Project 2: TSP Solvers

Mathew Thiel and William Sykes

March 4, 2025

1 Introduction

The first mention of the Traveling Salesman Problem (TSP) in mathematical literature came from Karl Menger after a 1930 mathematics colloquium [2]. The problem was formalized by A.W. Tucker and Merrill Flood at Princeton in the context of Flood's school-bus routing study [3]. Flood moved on to work at the RAND Corporation, which produced a breakthrough in TSP literature as Dantzig, Fulkerson, and Johnson published a method for solving the TSP with 49 cities. Seventeen years later, Held and Karp were next to produce a standard in TSP calculation with their branch and bound method [2]. In 1990, Gerhard Reinelt created TSPLIB, which is a library of TSP challenge instances up to 85,900 cities [2]. Applegate, Bixby, Chvatal, and Cook used the cutting plane method to create the Concorde solver, which was able to solve all 34 instances of TSPLIB [2]. Concorde, as of 2010, is considered "the best available solver for the symmetric TSP" [4]. Modern implementations of the TSP use methods ranging from 2-OPT, Genetic Algorithms, Particle Swarm, and Tabu Search. This project leverages Genetic Algorithms (GAs) and Particle Swarm Optimization (PSO) to solve Asymmetrical (ATSP) and Symmetrical TSP (STSP) problems. For ATSP, distances between cities are not symmetric and therefore vary between two nodes. Unlike ATSP, STSP has symmetric distances, meaning that the distance between two cities is the same in either travel direction.

2 Algorithm Introductions

2.1 Nearest Neighbor Algorithm

To get familiar with the Traveling Salesmen Problem (TSP) the Nearest Neighbor heuristic was implemented to find a tour of cities from a 2D distance matrix from the TSP LIB. This is done by going to the city with the shortest distance from the current city, marking cities as visited as they are visited to continue and eventually complete a tour by returning to the starting city. Out of curiosity a unique shortest path is generated for each city in the provided dataset. Trials would be ran for as many cities there are in the distance matrix for each crossover experiment in the genetic algorithms. The nearest neighbor algorithm is to be a way to compare the effectiveness of the genetic algorithm by generating a shortest path with each unique city as a starting point, where the min, max, and average total distance is tracked here and for each experiment, primarily observing the effect of increased stochasticity as we aim to explore a larger and larger search space as the number of cities increases.

2.2 Genetic Algorithm

The genetic algorithm process starts with generating an initial population of random tours. It then iteratively selects parents, generates offspring through various forms of crossover and mutation, and forms the next generation by selecting the fittest individuals (lowest total distance). The process repeats for a specified number of generations, with the aim of evolving the population towards an optimal or near-optimal solution by minimizing the total distance of the tour. Each individual in the initial population is a random tour of cities, and a permutation of cities representing a possible solution. A specified number of parents is selected from the population based on the objective function (total distance of the tour). These parents are used in the ordered crossover (0X) and an attempt at a new crossover function dubbed CX2 described by ncbi. Both crossovers are done in 2 parts. We start with the crossover of the genes from parent to parent creating 1 child for each crossover. Ordered crossover

does this by taking two parent tours and copies a continuous random subset of genes (cities) from one parent into the offspring then filling the remaining positions in the offspring with genes from the second parent, preserving their order, but starting from the end of the copied subset to ensure a valid tour without duplicate cities. This returns a child or offspring tour generated from the two parents, containing a unique set of cities inherited from both parents. For example, given parent1 = [0, 1, 2, 3, 4] and parent2 = [4, 0, 2, 1, 3], an offspring might be [0, 1, 2, 4, 3], depending on the randomly selected subset for copying from the first parent. CX2 is based on steps proposed by [7]. Imagine we have two parent tours, parent1 = [0, 1, 2, 3, 4] and parent2 = [4, 0, 1, 3, 2]. When we apply the CX2 crossover, first, an offspring is initialized with None values. We start a cycle with the first bit from parent2 for the offspring so offspring[0] = parent2[0] = 4, or [4, None, None, None, None]. From here we find 4 in parent1, which is at index 4, we take that bit from the same index in parent2 which is 2, to place in offspring1 at index 4. So offspring1 becomes [4, None, None, None, 2] Next we find 2 in parent1 at index 2 and take the bit from parent2 at index 2 which is 1 and place 1 in offspring1 at index 2. [4, None, 1, None, 2] we do this until the cycle is completed by reaching the original starting bit, 4. After forming the cycle, we fill the remaining None values with bits from parent1 where no specific cycle bit was assigned, ensuring that all genetic information from parent1 is preserved, guaranteeing an offspring with a complete and valid tour. These methods first create a new gene from the 2 parents then in the respective crossover functions called in the genetic algorithm driver function. We then mutate individuals generated by the crossover process, by iterating over each offspring and, with a probability equal to the mutation rate, we apply the mutation operation. This involves swapping two randomly selected genes (cities) within an offspring. This introduces further genetic diversity into the population, in an effort to avoid local minima and ensure the genetic algorithm explores a wider range of potential solutions. Each next generation is created by combining the parents and offspring to the given population size into a single candidate pool. So, once the mutations occur the best individuals are chosen for the next generation, and this process continues for a set number of generations. We play with the stochasticity of the two crossover strategies. Experimenting with each separately, then introducing a chance for different types of crossover to occur within the same population, all working in tandem with the mutation of individual offspring. This is in an effort to explore a wider solution space and maintain genetic diversity across generations, the more we explore the greater the potential for better overall solutions.

2.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) was initially introduced by Kennedy and Eberhart[8] in 1995 and was inspired by bird flocks and other similar communities. Unlike Genetic Algorithms (GAs), which leverage crossover and mutation operators to generate new individuals, PSO uses information shared between individuals, also called particles, to guide the swarm toward a convergence point[13]. PSO is considered a meta-heuristic as it does not make any assumption about the problem set, but no guarantee of finding the optimal value is given. An advantage of PSO is that, unlike algorithms like gradient descent, PSO does not require a gradient and therefore does not have a differentiation requirement [13]. In PSO, a swarm is a community of particles that all share information with each other based on personal and global best solutions. Particles "remember" their best path, best cost, and best global cost in order to traverse the problem space. The sharing of information happens through a velocity operator that determines whether the particle follows its own path, tends towards its local optimal solution, or tends towards the global optimal solution. At each iteration, each particle generates a new path that is evaluated against its personal and global best using a cost operator, which is generally the objective function of the optimization problem. A general framework for a continuous PSO algorithm is as follows:

1. Create a population of particles with a random initial position
2. Begin iterating through the collection of particles
   (a) Evaluate the cost of the current position of the particle
   (b) Compare the cost with the particle's personal best and the swarm's global best solution. If it is more optimal than either solution, save the current solution to the respective less-optimal solution.

(c) Update the velocity and position of the particle

3. Keep iterating through all particles until a specified number of iterations or stop criteria

The formulation for the position and velocity is given below in Equation 1[5]:

$$V_{t+1} = w * v_t + c_1 * \text{rand}_1(pbest_i - x_i) + c_2 * \text{rand}_2 * (gbest - x_i) \tag{1}$$

$$x_{i+1} = x_i + v_i \tag{2}$$

Equation 1 is the velocity update step, where w is an inertia factor that scales the velocity, v_i is the velocity at i, rand are randomly generated numbers that introduce stochasticity into the swarm, x_i is the position at i, *pbest* is the personal best of the particle, *gbest* is the global best of the swarm, and c_1 and c_2 are parameters that give more prevalence to their respective optimal. Increasing the value of c_1 increases the tendency for the particle to follow its personal best, while increasing c_2 increases the tendency for the particle to follow the global best. The inertia parameter w controls how much the velocity affects the algorithm. Since the solution vector for the TSP is discrete, we must extend the above formulation to handle discrete problems.

2.4 Discrete Particle Swarm Optimization

For this specific problem, the solution vector is a permutation of a whole step vector starting at 0 and ending at the number of cities in the problem. Each permutation is a "path" through the problem space, passing through each city (or node) exactly once and returning to the starting city. The evaluation of each solution, or cost function, is given by the sum of distances between each city on a path. This problem can also be represented as an undirected graph with each city as a node and edge weights equal to the distance between each node. Following Goldbarg et. al.[5], discrete PSO presents a unique challenge of defining velocity operators. Even in continuous PSO, there are essentially 3 possible movement choices:

1. Move to a new solution.
2. Move to the particle's personal best.
3. Move the the swarm's global optima.

Therefore, for a discrete version, we can represent the velocity as a ternary vector, with each value corresponding to a movement direction, i.e. 0 to move to a new solution, 1 to move to personal best, and 2 to move to global best. The choice of which direction to move is given by user-defined probabilities that change over n iterations by defined scalar multipliers. The values in the ternary probability set correspond to the values of the velocity operator, i.e. the first element in the probability set corresponds to the first element of the velocity operator. The logic for the discrete PSO algorithm is as follows:

1. Create a population of particles with a random initial position
2. Begin iterating through the collection of particles
   (a) Evaluate the cost of the current position of the particle
   (b) Compare the cost with the particle's personal best and the swarm's global best solution. If it is more optimal than either solution, save the current solution to the respective less-optimal solution.
   (c) Draw a velocity operator and update the position of the particle
3. Update the probability vector
4. Keep iterating through all particles until a specified number of iterations or stop criteria

The probability update is given by:

$$\begin{aligned} p_1 &= p_1 * lrP1 \\ p_2 &= p_2 * lrP2 \\ p_3 &= 1 - (p_1 + p_2) \end{aligned} \tag{3}$$

p_1 is the probability that the particle moves towards a new solution, p_2 is the probability that the particle moves towards its personal best, and $p3$ is the probability that the particle moves towards the global best. The learning rates $lrP1$ and $lrP2$ dynamically change the probabilities over n iterations. To move the particle to a new position, two methods are used. To move the particle to a new solution, we use a Kernighan-Lin bisection of the current path. To move the particle to an existing personal or global optimum, a backward-forward path re-linking operation is used.

2.5 Kernighan-Lin Bisection

The Kernighan-Lin bisection is a graph partitioning strategy that attempts to improve the cost of a solution by swapping nodes that overlap in two equal partitions. The seminal paper by Kernighan and Lin *An Effective Heuristic for the Traveling Salesman Problem*[10] describes the methodology behind their heuristic, which is essentially an adaptive k-opt movement. Kernighan and Lin state that a non-optimal solution T is known to be non-optimal because it has k elements that are "out of place", so the algorithm attempts to find k and subsets x_k and y_k element by element for the lowest cost[10]. This algorithm is the chosen local search algorithm for this implementation as Goldbarg et. al. state that the Kernighan-Lin algorithm is "... a recognized efficient improvement method for the TSP"[5]. Additionally, this method is used in the well-known Concorde solver created by Applegate et. al.[2], which is a very strong solver for the TSP. The implementation in this project makes use of the existing Kernighan-Lin bisection algorithm in the graph library networkx[1], which may be sub-optimal in this problem context.

2.6 Path Re-linking

Path re-linking is a search procedure that is normally used with algorithms like Scatter Search, Tabu Search, and the Greedy Randomized Adaptive Search Procedure (GRASP)[9]. Path re-linking has multiple forms, including forward, backward, mixed, truncated, etc. [12]. The general goal of path re-linking is to move an initial solution to a target solution through a series of swaps, or "re-links". In this implementation, a forward-backward approach is used, where the path re-linking strategy is applied from the initial array to the target array and vice versa, simultaneously [5]. The logic for this implementation of path re-linking is as follows:

1. Supply an initial path and target path to "re-link to"
2. Iterate through each element in the initial path or until a stopping criterion is reached.
3. At each iteration, define a target value x that is at the index of the current element in the target path.
4. Split the initial path into two splits on the index of the target value in the initial path.
5. Shift the initial path left until the target value in the initial path is at the same index as the target value in the target path.
6. Calculate the cost of this new path, and if it is smaller than the current optimal cost, this solution becomes the new optimal.

To use a forward-backward path-relinking, the same logic as above is used simultaneously, except in the backward case, the target path is being relinked towards the initial path. This increases the number of possible paths traversed, at the cost of increasing computation time to perform both operations.

3 Results: Genetic Algorithm OX, CX2, OXCX2 vs Nearest Neighbor heuristic

Results	0X	CX2	0XCX2	NN
Best	1345	1422	1366	1590
Worst	1786	1910	1820	1818
Average	1538	1679	1603	1689

Table 1: Results for ftv33 asymmetric distance matrix, 33 cities with a known minimum distance of 1287, Averaged for 33 runs per genetic algorithm crossover experiment

Results	0X	CX2	0XCX2	NN
Best	2085	2085	2085	2178
Worst	2153	2181	2154	2427
Average	2091	2123	2096	2294

Table 2: Results for GR17 symmetric distance matrix. 17 cities with a known minimum distance of 2085, Averaged for 17 runs per genetic algorithm crossover experiment

Results	0X	CX2	0XCX2	NN
Best	732	907	810	864
Worst	1015	1162	1112	1015
Average	871	1015	939	926

Table 3: Results for Dantzig42 symmetric distance matrix. 42 cities with a known minimum distance of 699, Averaged for 42 runs per genetic algorithm crossover experiment

4 Results: Discrete Particle Swarm Optimization with LK and Path-Relinking

Results	Best	Worst	Average Time (s)
ftv33.atsp	2370	2730	5.20
pr76.tsp	152332	153956	28.71

Table 4: Results for Discrete PSO with LK and Path Re-Linking, over 5 runs of the algorithm, each having 100 iterations. The best-known solution for pr76.tsp is 108159, and the best-known solution for ftv33.atsp is 1286

5 Analysis

5.1 Discrete Particle Swarm with LK and Path-Relinking

The implemented algorithm was much slower than anticipated. This could be due to an incorrect implementation of the methods outlined in [5]. The optima reached were also very poor. The algorithm is too slow to have a meaningful number of iterations, and the tendency to get stuck at convergence is large past 50-100 iterations. When running the ATSP problem for $\tilde{2}00$ iterations, it was possible to reach solutions as low as $\tilde{1}500$, which was only $\tilde{2}00$ off of the optimal solution. To improve the algorithm, I would improve the current LK implementation by creating custom k-opt logic, which was still in progress by the deadline. There are also multiple different path relinking methods and velocity operations to try, along with different representations of the problem. Goldberg et. al. and similar subjects represent the velocity vector as a composition of the various velocities, meaning each part of Equation 1 (separated by additions) is instead a composition of the next part. This could make

sense if the path is updated using Hamiltonian differences instead of a roll technique. However, the base implementation should still be the same. Through my testing, I found that the PSO algorithm converged to a point fairly quickly, but got stuck in that point frequently. To help solve this issue, I had the idea to include some randomness in path exploration. Before partitioning with the LK algorithm, I randomly shuffle the path a random amount of times so that the algorithm can see more problem spaces. This actually had a slight improvement in solution, but the speed of convergence was slower, especially at higher iterations. The probability operations also seemed to need work. In the reference paper [5], the probability values and learning rates are defined as $p_1 = 0.9$, $p_2 = 0.05$, $p_3 = 0.05$, $lr_1 = 0.95$, $lr_2 = 1.01$. The rationale behind these values is that initially, p_1 should be high so that the particles can explore the space and then gradually, the dominant probability should shift to the preference towards global optima. It makes sense to initially favor exploration, but increasing p_2 by a fixed scalar allows for exponential increase after > 100 iterations. Even if you balance probabilities to all equal 1, the probability that is increased will be absurdly high, while the other probabilities will be almost non-existent. Instead, I think that the learning rates should be functions of the number of iterations, where over time the rate of increase slows to a marginal amount at high iteration numbers. For example, p_1 should initially be high but slowly decrease to a lower bound, say 0.1. p_2 should follow an inverted parabola, where the probability initially increases, then decreases after hitting a maximum value, say 0.5. p_3 will still be $1 - (p_1 + p_2)$ so that the preference towards the global increases with higher iterations. This way, there is proper allowance for exploration, but tighter enforcement of convergence at higher iterations. High-Performance Computing (HPC) techniques would greatly benefit this algorithm, as essentially this is a PSO algorithm with two inbuilt local searches. A large speed improvement may also be to include stopping criteria in the path re-linking and LK algorithms. Instead of just a constant max number of iterations, a similarity metric could be implemented where if the algorithm is going down a previously traveled path, the search stops.

5.2 GA: OX vs CX2 vs OXCX2 vs NN

For the above results all parameters were kept constant for each strategy in the table. That being a population size of 500 with 150 selected parents, a mutation rate of .8, and 200 generations per iteration. Based on the results of like runs on the same distance matrices a genetic algorithm implementation with solely ordered crossover of parent genes to form offspring had the best shortest path found for ftv33 and dantzig42, on average finding a path 5 to 10% less than CX2, OXCX2, and the NN alg for each city. This was expected to beat out the NN algorithm for every experiment however I did not expect ordered crossover alone to do better than CX2 or a combination of the two, again even with mutations after crossover in every experiment. I believe this is due to misimplementaion of [7] steps to there proposal for a bit to indice focused crossover. I believe CX2 experiences more premature convergence, as when a cycle is completed without being swapped deeper into the genes the population begins to experience a high level of uniformity without containing sufficiently near-optimal structures[6]. I believe this is because CX2 always retrieves the first bit of the second parent where as OX2 randomizes where a gene is split and swapped between parents. The uniformity of picking the 1st bit, fails to take generate stochasticity and I believe leads to the population becoming more uniform towards a local instead of the global optima. I think this is confirmed with the OXCX2 experiement, where we randomize the type of crossover for each pair of parents, as CX2 might be introducing more uniformity then stochasticity into the population, actually diminishing the effect of OX2s ability to generate a larger search space through the random swapping of subsections. OXCX2 routinely performed better than CX2 alone, but I believe this is due to the power of OX2 crossover, rather than what CX2 is adding to the search space. That being said, attempts were made to shorten the time for objective functions and fitness to be calculated as this would have allowed populations in the tens of thousands, parents in the hundreds, and generations in the thousands, but this proved quite challenging. The GA would take about 30 seconds to a minute per iteration of the above experiements depending on the number of cities. What was found outside of these experiements however was no matter the crossover strategy used, increased population size, number of parents, and number of generations had the greatest effect on the best solution found by the algorithm. Runs on ftv33 with OX and OXCX2 with an initial population of 1000 individuals, 2000 generations and 300 generated parents per population yielded constant best solutions between 1324 and 1387, only 3 - 7% longer than the optimal solution for the asymmetric TSP, and 732 - 770 for Dantizig 42, only 4 - 10% longer than the known global optima. Manipulating the mutation rate would often speed up the convergence towards a local optima, a run

may stagnate after 200 generations and 200 generations later make a big leap towards a new optima. So I am curious how these algorithms would function on better hardware or with the objective function parallelized. Compared to the Nearest Neighbor heuristic for each city I am surprised how well this heuristic did. Often only doing a little worse than CX2 on average, however was never able to find a global optima, where as with CX2 it is possible to find. Combing the two crossover strategies was my attempt at breaking the uniformity and stagnation I would experience after about 1000 generations of OX and CX2, however I think this can only be done by implementing a possible extinction factor, where a group of individuals or tours are linked to a local optima and thus set as extinct to possibly break the stagnation the GA eventually experiences if the global optima is not found. Then we could possibly cut a section of a population that is causing convergence on a local optima and widen the search space.

References

[1] kernighan_lin_bisection — NetworkX 3.2.1 documentation.

[2] D. L. Applegate, R. E. Bixby, V. Chvatál, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[3] K. M. Curtin, G. Voicu, M. T. Rice, and A. Stefanidis. A Comparative Analysis of Traveling Salesman Solutions from Geographic Information Systems. *Transactions in GIS*, 18(2):286–301, Apr. 2014.

[4] D. Davendra. *Traveling Salesman Problem, Theory and Applications*. Dec. 2010.

[5] E. Goldbarg, G. Souza, and M. Goldbarg. Particle Swarm for the Traveling Salesman Problem. volume 3906, pages 99–110, Apr. 2006.

[6] D. Goldberg and P. Segrest. *Genetic Algorithms and Their Applications*. Massachusetts Institute of Technology, Lawrence Erlbaum Associates, 2024.

[7] A. Hussain, Y. S. Muhammad, M. Nauman Sajid, I. Hussain, A. Mohamd Shoukry, and S. Gani. Genetic algorithm for traveling salesman problem with modified cycle crossover operator. *Computational intelligence and neuroscience*, 2017.

[8] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, Nov. 1995.

[9] M. Laguna, R. Martí, A. Martinez-Gavara, S. Perez-Peló, and M. G. C. Resende. 20 years of Greedy Randomized Adaptive Search Procedures with Path Relinking, Dec. 2023. arXiv:2312.12663 [math] version: 1.

[10] S. Lin and B. W. Kernighan. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, 21(2):498–516, 1973.

[11] Q. T. Luu. Traveling salesman problem: Exact solutions vs. heuristic vs. approximation algorithms. *Baeldung on Computer Science*, Jun 2023.

[12] M. G. Resende, C. C. Ribeiro, F. Glover, and R. Martí. Scatter Search and Path-Relinking: Fundamentals, Advances, and Applications. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, International Series in Operations Research & Management Science, pages 87–107. Springer US, Boston, MA, 2010.

[13] X. Yan, C. Zhang, W. Luo, W. Li, W. Chen, and H. Liu. Solve Traveling Salesman Problem Using Particle Swarm Optimization Algorithm. 9(6), 2012.