DSA4212 Assignment 3 Group 27

Er Song Heng A0216099X, Derek Ng Wei Kang A0215125U, Chiong Shao Yong A0216613M

1. Introduction

The Travelling Salesman Problem (TSP) is an algorithmic problem where given a list of the coordinates of cities and the distances between them, we want to find the shortest possible route that visits each city exactly once and returns to the starting city. Despite its straightforward nature as a classic optimization problem, it belongs to a class of combinatorial optimization problems known as NP-complete. TSP is known to be a NP-hard problem, which means that it becomes increasingly difficult to solve as the size of the problem grows.

Using the file *cities.npy*, which contains the 2D coordinates of 1000 cities from 0 to 1, this assignment tasked us to find the permutation of these cities that minimizes the total euclidean distance, L, between all the cities in the tour.

There are two main classes of algorithms that deal with the TSP, exact and heuristic algorithms. Exact algorithms are ones that find the most optimal solution for any given optimization problem by exhaustively going through the entire search space directly or indirectly. Heuristic algorithms on the other hand, aims to approximate the optimal solution in hopes that it is near optimal. In the case of the TSP, the group has opted to use heuristic algorithms as opposed to exact algorithms as it will take too much time to exhaustively go through the entire search space of 999x1000! combinations for any given starting city.

We will also only be considering symmetric cases, where for example, the distance between city A and city B is the same as the distance between city B and city A.

2. Algorithms

2.1. Search Heuristic Algorithms

2.1.1. 2-opt

2-opt is a local search algorithm that takes a route that crosses over itself and reorders it so that it does not cross anymore. It swaps two edges in a tour each time and iteratively improves this operation until no further improvements can be made.

We initialize the initial route to be from city 0 to city 999 in order. As long as there is no significant improvement in the total distance of the route, we swap the positions of the cities with two ‘for’ loops. Each time we discover a path with a shorter total distance, we replace the optimal route with this new route. We set a small threshold for improvement at 0.001, which is akin to the requirement that the distance of the new route has to be less than 0.1% shorter than the previous optimal route. Using this simple implementation of the 2-opt algorithm, the time complexity is $O(n^2)$.

We obtained $L = 26.6$, with the tour starting at city 0 and ending at city 407, and the algorithm ran for about 18 hours.

Since the 2-opt algorithm can be layered together with other heuristic algorithms such as Nearest Neighbors, the 2-opt algorithm is a possible choice to improve upon the final routes optimized by these heuristic algorithms. This is corroborated by a study that claims that the 2-opt algorithm can significantly improve the accuracy by about 10.27%. (Nuraiman et al., 2018)[15]

2.1.2. Simulated Annealing(SA)

SA is a local search algorithm that is capable of breaking out of local optima. (Nikolaev & Jacobson, 2010)[16] The main idea behind the SA is to simulate physical annealing with solids in which solids are heated and cooled until a desired physical structure is reached. In the context of SA, this means that both better and worse solutions are accepted for the next iteration depending on some fixed criteria. This ability to accept worse solutions allows for escaping local optima in hopes of reaching the global optima.

The algorithm in the context of TSP is as follows:

1. Initialization:
    - Initialize a starting permutation path S (0,1,2,3,....999)
    - Initialize a starting temperature T.
    - Initialize a cooling schedule t.
    - Initialize the number of iterations M the algorithm runs for.
    - Calculate the L_OG with respect to S.
2. Updating for each iteration in M:
    - Update T = T * t
    - Set 500 iterations within each M
    - Within each iteration, randomly pick 2 cities in S to swap.
    - Calculate the L_new for this new permutation.
    - If (L_new<L_OG), accept this new permutation.
    - **If (x < exp((L_OG - L_new)/T)), accept this new permutation as well**, where x is a randomly generated uniform variable.
    - Otherwise, revert back to previous permutation.

The line in bold is the advantage of the simulated annealing where it accepts solutions that could be worse off than previous permutations. The time complexity of SA is at least O(n^4) where n is the number of cities we have. (Sasaki , 1987)[20]

On top of the base model above, experiments based on the literature have been done. The first proposition in the paper was that an annealing schedule was not needed for the solution to converge to a global optimum and as such an experiment without the scheduling was done to look at its effects. (Nikolaev & Jacobson, 2010)[16] From the table it can be seen that the results were indeed not very different.

No. of iterations	With scheduling	Without scheduling
200	524.91	518.42
400	517.92	517.44
600	521.14	516.99

Table of L values with and without temperature scheduling

The second proposition was that high starting temperatures make the search more inefficient thus experiments with decreasing lower temperatures were done. (Nikolaev & Jacobson, 2010)[16] Based on the table itself it does not seem to be the case.

Starting temperature	L values
1000	531.20
500	513.58
100	528.53
50	525.19

Table of L values for different starting temperatures

Relative to the other algorithms, the simulated annealing heavily underperforms due to poor choice of parameters. With the final L obtained L = 509 in 26.5 seconds. The path starts out at city 710 and the second last city is 164.

2.1.3. Lin-Kernighan heuristic

The Lin-Kerninghan heuristic (LKH) is one of the best and most successful local search algorithms proposed in 1973(Lin & W, 1973)[13] . LKH takes a Hamiltonian cycle and improves it by repeatedly swapping pairs of edges in the tour to obtain a shorter tour until it encounters a local minimum. The LKH favors small numbers of edges to replace at a step and is adaptive to converge to a solution even for large TSP instances. It also introduces two parameters during the search: backtracking depth which is an upper bound on the length of the alternating trail after backtracking, and infeasibility depth which is an alternating path length beyond which it begins to be required that closing the current trail yields an exchange to a new tour. The time complexity of the LKH is estimated to be $O(n^{2.2})$. (Lin & W, 1973)[13]

We used the Python 3 library, elkai, as it contains the LKH solver and has some wrapper code to expose it to Python. Hence, we did not need to compile and build the LKH solver ourselves. Using the function call `solve_float_matrix` with 10 iterations, we obtained the final tour output and calculated the sum of euclidean distances between each city in the tour ourselves.

We obtained L = 23.472, with the tour starting at city 0 and ending at city 407 and the algorithm ran for 130 seconds.

Subsequently, we used the 2-opt algorithm to improve upon the final route from the LKH algorithm as an additional layer. By calculating the change in cost between each pair of cities, we updated the best route to the new route if we observed any improvement in the cost. We obtained L = 23.461, with the tour starting at city 0 and ending at city 407 and the algorithm ran for about 130 seconds as well.

2.2. Hybrid Heuristic Algorithms

2.2.1. Genetic Algorithm

The Genetic Algorithm is an algorithm inspired by evolution. It replicates the mechanism of natural selection (survival of the fittest) and solves the optimization problem in TSP using crossover, selection and mutation operators. The different phases in the Genetic Algorithm are:

1. Initialization - An initial population of solutions is randomly created consisting of chromosomes, a list of genes (cities) in order.
2. Evaluation of fitness - The fitness score of the population is calculated. In the case of TSP, since a shorter distance equates to a higher fitness, we use the inverse of the distance as its fitness score. Hence, the higher the fitness score, the more preferred the route.
3. Selection - The fittest solutions are selected for reproduction through a rank-based selection based on the percentage of elites specified in the function.
4. Breeding - Two selected parents exchange genetic material. In our function, this means that a random length of the parents are swapped with each other.
5. Mutation - Some parts of the routes are changed randomly according to the mutation rate specified in the function. This introduces variations in the solutions further down the generations.

We implemented the Genetic Algorithm with some parameters: Population size, Elite Size, Mutation Rate, and Number of Generations (iterations). To experiment with it, we used a population size of 100, elite size of 20, mutation rate of 0.01 and 500 generations to obtain L = 471.8. We also plotted a graph of distance against generations and it is observed that the algorithm had difficulties converging even at 500 generations.

We further tuned the parameters for mutation rates (0.1, 0.01, 0.001), elite sizes (10, 20, 50, 80) and population size (100, 500, 1000). It was observed that a population size of 100, 20 elites, 0.001 mutation rate and 500 generations yielded L = 370.2. The Genetic Algorithm gives a high

L as compared to other simpler algorithms and this may be because that the Genetic Algorithm is not able to converge due to the large instance of the number of cities. This is corroborated by the study showing that the Genetic process uses permutations between cities to find the best route but those permutations are random therefore they offer no guarantee on the optimal path. (Abdulkarim & Alshammari, 2015)[1]

2.2.2. Ant Colony Algorithm

The Ant Colony Algorithm is an algorithm that takes inspiration from how ants discover optimal food paths for other ants to take. The general idea is that everytime an ant goes out and discovers new paths it will leave pheromones that will inform other ants that the path is a desired one. Shorter more optimal paths will have more pheromones and as more ants venture out, they will take these paths and eventually convergence occurs and all the ants take only the optimum path. The time complexity of the ant colony algorithm is still an ongoing field of research and is therefore unknown. (Neumann et al., 2009)[14]

In the context of TSP, the base ant colony algorithm goes as such.

1. Initialisation
    - Initialise number of ants we want, N
    - Initialise number of iterations, M
    - Initialize pheromones of each path to be equal
    - Initialize an evaporation factor
    - Initialize alpha and beta
2. Updating
    - For each iteration M
    - For each ant, it will travel through each city and its travel path will be determined by the amount of pheromones in each path and a probabilistic function that is dependent on the pheromones.
    - After the ant has traversed through all the cities, the pheromones are updated based on how much was added from this current ant and the evaporation factor.

$$p_{ij}^{k}(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]^{\alpha}\cdot[\eta_{ij}(t)]^{\beta}}{\sum\limits_{s\in J_k(i)}[\tau_{is}(t)]^{\alpha}\cdot[\eta_{is}]^{\beta}}, & \text{if } j\in J_k(i) \\ 0, & \text{otherwise} \end{cases}$$

<u>Probabilistic function</u>

* The left inner term of the numerator = pheromones between city i and j at time t
* The right inner term of the numerator = the inverse of distance(i,j)

As a small extension to the base Ant Colony Algorithm, the Ant Colony System was also implemented. There were 3 main differences between this extension and the base model that were implemented. Firstly, the probabilistic function to decide which path the ants take is different. Second, the global updating rule for pheromones is different, i.e. the way the evaporation factor updates the pheromones. Lastly, the way each ant updates the pheromone levels for each path they took.

Using guidelines from the literature to decide which parameters we should use (Wei, 2014)[23], we obtained an L = 28.377 which took 10 hours to run. The path starts from city 851 and the second last city is 393. The parameters were also experimented on the Ant Colony System algorithm but the L that was obtained was L = 184.59 and it took 12 hours to run. The path started from city 1 and the second last city was 761.

2.3. Tour Construction Heuristic Algorithms

2.3.1 Nearest Neighbor

The algorithm works by choosing an arbitrary node to start. In each step, a node that is not in the path and is the closest to the node last added into the path is then chosen and added into the path. The last step is to connect the last node with the starting node to form a cycle. The time complexity of such a algorithm is $O(n^2)$. (Rosenkrantz et al., 1977)[18]

In terms of the TSP, the nodes are each city traveled. The nearest neighbor algorithm implemented by the group starts at City 0, and proceeds to look at the distance between the current city and cities not yet in the path. The city that is the shortest distance away from the current city is then added to the path. This step is then repeated until all cities are in the path.

We obtained L=28.9 with the tour that started at city 0 and the second last city was , while the time needed to run the algorithm is at 1.51 seconds.

2.3.2 Insertion Algorithm

The basic idea of the algorithm is to construct an approximate path first before inserting a node in the approximate path in each step of the algorithm
The steps of the insertion algorithm is as follow:

1) Selection step: to select a node based on a certain criteria that we can use to insert in the approximate path.
2) Insertion step, where we decide which position to insert the selected city into.

The algorithm stops once all nodes are included in the path.
Two insertion methods that the group used for the problem are the Nearest and cheapest insertion methods.

2.3.2.1 Nearest insertion

For the nearest insertion method, the selection step will select the node that is closest to any node currently in the path.
For the insertion step, it looks at the current approximate path and finds the adjacent nodes I and J, where the node to be inserted, node K will have a minimal insertion cost. Insertion cost is defined as:

$$distance(node\ I,\ node\ R)\ +\ distance(node\ R\ ,node\ J)\ -\ distance(node\ I\ ,\ node\ J)$$

Time complexity of this algorithm is $O(n^2)$. (Rosenkrantz et al., 1977)[18]

The group went through all possible starting cities (from city 0 to city 999). Initial approximate path to start with is chosen as the path between the starting city and the city that is the nearest to the starting city (shortest distance away from the starting city).
In each step of the algorithm, the selection and insertion step is performed as described above, where the nearest city to any city in the path is selected, and its position to be inserted into is based on the minimal insertion cost.

The approximate time needed to loop through 1 instance of the algorithm is around 15 seconds. In our code, we have looped through all possible starting cities from city 0 to city 999, time needed to perform this is 4 hours 30 mins .
The best city to start from is city 588, where we obtained L= 28.7.

2.3.2.2 Cheapest Insertion

The cheapest insertion method is a more accurate version of the nearest insertion method. The selection step of the algorithm will calculate the insertion costs for all nodes that are not in the approximate path for all possible insertion positions, and select the node that leads to the minimal insertion cost. The insertion step is the same as the nearest insertion where the selected city is inserted in the position with the minimum insertion cost.
The time complexity of this method is $O(n^2 log(n))$. (Rosenkrantz et al., 1977)[18]

Since the cheapest insertion method is an extension of the nearest insertion method and it takes a longer time to run as compared to the nearest insertion method, we have only used the algorithm with 1 starting city. The initial approximate path is constructed in the same way as the nearest insertion method, which is the path between the starting city and the city that is the nearest to the starting city (shortest distance away from the starting city).
In each step of the algorithm, we computed the insertion cost for each city not in the approximate path at each insertion position, and selected the city to be inserted in the particular insertion position that leads to the smallest insertion cost.

The starting city is chosen as city 588 since it leads to the smallest L in the nearest insertion method. We obtained L=27.7, which is lower than the L in the nearest insertion method, showing that the cheapest insertion method is an improvement of the nearest insertion method. The time needed to run the algorithm is about 4461 seconds.

2.3.3 Greedy Algorithm

The algorithm makes the choice of the next city based on a certain criteria in order to obtain the optimal solution to the problem. In particular, Dijkstra's Shortest Path Algorithm (Russell & Cohn, 2012)[19] is used where we start at an arbitrary node. The criteria to choose the next node to visit will be based on the shortest distance to the current node. The time complexity is $O(n^3)$.

Implementation on TSP is where we choose the next city to visit based on the shortest distance to the current city. We apply the algorithm on all possible starting cities (from city 0 to city 999) to find the starting city that leads to the smallest total distance L.

The starting city that produces the lowest L is city 484, where we obtained L=27.3. Time needed for the algorithm to run is 3786 seconds.

2.3.4 Christofides Algorithm

The Christofides Algorithm is an approximation method that is referred to as a 1.5 approximation method as the final cost of the solution is at most 1.5 times the lower bound of the lowest cost one can get, based on the minimum spanning tree (MST).

In the context of TSP, the algorithm is as follows (Bo'sBlog, 2011)[2]:

1. Find a MST either using Prim or Kruskal, in our case we used Prim.
2. Let O be the set of vertices with odd degrees in the MST.
3. Find a minimum cost perfect matching that we will refer to as M on O.
4. Shortcut the Eurelian tour.

To elaborate on what the algorithm is doing, the algorithm first finds a MST which is a graph with the lowest total distance between cities that contains all the cities. This was done using the Prim's algorithm which essentially starts from a single city and iteratively adds the shortest distance paths that are connected to the updated path and not already in the graph.
Next O is obtained by selectively picking out cities that have an odd number of edges joined to them in the MST. Afterwards perfect matching is done where we selectively pick out minimum edges to connect O such that the edges contain each node in O at least once, this gives M.
Next to shortcut the Eurelian tour, we connect the edges derived in M to T and then we remove all but the first occurrence of each node to get the final graph.

The L that was obtained was L = 26.53 and it took 124.6 seconds to run. The path starts from city 5 and the second last city was 242. Note that the city that we choose to start from here is determined by the city we chose to start from in Prim's algorithm when deriving the MST.

3. Conclusion

It can be concluded that for *cities.npy*, the Lin-Kernighan heuristic together with the 2-opt algorithm yielded the lowest L at 23.461 in a reasonable time frame. The permutation of cities to obtain L = 23.461 can be found in the Jupyter notebook for the LKH algorithm. Future work that can be done to get a better optimization of total cost could be to layer a 2-opt algorithm on other heuristic algorithms, or explore other algorithms such as Particle Swarm Optimization.

References

1.Abdulkarim, H., & Alshammari, I. (2015). Comparison of Algorithms for Solving Traveling Salesman Problem. International Journal of Engineering and Advanced Technology (IJEAT), 4(6), 2249–8958. https://www.ijeat.org/wp-content/uploads/papers/v4i6/F4173084615.pdf

2.Bo'sBlog. (2011). Traveling Salesman Problem and Approximation Algorithms. Bochang.me. https://bochang.me/blog/posts/tsp/

3.BraveDistribution. (2023, March 7). pytsp. GitHub. https://github.com/BraveDistribution/pytsp/blob/master/pytsp/genetic_algorithms/simulated_annealing.py

4.Brilliant.org. (2016a). Dijkstra's Shortest Path Algorithm | Brilliant Math & Science Wiki. Brilliant.org. https://brilliant.org/wiki/dijkstras-short-path-finder/

5.Brilliant.org. (2016b). Greedy Algorithms | Brilliant Math & Science Wiki. Brilliant.org. https://brilliant.org/wiki/greedy-algorithm/

6.Dimitrovski, F. (2023, April 19). elkai - a Python 3 TSP solver. GitHub. https://github.com/fikisipi/elkai

7.Dorigo, M., & Gambardella, L. M. (1997). Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation, 1(1), 53–66. https://doi.org/10.1109/4235.585892

8.ezstoltz. (2023, April 11). genetic-algorithm. GitHub. https://github.com/ezstoltz/genetic-algorithm

9.Geek for Geeks. (2013, November 4). Approximate solution for Travelling Salesman Problem using MST. GeeksforGeeks. https://www.geeksforgeeks.org/approximate-solution-for-travelling-salesman-problem-using-mst/

10.Giraud, V. (n.d.). Travelling Salesman Problem (TSP): Concept – datascience.lc. DATASCIENCE.LC. Retrieved April 20, 2023, from https://datascience.lc/travelling-salesman-problem-tsp-concept/

11.Induraj. (2023, February 23). Implementing Ant colony optimization in python- solving Traveling salesman problem. Medium. https://induraj2020.medium.com/implementation-of-ant-colony-optimization-using-python-solve-traveling-salesman-problem-9c14d3114475

12.Kuo, M. (2020, January 1). Solving The Travelling Salesman Problem For Deliveries | Routific. Blog.routific.com. https://blog.routific.com/blog/travelling-salesman-problem

13.Lin, S., & W, K. B. (1973). An Effective Heuristic Algorithm for the TravelingSalesman Problem. Operations Research, 21(2), 498–516. https://doi.org/10.1287/opre.21.2.498

14.Neumann, F., Sudholt, D., & Witt, C. (2009). Computational Complexity of Ant Colony Optimization and Its Hybridization with Local Search. In C. P. Lim, L. C. Jain, & S. Dehuri (Eds.), Innovations in Swarm Intelligence (pp. 91–120). Springer Berlin Heidelberg. https://doi.org/10.1007/9783642042256_6

15.Nikolaev, A. G., & Jacobson, S. (2010). Simulated Annealing. In Handbook of Metaheuristics (Vol. 146, pp. 1–39). https://doi.org/10.1007/9781441916655_1

16.Nuraiman, D., Fadilah Ilahi, Yushinta Dewi, & Dzaki, A. (2018). A new hybrid method based on nearest neighbor algorithm and 2-Opt algorithm for traveling salesman problem. In 2018 4th International Conference on Wireless and Telematics (ICWT) (pp. 1–4).

17.Pandit, R. (2023, March 20). Traveling Salesman Problem. GitHub. https://github.com/rohanp/travelingSalesman

18.Rosenkrantz, D. J., Stearns, R. E., & II, L. (1977). An Analysis of Several Heuristics for the Traveling Salesman Problem. SIAM Journal on Computing, 6(3), 563–519. ABI/INFORM Collection. https://doi.org/10.1137/0206041

19.Russell, J., & Cohn, R. (2012). Dijkstra's algorithm. Book on Demand. https://books.google.com.sg/books?id=LjPtMgEACAAJ

20.Sasaki , G. H. (1987, October 1). Optimization by Simulated Annealing: A Time-Complexity Analysis. Apps.dtic.mil. https://apps.dtic.mil/sti/citations/ADA185547#:~:text=Our%20results%20indicate%20th at%20if

21.Stack Overflow. (n.d.-a). 2-opt algorithm to solve the Travelling Salesman Problem in Python. Stack Overflow. Retrieved April 20, 2023, from https://stackoverflow.com/questions/53275314/2-opt-algorithm-to-solve-the-travelling-sa lesman-problem-in-python

22.Stack Overflow. (n.d.-b). python - Travelling Salesman in scipy. Stack Overflow. Retrieved April 20, 2023, from https://stackoverflow.com/questions/25585401/travelling-salesman-in-scipy

23.Wei, X. (2014). *Parameters Analysis for Basic Ant Colony Optimization Algorithm in TSP. International Journal of U-and E-Service, 7(4), 159–170. https://doi.org/10.14257/ijunnesst.2014.7.4.16*