Homework 10

Sam Daitzman // DSA Spring 2020

Exercise 1: Greedy Heuristic Traveling Salesman Algorithm

> Implement a greedy heuristic algorithm for the traveling salesman problem. We discussed a few options during class, but you are also free to invent your own. Be sure tospecify your algorithm below in a couple of sentences and explain why it is a greedy approach.

I will implement a greedy heuristic-based traveling salesman algorithm using a nearest-neighbor approach. The nearest-neighbor approach tries to minimize the number of trips taken all the way across the map, traversing redundant distance, by using node distance as its heuristic. It simply begins at one node, marks it as visited, gets the node that is closest, travels there, and repeats. This will perform decently well for many basic cases, but can be tripped up easily and does not always find anything near an optimal solution. It is a greedy algorithm because at every instance of a node, it will choose the path that will add the least additional distance. This is its strength and its weakness—at each step, it will only add the minimum additional length possible, but this can add more distance later than is saved at the local substep.

Exercise 2: Local Search Heuristic Traveling Salesman Algorithm

> Implement a local search heuristic algorithm for the traveling salesman problem.We discussed a few options during class, but you are also free to invent your own. Be sure to specify your algorithm below in a couple of sentences and explain why it is a local searchapproach. To find an initial solution, your algorithm should run the greedy algorithm youimplemented above.

To implement a local search heuristic-based traveling salesman algorithm, I will use a two-opt approach. The two-opt approach will take two edges somewhere in the graph, try swapping them as an experiment, and accept the swap if it improves the overall distance. It can be run to continue doing this until no improvement is found. For a pair of edges (u_1, u_2) and (v_1, v_2) it would try the alternate edges $(u1, v2)$ and $(v1, u2)$ which could reduce the total distance by eliminating an unnecessary crossing pathway.

In my implementation, I also support constraining the total number of iterations of the algorithm, and the size of the local neighborhood that will be inspected around each edge to find other edges to swap. The algorithm will continue until it hits these constraints, and in the case of the max iterations constraint it will return its best incarnation of the list yet as soon as it hits the constraint.

Exercise 3: Results

> Compare the results of your algorithms by recording the runtimes and optimality gaps. Depending on your chosen algorithm, you should also consider different starting conditions that may affect the performance. Record your results in a table below. Then, in a few sentences, comment what you observe. Do the results match what you expected?

Optimality

set	*min* ideal	*min* NN	*min* 2opt	*min* both	*gap* NN	*gap* 2opt	*gap* both
gr17	**2085**	2187	4722	2088	102	2637	3
gr21	**2707**	3333	6620	2816	626	3913	109
gr24	**1272**	1553	3436	1278	281	2164	6
gr48	**5046**	6098	19837	5231	1052	14791	185

Generally, these optimality results make a lot of sense. Obviously, none of my approaches quite reached an optimal solution, since I used quite a simple search heuristic and a naive nearest-neighbor implementation. I was surprised to see that two-opt actually performs *worse* than nearest-neighbor when not run with a sensical start path. Below, we also see that 2-opt is significantly less time-efficient (though this could be limited with the constraints on total iterations and local neighborhood size that my implementation allows for).

Runtime

set	t_{nn} (ms)	t_{2opt} (ms)	t_{both} (ms)
gr17	0.18715858459472656	4.389286041259766	4.625797271728516
gr21	0.2627372741699219	7.914066314697266	9.41014289855957
gr24	0.40793418884277344	12.612104415893555	16.251325607299805
gr48	1.2819766998291016	129.57310676574707	116.81389808654785

The runtime here is very interesting. We see that nearest-neighbor is the most efficient by far for every single scenario tested. I was surprised to see how quickly adding more nodes increases the total runtime, although it makes sense since the total number of combinations increases exponentially.

Questions/Things to Follow Up On

- 2-opt vs. 3-opt vs. Lin-Kernigham heuristic. I read about these, but don't fully understand them all, and want to try programming (and/or visualizing) them later.