Solving the Traveling Salesperson Problem

Liam Sarwas

The traveling salesperson problem (TSP) is a challenging problem that scales quickly and becomes impractical to solve with a brute force algorithm. There are n!/2 total solutions, so a brute force solution would necessarily search all of them in order to find the optimum solution, which is impossible for large n. In order to solve the TSP, I implemented two algorithms, hill-climbing with random restart and simulated annealing.

I implemented the random graphs as sets of nodes, with the costs of each path stored in a dictionary, with the keys being tuples of nodes. Paths were then just lists of nodes indicating the order in which the nodes were visited. This allowed for efficient generation of initial paths as well as quick generation of successor states by permutation of pairs of nodes in the path. Scoring was done by iterating through the path and using each pair of elements (e.g. the path 'ABCA' contains pairs 'AB', 'BC', and 'CA') as keys into the dictionary and summing the returned values.

My hill climbing with random restart was implemented by saving the current best state, finding all possible next states and scoring them, then replacing the current state with an improved state. If no improved state could be found, the algorithm had found a local optimum and restarted. Choosing the number of restarts was a balance between time and accuracy. I ended up going with 50 restarts as it was still sufficiently fast to provide almost instant responses while also being very accurate.

Simulated annealing was done by progressively iterating until the temperature had gotten cold enough to stop. During each iteration, a certain number of next moves (I eventually decided on 50, improving the quality of the iterations without adding too much overhead) were randomly chosen and scored. If they were improvements they would be selected, and if they were worse moves, then they would only be selected with a certain probability given as a function of the current temperature and the magnitude of the error (how much worse the new state was). I ended up using an exponential decay model for my temperature function, T = 100\*0.975N. With a cooling factor of 0.975, I had enough iterations to find a good optimum, while still being a very rapid search.

The numbers that I chose for my random restarts and cooling schedule were also deliberately chosen so that the hill climbing and simulated annealing methods would have roughly comparable numbers of nodes searched so that their performance evaluations would be comparable. Currently, on a 10 node graph they are searching roughly 10,000 states. Before, I tested it heavily on up to 30,000 searched states per iteration, and on a 10 node graph, both hill climbing and simulated annealing were in agreement about the optimal solution on every iteration of every graph. I backed my algorithms off so that I would get more interesting results.

The following graph is a depiction of the average performance of the different algorithms over the five different graphs.

While the averages between the two algorithms have substantial differences, the best case result for both algorithms was usually much closer, with simulated annealing having more variation in results. I was surprised at the performance disparity of the two algorithms. I was expecting them to be very close, and if one was better that it would be simulated annealing. I expect that I would see improvements with even more refinement of the cooling schedule, and that the small size of the search space made it easier for the many random restarts to keep the hill climbing algorithm from falling in the pits of local optima.

I do think that linear programming would be very useful for solving this problem, particularly for small to medium cases where exactness is critical. However, it will still not be able to scale to very large n, where approximations will be absolutely necessary.