METAHEURISTICS FOR OPTIMIZATION

TP3 Simulated Annealing

Tien-Tso Ning

1. Introduction

The Traveling Salesman Problem (TSP) is a type of optimization problem where we have a set of N cities, with some given distances between each of the cities. The problem is optimized by finding the shortest tour that traverses every city exactly once, but starting and ending at the same city (implied by the "tour"). The TSP solution can be organized as a list (array) where each index represents the order in which you visit the city, with the value in the list being the city itself. The search space S is a set of feasible solutions to our TSP problem, and in our case would be the set of possible permutation of cities. The neighborhood of a given potential solution s is defined by the movement, which yields a neighbor, and for the TSP is defined as a swap of two cities (one of the available permutations). The optimization of a TSP problem would thus be minimizing the total distance of the tour. To optimize this TSP problem, we have chosen to use the Simulated Annealing metaheuristic.

2. Simulated Annealing

2.1. *Initial Configuration*

The initial configuration of Simulated Annealing is a sequence of cities in a random order (but still starting and ending on the same city, implied by "tour"). This can be implemented by taking the list of cities and randomly choosing three values x, i, j, where x is the number of swaps, and i and j are the indexes to be swapped. However, the starting point and ending point should be the same city. The resulting list from this implementation is a potential solution s in the search space S.

2.2. *Initial Temperature*

The initial temperature is calculated by computing the change in energy (denoted by ΔE). The change in energy is computed by taking a sample of 100 movements (a permutation of two cities) and then by filling in the equation given by:

$$e^{\frac{-\langle \Delta E \rangle}{T_0}} = 0.5 \tag{1}$$

This can be implemented by taking an initial solution s (given by the previous implementation) and making a single swap of two indexes (which is a movement), resulting in the new potential solution s'. The energy of both solutions s and s' can be calculated using the equation referenced above, and the change in energy can be described

by subtracting. This process is repeated 100 times, and the average can be computed by taking the sum of all the changes in energy, and dividing by 100. The choice of the initial temperature is important because we want this temperature to be high enough that we can explore the search space adequately, but we also want to initialize the temperature such that the starting acceptance/rejection probability for movements that could degrade our fitness function to be 50%.

2.3. *Elementary Configuration Update*

At this step, we randomly transform the solution by making a movement (permutation of two cities). Note that this movement is accepted or rejected based on the condition of the acceptance/rejection rule. This can be implemented similar to the initial configuration. The movement is defined by swapping two index locations, and those two indexes are chosen by randomly choosing two values, i, j, where i and j are the index values we are swapping.

2.4. *Acceptance/Rejection Rule*

The probability of accepting a movement is given by the equation:

$$P = \begin{cases} 1 & \text{if } \Delta E < 0 \\ e^{\frac{-\Delta E}{T}} & \text{otherwise} \end{cases} \tag{2}$$

Where we accept the swap always if the change in energy will improve the overall energy of the system (change in energy is negative). We will accept with a probability if the change in energy does not improve the overall energy of the system (change in energy is positive). The change of energy is defined by taking the energy of the neighbor solution and subtracting the energy of the current solution. The implementation is simply an application of the formula.

2.5. *Equilibrium Conditions*

Equilibrium is a state that is reached when a certain number of acceptances were made in the given iteration, or a total number of attempts of acceptance have been made. Once equilibrium has been reached, it is the indication that we need to lower the temperature (by a small amount) in order to continue the SA process. This can be implemented by choosing a number of acceptances and a number of total attempts, and establishing a count for each value. Once the count surpasses the number, proceed to the next iteration.

2.6. *Temperature Reduction*

Temperature should be lowered slowly, since quick temperature drops will make it more difficult to converge to the global minimum in favor of converging to local minimums, thus causing the original starting solution to have an effect on whether or not the algorithm converges to the global minimum, which would be bad for optimization. This implementation is simply an application of the formula, reducing the temperature by 10%.

2.7. *Freezing Condition*

Freezing is the stop condition of the algorithm. Once temperatures have been lowered a given amount of times (three in our case) and no improvement to the energy function has been made, the algorithm is considered to have reached the optimum. This can be implemented by checking the energy fitness at each iteration, and keeping track (in a list) the energy fitness of each iteration. If the energy fitness does not improve (meaning energy fitness of previous iterations are smaller than or equal to the current energy fitness iteration), we should end the optimization algorithm, and the solution that remains is the optimized solution.

2.8. *Baseline: Greedy Algorithm*

The greedy algorithm, which is used as a baseline of comparison for this assignment, is the algorithm that takes the shortest path available at each time-step. Thus, given the neighborhood of potential movements, this algorithm always takes the neighbor with the lowest distance. We would expect that this algorithm be a good baseline since it naively takes the shortest possible path at each time-step and is thus deterministic given the same starting point, allowing us to compare our results.

3. Results and Discussion

3.1. *Simulated Annealing vs Greedy Algorithm*

The results obtained for Simulated Annealing and Greedy Algorithm are displayed in Table 1. Overall, Simulated Annealing was able to optimize and provide a ΔE of -19 to -31, compared to the Greedy Algorithm, which overall did not improve the energy of the system much if at all. The visualization of sampled solutions for the city in cities.dat is given in Figure 1 and the city in cities2.dat are given in Figure 2.

3.2. *Randomly Generated TSP of Size 50,60,80, and 100*

For this section we randomly generated TSPs of sizes 50, 60, 80, and 100 for our algorithm to solve. The Mean and Standard Deviation of run-time is given in Table2. Naturally, as the size of the cities increases, the average run time increases as well.The Mean and Standard Deviation of ΔE are provided in Table3. The Mean of ΔE is similar for city sizes and the standard deviation as well, implying that our algorithm is pretty consistent across the size of the cities.

3.3. *Parallel Tempering*

Parallel Tempering is the process of running many parallel versions of Simulated Annealing, each at a different fixed-temperature. The temperatures of the parallel simulations are arranged such that $T_1 < T_2 < \ldots < T_M$ where M is the total amount of simulations. Neighboring configurations of each simulation can be swapped, and the swap is accepted by the probability given by the equation:

$$p(i,j) = \min\left(1, e^{(\beta_i - \beta_j)(E_i - E_j)}\right) \tag{3}$$

Where E_i is the energy of the given configuration. The idea is that configurations of higher energy will be swapped towards lower temperatures and lower energy configurations will be swapped towards higher temperatures. When M is set to $2, 5, 20$, we can expect that the simulation with $M = 2$ would perform badly since if the temperatures are close together, there is no cooling and optimization would not occur. If the temperatures are too far apart, there is no gradation and will cause issues of cooling too fast, similar to the problem experienced by simulated annealing when the temperature drops at each iteration is too steep. $M = 5$ would make an improvement to get better gradation, and $M = 20$ would run the best.

4. Results

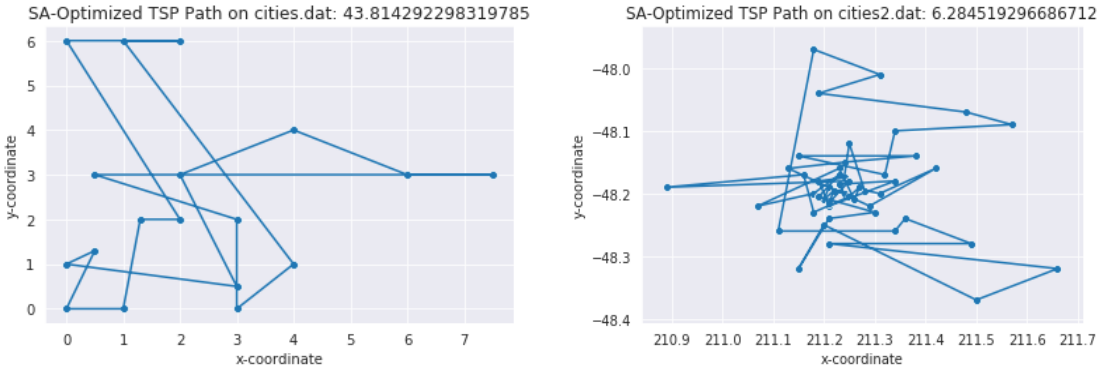

Figure 1.: Visualization of Path in SA and GA for cities.dat

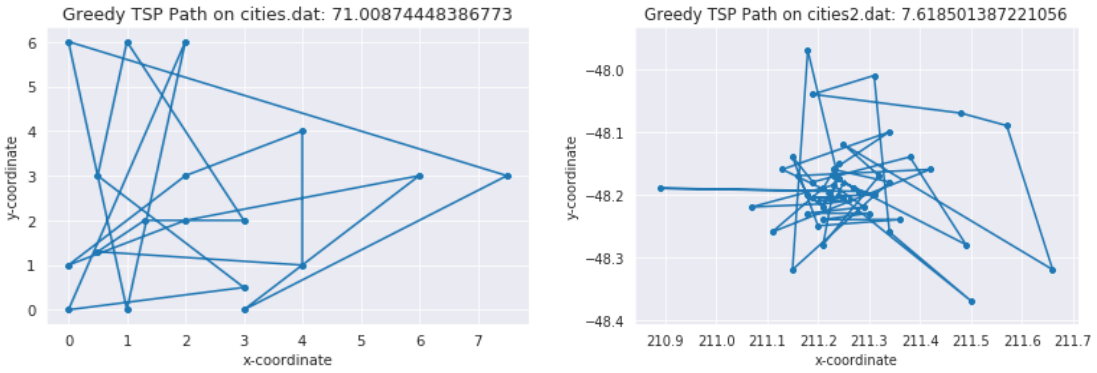

Figure 2.: Visualization of Path in SA and GA for cities2.dat

Table 1.: ΔE of Simulated Annealing compared with Greedy Algorithm over ten runs.

ΔE	Simulated Annealing	Greedy Algorithm
Run 1	-21.519999499605	-1.4210854715202004e-14
Run 2	-23.891679514575372	0.0
Run 3	-25.59323274050066	7.105427357601002e-15
Run 4	-35.59837197734834	2.1316282072803006e-14
Run 5	-34.12143077518981	0.0
Run 6	-19.387414184223083	0.0
Run 7	-19.016588482272617	-7.105427357601002e-15
Run 8	-28.75908635477053	-1.4210854715202004e-14
Run 9	-21.37538835843113	-7.105427357601002e-15
Run 10	-31.02510603726602	-1.4210854715202004e-14

Table 2.: Mean and Standard Deviation of Run-Time in seconds.

City Size	Mean	Standard Deviation
50	26.357562	12.652278112875
60	29.485688	5.1494888825532
80	50.826714	10.607344193407
100	64.778170	31.935345222117

Table 3.: Mean and Standard Deviation of ΔE.

City Size	Mean	Standard Deviation
50	-1512.2186414097	426.7699694825
60	-1298.5356601218	236.07865587345
80	-1386.178306334	426.26291023292
100	-1473.7234803167	373.3927829686