DSA4212 Assignment 3
Group 60

Nicholas Russell Saerang, A0219718W, e0550376@u.nus.edu
Clifton Felix, A0219735X, e0550393@u.nus.edu
Jason Ciu Putra Sung, A0219734Y, e0550392@u.nus.edu

1 Introduction

Given the 2D coordinates of 1000 cities, we want to find a permutation of these cities such that the cost of the tour that visits each city in these order and comes back to the initial city is minimized. The fastest known exact algorithm runs in exponential time, which is not feasible in this context. However, there are many methods that can be used to approximate this problem. Therefore, we will explore various approximation algorithms to solve this problem, also known as the Travelling Salesman Problem (TSP).

2 Approaches

2.1 Greedy

The simplest algorithm that we can use as a benchmark is a greedy algorithm. It loops from the first city to the last city, and for each city, we add the shortest edge that connects the city to another city that has not been visited, i.e. not part of our partially constructed tour. With this simple algorithm, we were able to produce a tour of cost **28.94**. Below is the pseudocode of the greedy algorithm, referenced from [kat].

```
set tour[0] = 0
set used[0] = true
for i = 1 to n-1:
  set best = -1
  for j = 0 to n-1:
    if not used[j]:
      if best = -1 or
        D(tour[i-1],j) <
          D(tour[i-1],best):
          set best = j
  set tour[i] = best
  set used[best] = true
return tour
```

2.2 MST

The fundamental principle of the minimal spanning tree (MST) method is to first generate the MST of the complete graph of the cities, and then utilize the tree to form a tour that visits each city exactly once. The MST is constructed using Prim's algorithm and depth-first search (DFS) is used to traverse the resulting MST. Lastly, we added an edge from the last visited city back to the starting city. This MST approach normally runs in $O(n^2 \log n)$ time complexity, where n is the number of cities, but due to the graph being complete, i.e. dense, we can use the dense variant of the MST algorithm which runs in $O(n^2)$ time. It takes 3.1 seconds to run this approach locally, which is quite fast. Using the minimum spanning tree, we were able to produce a tour of length **32.13**, which is worse than the greedy algorithm. However, since the MST approach is a 2-approximation algo-

rithm, we know that the optimal tour cost will not be lower than half of what we got.

2.3 Simulated Annealing

According to [wik23b], simulated annealing is a meta-heuristic algorithm that is based on the physical process of annealing, where a material is heated and then slowly cooled to reach a low-energy state. It is one of the approximation algorithms for TSP that heavily depends on randomized improvement and local search. In the context of TSP, simulated annealing starts with an initial tour and iteratively improves by changing the tour randomly, and accepting these changes based on a probability distribution. A temperature parameter that controls the probability of accepting a sub-optimal solution directs the algorithm and enables it to escape local optima.

Specific to the context of our experiment, we set our initial temperature to be $T_0 = 10^7$ and our hyperparameter $\alpha = 0.99$. For the next 1 million iterations, in every iteration k, we flip a randomly chosen contiguous subsequence of our current TSP tour and check if the cost c_1 improves from our initial tour cost c or not. If it does, we move on with this new configuration. Otherwise, we accept the less optimal tour with a probability of $e^{\frac{c-c_1}{T_k}}$. Before going to the next iteration, we update our temperature $T_{k+1} = \alpha T_k$.

Using $[0, 1, \cdots, 999]$ as our initial tour leads to a final tour cost of **29.93**. However, if we start the simulation with the greedy tour as our initial configuration, we end up with a slightly better result, which is **29.43**. Running one million iterations takes about 10-12 minutes, which is way longer than the previous MST approach.

2.4 Christofides' Algorithm

Christofides' algorithm, according to [wik23a], is another approximation algorithm for the Travelling Salesman Problem (TSP) that guarantees a solution within a factor of 3/2 of the optimal solution. The algorithm was proposed by Nicos Christofides in 1976.

The steps involved in this algorithm are as follows:

1. Construct the MST of the complete graph of the cities
2. Construct a set of odd-degree vertices
3. Compute a minimum-weight perfect matching on the set of odd-degree vertices
4. Combine the MST and the minimum-weight perfect matching into a multigraph
5. Find an Eulerian tour of the multigraph
6. Generate the TSP tour by traversing the Eulerian tour and skipping any repeated vertices, except for the start and end vertices

The time complexity of Christofides' Algorithm for TSP is $O(n^4)$, where n is the number of cities. This is dominated by the time required to compute the minimum-weight perfect matching, which has a time complexity of $O(n^3)$ using the Hungarian algorithm, and the time taken to compute the Eulerian tour, which is $O((n^2)^2) = O(n^4)$.

Since the algorithm takes multiple steps, we decided to use the black-boxed version of this algorithm, provided by NetworkX at [chr]. The algorithm runs in about 2 minutes and produces a tour of cost **26.6**, which is a new improvement across all the methods we have tried so far.

2.5 k-opt Heuristic

The k-opt heuristic, also known as the Lin-Kerninghan heuristic, is a technique in solving TSP that selects k mutually disjoint edges, deletes them, and reconnects the affected vertices in another way to form a shorter tour.

The k-opt method implemented in our experiment involves multiple values of k, where we perform each one of them repeatedly in a particular order. Various source codes such as [Nga] and [Loo] allow us to understand more about how each of these methods performs and optimizes the current configuration of the TSP tour.

2.5.1 4-opt

This movement is also known as the "double bridge" move. As the name of the method suggests, it takes four random edges, removes them, and reassembles the tour in some other way.

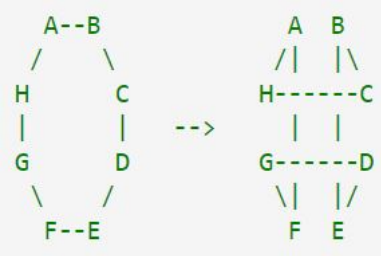

Figure 1: The 4-opt method

Using the figure above as an example, we remove edges AB, CD, EF, and GH. Then, we reassemble the tour by connecting the edges AF, BE, CH, and DG. It might produce a higher tour cost, but similar to simulated annealing, it allows us to escape local minima, hoping the other opt methods will be the ones that bring the cost all the way down to its minima.

The implementation is rather simple in code. We select 3 random integers from 1 to $\frac{n}{4} = 250$ inclusively, denoted as a, b, and c, and cut the tour at points 0, a, $a + b$, and $a + b + c$, and reassemble them using list slicing operators in Python, or vector copying in C++.

2.5.2 2-opt + tabu search

The 2-opt technique, also known as the pairwise exchange technique, involves removing two edges and replacing them with crossing edges, supposedly forming a shorter tour. According to [Joh97], this heuristic is on average slightly better than Christofides' algorithm, which can be improved if the starting tour is the greedy tour.

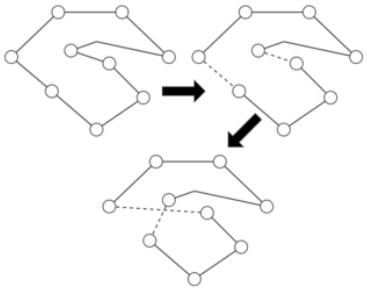

Figure 2: The 2-opt method

In addition to the pairwise exchange technique, we also employed another technique known as tabu search. According to [wik23c], tabu search is a metaheuristic search method similar to local search where given a potential solution x, one can move into an improved solution x^* as a successor state of x until a certain score threshold is satisfied. The word "tabu" indicates things that are sacred and thus cannot be touched. Similarly, when we found a potential solution but we have yet to reach a certain tabu threshold that acts like an expiration date, we are unable to use the potential solution until such threshold is met.

Specific to our experiment, there are three hyperparameters that made up the whole 2-opt and tabu search technique.

- `asp_ratio`, which determines the threshold ratio between the current best cost and the newly found tour cost in order to accept as a new best solution regardless of whether it's below the tabu threshold or not. For this experiment, this value is set to 15.
- `tabu_tenure`, the tabu threshold for a particular solution. This will be decremented over time and once the value reaches zero, the same solution may be

revisited again. For this experiment, this value is set to 5.

- `niter`, the number of iterations the tabu search should be done. For this experiment, this value is set to 5.

2.5.3 2.5-opt

The 2.5-opt technique, although may sound peculiar due to the ".5" term, also plays an important role in the overall k-opt method. The workflow is as follows:

- Choose three consecutive cities in the tour and two other consecutive cities in the same tour.
- Call the first three A, B, C, and the next two D and E.
- Initially, AB, BC, and DE are existing edges that are part of the tour. Remove them and replace them with edges AC, BD, and BE.
- Set this tour as the new tour if its cost is lower than the previous tour cost.

2.5.4 3-opt

The 3-opt technique is the most well-known technique among all parts of the k-opt method. This is actually a more general version of the 2.5-opt method since it involves three separate pairs of cities instead of just five cities, where the two pairs actually have a common city. Below is a visualization of how the 3-opt method works.

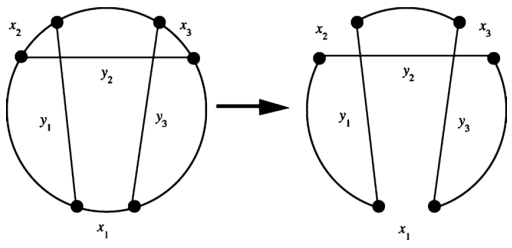

Figure 3: The 3-opt method

The workflow of the 3-opt method is as follows:

- Choose three edges that are not adjacent to each other, say AB, CD, and EF.
- Remove these edges and see if one of these set of replacement edges may shorten the tour:
  - AD, BE, and CF
  - AE, BD, and CF

  Due to the small value of the dataset, some minor tweaks are required in this step, which is to accept the new best cost only if the margin of improvement is larger than 5×10^{-8}.

Starting with the greedy tour, the final k-opt method consists of running the 4-opt, 2-opt, 2.5-opt, and 3-opt methods repeatedly in that particular order until there is not much improvement. After running each method for 10 iterations, we were able to produce a TSP tour of cost **24.78**, which is indeed an improvement from Christofides' algorithm. Alternatively, we also ran the same algorithm in C++, with various time limits between 5 and 20 seconds, and ended up with the best tour cost of **24.59**.

2.6 Ant Colony Optimization

Ant Colony Optimization (ACO) is a metaheuristic algorithm inspired by the behavior of ants in finding the shortest routes between their colony and food sources. It works by modeling the ant colonies' behavior of looking for food. Each ant in the colony represents a potential solution to the problem, and the colony as a whole search for the best solution by depositing pheromone trails on the edges of the graph representing the problem instance. The pheromone level on an edge represents the quality of the edge and is updated by the ants as they move through the

cities. Over time, the pheromone level on the edges that make up good solutions tends to increase, while the pheromone level on the edges that make up poor solutions tends to decrease. This leads to a positive feedback loop in which ants are more likely to choose edges that are part of good solutions and less likely to choose edges that are part of poor solutions.

By referring to the source code in [Ppo], we created an Ant Colony Optimization solver for TSP. After over 101 iterations or generations, which took about 24 minutes, it managed to reach a tour cost of **29.28**, which is not better than the k-opt method and even slower than it.

To be more specific, here's how the ACO solver works according to [Yam]:

- Ants are initialized in random cities. We set the hyperparameter that controls the number of ants to 8.
- Each of the ants moves along the graph and traverses through all cities. An ant moves from city i to city j with probability

$$P_{ij}(t) = \frac{\tau_{ij}^{\alpha} + \eta_{ij}^{\beta}}{\sum \tau^{\alpha} + \eta^{\beta}}$$

  where τ_{ij} is the amount of pheromone along the edge ij and η_{ij} is the inverse of the length of edge ij, i.e. $\eta_{ij} = \frac{1}{d_{ij}}$. Initially, the values of τ are all $\frac{1}{n^2}$. α and β are control hyperparameters, which we set to 1.0 and 9.0, respectively.
- As an ant moves from one city to another, it leaves a trail of pheromones for the next ants to follow. This means the more ants following this path, the stronger the pheromone amount. For each ant, it leaves $\Delta\tau_{ij} = \frac{1}{d_{ij}}$ units of pheromone. Our pheromone formula becomes as follows.

$$\tau_{ij}^{k*} = \tau_{ij}^{k} + \Delta\tau_{ij}$$

- Pheromones also tend to evaporate. We multiply all pheromone amounts in each edge by another hyperparameter called the evaporation rate, known as ρ, which we set such that $1 - \rho = 0.5$. Our final pheromone formula becomes as follows.

$$\tau_{ij}^{(k+1)} = (1 - \rho)\tau_{ij}^{k*}$$

  Note that the last two steps are interchangeable based on personal preferences. We decided to evaporate the pheromones first before adding the pheromone trail because that is how [Ppo] implements the ACO solver.

While ACO does not give us the best solution, it gives us another example of how randomized improvement works in solving TSP, specifically on the part where the ant chooses a random unvisited neighbor with a certain probability.

2.7 PyConcorde

The last approach is PyConcorde, which wraps the C-language Concorde TSP solver into its Python equivalent. According to [con], Concorde has been around for around a decade, consisting of hundreds of functions available to deal with TSP problems. It has been used to get optimal tours for 110 other TSP instances with the maximum number of cities being 85,900, as shown in [ABC+09]. Other than solving TSP, Concorde is also used to perform gene mapping, vehicle routing, and many more as stated in [wik21].

Based on an anonymous comment on [red], Concorde uses an advanced version of the Lin-Kernighan heuristic called the Chained Lin-Kernighan approximation, along with other optimization methods such as the branch-and-bound method and the branch-and-cut method. The heuristic on its own has been proven to have a very small cost margin to the optimal tour cost, as shown in [ACR03], where it solves a TSP instance of 14,051 cities

with a cost gap of 0.003% and is able to solve the one with 15,112 cities optimally, i.e. 0% cost gap.

The branch-and-bound method, in summary, breaks down a problem into smaller sub-problems and makes use of a bounding function to eliminate sub-problems that do not have the optimal solution. On the other hand, the branch-and-cut method uses the branch-and-bound method alongside the plane-cutting technique to tighten the linear programming constraints. The details of these techniques are omitted because they have been black-boxed within Concorde itself.

In addition, [LHW20] states that Concorde uses the QSopt linear programming solver, which according to [qso] is another framework that claims to be an exact linear programming solver, to form an exact computation algorithm for the TSP. Note that TSP can be represented as an integer linear programming problem, as described in [wik23d]. Therefore, we can say that Concorde is an exact TSP solver.

According to [LHW20], the exact runtime of Concorde is yet to be known, but rather the observed runtime, whose median is $O(ab^n)$, where $a \approx 0.21, b \approx 1.24$, and n is the number of cities.

In our experiment, we used PyConcorde from [Jvk], and unsurprisingly after about a minute we managed to get a relatively low tour cost of **23.32**, which is the smallest tour cost that we were able to get for this problem.

The PyConcorde library itself is pretty handy for general users. It makes use of the `TSPSolver` class and we can construct the TSP instance from a `.tsp` file or from a list of coordinates. After solving, it returns information on whether the algorithm converges with an optimal tour or not, as well as the sequence of cities to visit, indicating the optimal tour.

3 Conclusion

Algorithm	Cost
Greedy	28.94
MST	32.13
Simulated Annealing	29.43
Christofides' Algorithm	26.60
k-opt Heuristic	24.59
Ant Colony Optimization	29.28
PyConcorde	23.32

Table 1: Algorithm Performance

All algorithms' performances are shown in Table 1 above. While the experiments concluded with PyConcorde as the framework that produces the best solution for the TSP problem with a tour cost of **23.32**, there are many other methods yet to be mentioned in this report, such as the genetic algorithm, bitonic TSP, cross entropy method, and the convex hull method. More details of the recently mentioned methods are available on [wik23d].

4 Bibliography

[ABC+09] David L Applegate, Robert E Bixby, Vašek Chvátal, William Cook, Daniel G Espinoza, Marcos Goycoolea, and Keld Helsgaun. Certification of an optimal tsp tour through 85,900 cities. *Operations Research Letters*, 37(1):11–15, 2009.

[ACR03] David Applegate, William Cook, and André Rohe. Chained lin-kernighan for large traveling salesman problems. *INFORMS Journal on Computing*, 15(1):82–92, 2003.

[chr] Christofides. URL: https://networkx.org/documentation/stable/reference/algorithms/

generated/networkx.algorithms.approximation.traveling_salesman.christofides.html.

[con] Concorde tsp solver. URL: https://www.math.uwaterloo.ca/tsp/concorde/.

[Joh97] David S Johnson. A case study in local optimization. *Local search in combinatorial optimization*, pages 215–310, 1997.

[Jvk] Jvkersch. Jvkersch/pyconcorde: Python wrapper around the concorde tsp solver. URL: https://github.com/jvkersch/pyconcorde.

[kat] Travelling salesperson 2d. URL: https://open.kattis.com/problems/tsp.

[LHW20] Yongliang Lu, Jin-Kao Hao, and Qinghua Wu. Solving the clustered traveling salesman problem via tsp methods. *arXiv preprint arXiv:2007.05254*, 2020.

[Loo] Lookuz. Lookuz/cs4234-stochastic-local-search-methods. URL: https://github.com/Lookuz/CS4234-Stochastic-Local-Search-Methods/tree/master/TSP.

[Nga] Ngamanda. Ngamanda/kattis-tsp. URL: https://github.com/ngamanda/Kattis-TSP/blob/master/main.cpp.

[Ppo] Ppoffice. Ppoffice/ant-colony-tsp: Solve tsp using ant colony optimization in python 3. URL: https://github.com/ppoffice/ant-colony-tsp.

[qso] Qsopt linear programming solver. URL: https://www.math.uwaterloo.ca/~bico/qsopt/.

[red] R/compsci - how does concorde claim to be a tsp solver? URL: https://www.reddit.com/r/compsci/comments/8auwm9/how_does_concorde_claim_to_be_a_tsp_solver/.

[wik21] Concorde tsp solver, Aug 2021. URL: https://en.m.wikipedia.org/wiki/Concorde_TSP_Solver.

[wik23a] Christofides algorithm, Jan 2023. URL: https://en.wikipedia.org/wiki/Christofides_algorithm.

[wik23b] Simulated annealing, Feb 2023. URL: https://en.wikipedia.org/wiki/Simulated_annealing.

[wik23c] Tabu search, Mar 2023. URL: https://en.wikipedia.org/wiki/Tabu_search.

[wik23d] Travelling salesman problem, Mar 2023. URL: https://en.wikipedia.org/wiki/Travelling_salesman_problem.

[Yam] Yammadev. Implementing ant colony optimization (aco) algorithm for a given symmetric traveling salesman problem (tsp). URL: https://github.com/yammadev/aco-tsp/blob/master/aco-tsp.ipynb.