Description

For the TSP, we decided to implement the simulated annealing function. Inside our function, we used a swap nodes function in order to calculate a “neighbor” path to use in our annealing loop. Before we start the loop, we generate an initial path by parsing the input file then randomly selecting nodes.

The simulated annealing function is given a max temperature, cooling rate, and a starting problem. During each iteration, the temperature decreases. In the loop, we compare our current path to some neighbor path determined by randomly swapping two nodes in our current path. Then we use a probability function to decide whether to keep our current path or pick the new one. If the new neighbor path has a lower cost, we have probability of 1 of taking this as our new path. If the new neighbor path has a higher cost, we have a more complex formula for determining the probability of accepting this path. Here we take a ratio of the current cost - new proposed cost / temperature. Then we compare e^ratio with some random probability from 0 to 1, and if e^ratio is larger, we accept this new path. This is to prevent us from getting stuck at some local minimum.

Then, we run our simulated annealing function in a loop until the inputted time is up.

Rationale

At first we debated between genetic algorithms and simulated annealing. We simply choose simulated annealing because it seemed simpler to implement. We also wanted to make sure not to pick an algorithm that could potentially get caught in local minimums, and miss the best solution, such as greedy or local search algorithms.

Additionally, we decided not to calculate all pairwise distances up front, in order to have more time for running the algorithm. So, we calculated only 2-4 edge costs each time we ran our swap node function.

Pseudocode

```
List, cost = generate_Path(list_of_cities)
While(time_left > 0):
    List, cost = simulated_annealing(List, cost,  max_temp, cooling rate, number_iterations)
End while

Simulated_annealing(List, cost, max_temp, cooling_rate, number_iterations):
        Curr_list = List
        Curr_cost = cost
        Best_list = Curr_list
        Best_cost = Curr_cost
        Curr_temp = max_temp
        While i < number_iterations:
                Candidate_path, candidate_cost = swapNode(Curr_list, Curr_cost)
                If candidate_cost < curr_cost:
                        Curr_list, curr_cost ⇐ candidate_list, candidate_cost
                        If candidate_cost < best_cost
                                Best_list, Best_cost ⇐ Curr_list, curr_cost
                Else:
                        Cost_differnece = curr_cost - candidate_cost
                        If e^cost_difference/curr_temp > random_probabilty:
                                Curr_list, curr_cost ⇐ candidate_list, candidate_cost
                Increment I
                Decrement max_temp with cooling rate
        Return Best_list, Best_cost
```