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 tried every possible permutation on the city set and returned the route with the shortest length. This wasnt practical for any of the given test sets, but it is a good approach for a small set of cities (like less than 10). 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 in 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 began by constructing a graph representation of the cities by calculating the distances between every pairs of nodes. It then picked the smallest distance from the list, and added that edge and the two cities that connected to it to the graph. Next all the edges connected to the set of nodes in our tree, with the exception of any internal ones (if they spanned two nodes already in our tree) were added to the test set. From this set, the minimum distance edge was picked once again. This process was repeated until our tree included every city. The final step was to do a quick pre-order-traversal of this tree, and then the result of this gave a pretty good starting route.

Optimizers

These functions took as input a set of cities and a valid route and would try making changes, and then continually test to see if any of the changes yielded a shorter route. Then these functions would be run again on the new, improved route. They could then be run continually, and the longer the running time, the better the route would be - unless the route got 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. This function had a couple of different flavors, including a randomized and a systematic method. The second flavor attempted every possible swap and the first just picked from a random subset of these possibilities.

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. We only tried it with subroute lengths less than ten.

Client-Server

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 the various optimization algorithms, to see which would get us the furthest in the least amount of time (i.e. rate of improvement).

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, a 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.