Project 5: Traveling Salesman Problem – Genetic Algorithm with Wisdom of Artificial Crowds

Nick York
CES Department
Speed School of Engineering
University of Louisville, USA
neyork01@louisville.edu

1. Introduction

The Traveling Salesman Problem ("TSP") was introduced in Project 1 and the concept of the Genetic Algorithm ("GA") in Project 4. The TSP requires finding the minimum cost tour ("optimal tour") for a salesman to visit a list of cities without visiting a city more than once, except for the starting city. The salesman must return to their starting city at the end. Each city is represented by Cartesian coordinates and numbered starting at 1. The coordinates are provided by a .tsp file ("TSP file"). The cost to travel between two cities is the Euclidean distance between them, calculated as:

$$cost = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (1)$$

A GA is a heuristic based on the theory of evolution. The theory of evolution states the fittest individuals survive to maturity to reproduce and pass their genes onto their offspring. Specifically, the GA implemented in Project 4 encodes data (cities) as genes and ordered collections of genes (possible tours) as chromosomes. Collections of five hundred chromosomes are constructed by randomly sampling genes without replacement and are encoded as populations. A population changes as new chromosomes replace old ones over successive generations. Chromosomes are selected for a chance to reproduce each generation based on their fitness. A chromosome's fitness is calculated as one divided by the chromosome's cost, which is based on the sum of costs between genes calculated using Equation 1. The fifty fittest chromosomes are guaranteed to be included in the selection pool, but not to reproduce.

During reproduction, a crossover operator exchanges parts of two chromosomes to produce two child chromosomes. Afterward, a mutation operator changes one or more genes for some number of child chromosomes based on a mutation rate of 0.01. Crossover and mutation repeat until the child population is the same size as the parent population. The child population replaces the parent population and repeats these steps until the stopping criterion of one thousand generations is met.

Project 5 introduces the concept of Wisdom of Artificial Crowds ("WoC"). WoC utilizes the collective wisdom of a crowd to arrive at a solution that is more optimal than the experts' solutions alone. WoC relies on four properties to find an optimal solution. First, a crowd must be diverse conceptually (how they arrive at a solution) and cognitively (how “smart” they are). Second, members of a crowd must arrive at solutions independent of other members so that new solutions are considered, regardless of how a solution is found or how optimal it may be. Third and fourth are decentralization and aggregation, which mean no one individual directs the actions of the crowd and that solutions are aggregated in a way that preserves their independence (i.e., solutions are not negatively weighted relative to the most optimal solution).

The goal of Project 5 was to write a program (“the program”) to run our GA from Project 4 multiple times, retain a percentage of chromosomes with minimal costs, and implement WoC to aggregate these chromosomes and combine them into a new chromosome. The GA and WoC run a fixed number of times to compile statistics, including how optimal the WoC chromosome is. Each GA run lasts a fixed number of generations based on the TSP file used.

2. Approach

Overview

I wrote a Python program that allows the user to specify which TSP file or files to use based on a provided filename or file path provided by the user. The user is presented with two options: 1) solve for a single file, and 2) solve for multiple files. Next, the program reads and parses the coordinates in the TSP file(s) based on their selection. If the user selected option 2, then the program parses the coordinates of and solves for the TSP one file at a time.

I used the same code for the GA as Project 4, with minor changes. For instance, I fixed the population size and mutation rate at five hundred and 0.01. Because the population size is fixed, I changed the number of elites added to the selection pool from a calculation based on the population to fifty (one-tenth of five hundred). Finally, I set the number of generations for each file so as to run them for periods of time I found produced optimal results.

Likewise, I used the same code for compiling statistics – chromosomes, costs, and runtimes for each iteration of the GA and WoC are aggregated. After all iterations are run, statistics are compiled on the aggregated values to retrieve averages of cost and runtime, as well as standard deviation. Each GA and WoC chromosome is compared with the best across iterations to find the optimal chromosome.

Two new statistics for WoC are Elites Used and the % Difference. Elites Used specifies how many of the most optimal chromosomes from each of the ten populations is aggregated for WoC. % Difference is calculated as one minus the ratio of WoC optimal chromosome’s cost and the GA optimal chromosome’s cost. 0% indicates both costs are identical. A negative value indicates WoC optimal chromosome performed worse, and vice versa for a positive value.

Wisdom of Artificial Crowds

Initially, I followed the aggregation method outlined in *“Wisdom of the Crowds in Traveling Salesman Problems”* by Yi, Steyvers, Lee, and Dry that was provided to students to implement WoC. The authors used the proportion of edges shared between solutions to create an agreement matrix. Specifically, the authors aggregate “individuals’ solutions into a n × n agreement matrix, where n is the number of cities in the problem” and each cell a_{ij} contains the proportion of solutions that include city *i* connecting to city *j*. Afterward, a nonlinear monotonic transformation function is used

$$c_{ij} = 1 - I^{-1}_{a_{ij}}(b_1, b_2) \quad (2)$$

where c_{ij} is the cost to travel from city *i* to city *j*, and $I^{-1}_{a_{ij}}(b_1, b_2)$ is the inverse regularized beta function whose parameters b_1 and b_2 must be at least one. Equation 2 transforms large a_{ij} values to low c_{ij} values. The goal is to find an optimal tour, prioritizing low values to add connections between cities. The authors solve for the optimal solution using the Lin-Kernighan heuristic (see References). The Lin-Kernighan heuristic works by swapping pairs of sub-tours until a better tour is found.

However, I was unable to implement the Lin-Kernighan heuristic myself. Instead, I used the aggregation method and heuristic described in *"Wisdom of Artificial Crowds – a Metaheuristic Algorithm for Optimization"* by Yampolskiy (our professor), Ashby, and Hassan (see References). Their approach to aggregation is to use the sum of edges shared between a percentage of the most optimal solutions ("elites"), to create an agreement matrix. Then the authors use a novel heuristic to find a new optimal solution using the agreement matrix, which they describe at a high-level.

The authors heuristic builds a new solution one connection at a time. The heuristic starts by adding the connection with maximum agreement between solutions (i.e., city 5 to city 4 makes the solution [5, 4]). Then, the heuristic searches for a connection with maximum agreement that includes the solution's starting city (i.e., 5) and that is not already in the solution (i.e., 4). The same step is repeated for the solution's ending city. Whichever connection has higher agreement is added to the solution (i.e., city 3 to city 5 makes the solution [3, 5, 4]). If there's a tie between connections, then a connection to the closest city, calculated using Equation 1, to the last city inserted and not yet in the solution is added. This process repeats until there are no more connections to add, and the new solution is built.

To implement Yampolskiy et al.'s aggregation method, the program starts with a for-loop that iterates over different percentages of elites to use. For i in range(5, 105, 5) iterates over values from 5 to 100 (upper bound is not inclusive), incrementing by 5. Because the population is fixed, the percentage of elites used varies from 1% to 20%. The for-loop appends a list containing the top i Chromosomes based on the current value of i and appends them to a list, chromosomes_list. Then, chromosomes_list and i are passed as arguments to create_agreement_matrix().

create_agreement_matrix() starts by taking the length of a single Chromosome from chromosomes_list and assigning the value to length. Then, it initializes a 2D numpy array of zeroes with dimensions of length x length and assigns it to agreement_matrix. Finally, the function iterates over each list of Chromosomes from chromosomes_list and further iterates over each Chromosome in those lists to count the connections in them. Each connection is composed of two Genes, indexed by index1 and index2. These indices are used to add one to the corresponding matrix cell (i.e., index1=2 and index2=5 results in agreement_matrix[2,5] += 1). The final value of agreement_matrix is returned by the function.

Next, a cost matrix of Euclidean distance between each Gene is created by passing genes, the list of Genes parsed from the TSP file, to find_cost_matrix(). Find cost_matrix assigns the length of genes to length and uses it to initialize a 2D numpy array of zeros with dimensions length x length, and assigns it to cost_matrix. Then, an outer for-loop iterates over index1 in range(length) and an inner for-loop over index2 in range(length). If index1 is not equal to index2, then the indices are used to retrieve two Genes from genes and extract their Cartesian coordinates. Finally, the cost is calculated using Equation 1 and assigned to the corresponding matrix cell. The final value of cost_matrix is returned by the function.

Finally, WoC can use the aggregated Chromosomes to find a new Chromosome. First, genes, agreement_matrix, and cost_matrix are passed to combine_solutions(). Inside the function, the length of genes is found and assigned to length. An empty list new_genes is initialized to hold indices to genes as they are added to create the new Chromosome at the end. Then, the numpy function argmax is used on agreement_matrix to return the maximum value index of the flattened array. The index is assigned to i. Because the array is flattened, the cell's column and row are found by $\lfloor i \div length \rfloor$ and $i \ \% \ length$. The row and column are added to new_genes as the first and second elements, and the initial length of new_genes is found and assigned to new_length. To track the last index inserted, and whether it was inserted at the start or end of new_genes, last_index_inserted is initialized to indicate the last index inserted can be accessed from the end of new_genes (new_genes[-1]). If 0, then the last index inserted can be accessed from the start of new_genes (new_genes[0]).

The remainder of the heuristic is inside a while-loop. While new_length is less than length, the heuristic iterates over each row (for the starting index of new_genes) and column (for the ending index of new_genes) of the agreement matrix to find the maximum value connections and their corresponding indices. The new indices and their agreement values are returned and compared. Whichever agreement value is greater has its index inserted into new_genes. If there's a tie, then last_index_inserted is used to find the minimum value connection in cost_matrix and insert its index into new_genes. new_length is incremented by one at the end of the while-loop. The process continues until new_length is equal to length. The final value of new_genes is used to initialize a new Chromosome. The new Chromosome is returned and WoC is finished.

3.1 Data

For this project the students were provided six TSP files to test our program. Each file has as many coordinates as the number in the filename. The TSP files are: Random11.tsp, Random22.tsp, Random44.tsp, Random77.tsp, Random97.tsp, Random222.tsp.

3.2 Results

The results of the heuristic for all six datasets are written to .csv files under a folder named Results (Figure 1 and Figure 2). Figure 1 lists the GA statistics. Figure 2 lists the WoC statistics. The best solution for GA and WoC for each dataset is also plotted and can be displayed as a GUI or (by default) saved as a .png file under Results.

File	Generations	Best GA	Avg. GA	Std Dev. GA	Avg. Runtime GA (s)	Total Runtime GA (s)
Random11	20	351.05	351.43	1.15	0.52	5.18
Random22	100	416.09	420.73	7.49	3.67	36.74
Random44	1000	556.61	561.23	3.60	54.94	549.44
Random77	1000	742.92	757.89	12.45	227.75	2277.46
Random97	1500	830.07	849.48	17.64	162.62	1626.18
Random222	2000	1458.74	1579.32	75.73	535.28	5352.77

Figure 1. Statistics for each dataset using solely GA.

File	Best WOC	Avg. WOC	Std Dev. WOC	Avg. Runtime WOC (s)	Total Runtime WOC (s)	Elites Used	% Difference
Random11	351.05	351.05	0.00	0.01	0.23	5	0%
Random22	416.09	416.09	0.00	0.02	0.40	5	0%
Random44	562.07	575.83	41.28	0.06	1.20	5	-1%
Random77	807.31	850.37	12.10	0.08	1.58	100	-9%
Random97	913.38	936.97	27.35	0.11	2.26	85	-10%
Random222	1417.60	1506.32	79.82	0.43	8.63	35	3%

Figure 2. Statistics for each dataset using a combination of GA and WoC (shortened to WoC for conciseness).

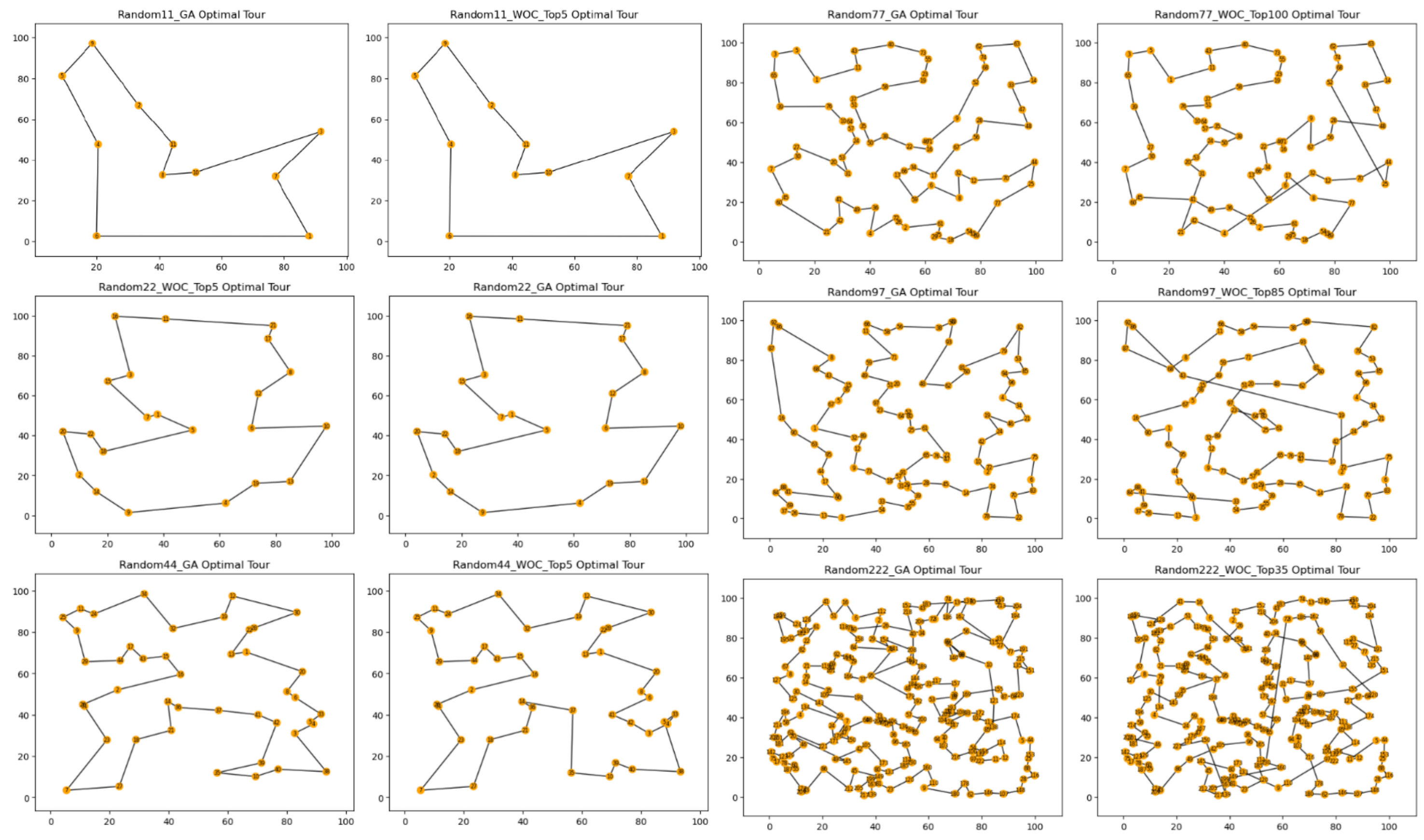

Figure 3. Plots of the best solution for each dataset using GA and WoC.

Discussion

Yampolskiy et al.'s heuristic creates a valid solution without the need to alter it, unlike the Lin-Kernighan heuristic used by Yi, Steyvers, Lee, and Dry. Lin-Kernighan may remove or add an edge to a sub-tour that makes the solution invalid and requires adding and removing other edges to make it valid again. The downside is that Yampolskiy et al's solution does not optimize sub-tours, instead building the solution greedily edge by edge. This results in locally optimal connections between cities, but potentially suboptimal solution.

Size of the problem appears to factor somewhat into the % Difference between the best GA solution and the best WoC solution. WoC did as good as GA for Random11 and Random22 (both 0%), whereas it did worse for Random44, Random77, and Random97. Surprisingly, WoC found a 3% better solution for Random222, the largest dataset tested. Figure 3 shows that despite finding an overall more optimal solution, WoC did introduce locally suboptimal connections (several long edges crossing over many others can be seen in WoC that did not appear in GA). A possible reason Random11 and Random22 did so well is there are fewer edges and consensus is reached by the most optimal Chromosomes at an early generation. The larger datasets have far more connections and run for more generations to arrive at an optimal solution, which may make the stopping criterion more crucial to allow for improvement. On average, WoC did worse than GA, with three instances of worse performance, two instances of no improvement, and one instance of improvement.

In terms of speed, the GA average runtime of each dataset is slower than the final application of WoC, but WoC relies on the GA to aggregate solutions. Because of this, combining GA and WoC takes slightly longer than GA alone, albeit by milliseconds. Hypothetically, if fewer generations could be used for GA while getting the same or better optimal solution by applying WoC afterward, then the hybrid method may perform faster. As it is, I was unable to optimize the program to achieve a reduction in running time relative to GA alone.

References

Lin-Kernighan heuristic

"Wisdom of Artificial Crowds – a Metaheuristic Algorithm for Optimization" by Yampolskiy, Ashby, and Hassan. 2011.