estimation is run in every iteration of the descent algorithm and we thus want the cost algorithm to be quick. Once we find a good walk across all edges, we simply use the shortest path matrix we produced earlier to find the optimal drop-off location for each TA and their cost in O(n) time.

We decided that, if our next goal is to find a good initial set of dropoff points to start the descent algorithm with. We observed that the set of homes is a great starting point of start, but we decided to produce another set of initial locations for a better view: possibly another “local-min” that might be better. A good strategy might be to find some initial guess of the optimal path, perhaps using some heuristic, and then pick the best dropoff points along this path. If we’ve already used shortest paths on every vertex to calculate the “distance matrix”, it is very easy to calculate the expense of dropping off a TA at any location in the graph, and thus it is also easy to find, on a given path, the best dropoff for a particular TA.

To find our initial path, we make a number of copies of the graph , modify each pair of consecutive edges (with the exception of the final layer, which has no outgoing edges), weight

each edge appropriately, and then run a shortest paths algorithm on this compound graph. From this output we extract the shortest path that ends at a start vertex. This way, we can reliably obtain a path that contains repeated vertices, which the optimal solution certainly will if Rao is to return to his starting point.

After running the program and revising several portions of our code, we realized that the running time of 50.in inputs is extremely quick, only about a minute at max. Because our algorithm terminates at a local-minimum, we do not know if this local-min is the global-minimum. Because the scope of 50.in inputs are narrow, we decided to run several random-walk descent algorithms on randomly generated initial locations. In comparison to descent12mix, this random-walk algorithm iterates through the set of moves in a random order and makes a move if it improves the cost, regardless if it is the steepest one. Thus, the algorithm is expected to view many local-min points that exist on the program and we take the best one out of them. This algorithm outperformed the original algorithm for 50.in in every input, but did not work very well with 100.in and 200.in when the scope of the input increases exponentially.