Travelling Salesman Problem - CS325

Joshua Villwock Jaron Thatcher Ryan Phillips

March 14, 2014

Our Solution

Introduction

To prepare for solving the given test cases within the testing window, we wrote a suite of TSP-related functions/programs. These can be divided into four categories: analysis, solvers, optimizers, and a client-server system. These elements were then used together in a strategy that was actively run by all participants during the testing window.

Analysis

In order to determine how effective each approach was we generated and ran randomized test cases over a range of inputs. Matplotlib was then used to generate a number of plots including the city sets, particular routes of interest, route length vs. time, and route length vs. N.

Solvers

To get an initial solution, we wrote a couple of basic TSP-solvers. These included the following types of algorithms: exact, greedy, and minimum spanning tree.

Exact simply tries every possible permutation on the city set and returns the route with the shortest length. This was not practical for any of the given test sets, but it is a good approach for a very small set of cities. This algorithm has a runtime of O(n!), where n is the number of cities.

The greedy algorithm starts at a specified starting city and then picks the closest city to it as the next stop on its route. This new city is removed from the set of possible route destinations, and the process is repeated as long any unvisited cities remained. This algorithm has a runtime of O(n*n). A variation of this which gave consistently better results (at the expense of another factor of n) simply tries all possible starting cities.

The tsp solver constructs a graph representation of the cities by calculating the distances between every pairs of nodes. It then picks the smallest distance from the list, and adds that edge and the two cities connected to it to a tree. Next all the edges connected to the set of nodes in our current tree, with the exception of any internal ones are added to the active test set. From this set, the minimum distance edge is picked once again. This process is repeated until the tree includes every city. The final step is to do a quick pre-order-traversal of this tree to get a complete route.

Optimizers

These functions take as input a set of cities and a valid route and make specific changes and check for improvement. Then these functions can be run again on the improved route. In practice they are run continually, and the longer the running time, the better the route is - unless the route gets stuck at a local or global minimum.

Reverse Subroute Optimization: Given a particular section length, this function reverses the order of every possible sub-route of this size.

City Swap Optimization: Takes 2 or more cities and swaps their positions in the route.

City Transpose Optimization: For every city in the route, try shifting it to every other possible position in the route. There was also a variation of this function that performed this same operation with not just a single city, but with a subroute.

Exact Section Optimization: Given a particular subroute, it tries every possible permutation to see which is optimal. This uses the same approach as the exact TSP solver, so the same limits apply.

Client-Server

In order to take full advantage of our 24-hour work time, we implemented a fairly simple client-server system to allow us to parallelize the work, especially considering python is single threaded. The server runs, listens for connections, and keeps track of where we are, what kind of work we are doing, and our current best list.

The clients ask for an IP when you start them up, but after that run completely automated, sending a packet to the server asking for work to do, whenever they are not currently working. The server then relies to this request with one of several different packets, telling the client to update it's local cache, or to do some specific type of calculation, or improvement.

Finally, we also had a Monitor client that is really more like a controller, which we could use to instruct the server to change modes, save or load progress, or other parameters specific to the improvement algorithm used.

Strategy

Two different strategies were used: one for the first two test cases and the other for the third test case.

Test1/Test2

We began by running both the MST solver and greedy-all() (which just runs greedy on every possible starting city). This gave us two different starting solutions for our hill climb. Next we ran complete cycles of all of the optimization algorithms on both.

To avoid getting stuck at local minimums, we tried different orderings of optimizers. For most testing seeds, running the optimizers in a fairly random order seemed to do the best job of this, as it avoided getting stuck in a cycle of test-routes. Also, we occasionally arrive at a state in which a single change wouldnt improve the route, but that a sequence of changes was required, some of which would possibly make it worse. To this end, optimizer combinators were written which would run several optimizers, but only check the length at the end of the sequence, rather than throughout the process.

While testing we ran into the case where running a complete cycle of all optimizers failed to improve the route. But by looking at plots of the routes we could clearly see that they were not optimal. It was at this point that we wrote a couple more optimizers (transpose and exact section) in order to progress a bit further.

Test3

Since the third test case was so much larger than the others, the client-server system was used between the members of the group. We didnt implement all our solvers and optimizers for this, but instead picked the ones that could best take advantage of the parallelism. Greedy was a perfect candidate for this, as each client was given a different starting city to try. Then for optimization, we mostly ran subroute reversals, dealing a different set of segment sizes to test to each client. Some of the other methods were attempted as well, but subroute reversal seemed to yield the best rate of improvement for the data set. Within the testing window, the client-server was running test case 3 pretty much non-stop and it never got to a local minimum.