MANG6542: Computational Methods for Logistics

INDIVIDUAL COURSEWORK

THE “QUASI” TRAVELING SALESMAN PROBLEM

Word count: 1436

Student ID: 33815232

TABLE OF CONTENTS

I. Literature Review on Traveling Salesman Problem

The Travelling Salesperson Problem (TSP) is the one of the most prominent NP-hard combinatorial optimization problems (Sathya & Muthukumaravel, 2015). Several researchers have studied this problem for decades, and many solutions have been proposed, which are implemented in various situations including warehousing, material handling, and building planning (Kim et al., 1998). TSP data consists of a finite number of cities and the cost of travel between each pair of them; the goal is to find the shortest Hamiltonian cycle - a cycle that passes through all of the cities once and returns to the original point of departure. For the symmetric TSP case, which is also the focus of this report, the distance is the same regardless of the direction of travelling (Cummings, 2000).

TSP is an NP-hard problem, considering that the time complexity of TSP is not polynomial but exponential with a complexity of O(n!) (where n is the number of nodes). This also implies that if a more significant number of cities are inspected, evaluating every possible solution within a "reasonable" time frame is infeasible (Hayes, 2022). This report discuss both exact and approximate solutions to TSP. While the TSP exact solutions is considered unpractical by trying all the permutation combinations, the approximate heuristic algorithms help reducing the complexity. This approximation approach might not generate an minimum tour length, however, by providing the answer in a shorter amount of time compared to the optimum one, it has been implemented extensively (Abdulkarim & Alshammari, 2015).

1. Optimal algorithm for the TSP

The Brute-force algorithm is the most straightforward solution since it tests every possible city permutations to achieve the shortest tour. However, this algorithm is suggested to be the most time-consuming and expensive method (Rajarajeswari & Maheswari, 2020) as finding the exact solution to a TSP with n cities requires checking (n - 1)! possible tours. Alternatively, exact methods can be used to generate optimal results. A branch and bound (BB) algorithm, which is proposed by Land & Doig (1960), is a well-known algorithm of this kind. When compared with Brute-force algorithm, BB algorithm is better as without fully investigating each branch or relying on any form of randomness, this approach always find the optimal solution by spliting all feasible tours into subsets, followed by calculating the lower bound of each subset to remove the unpromissing subsets (Chatting, 2019). However, the time complexity is not efficently reduced in this method. As a potential substitute, heuristics or approximate methods are frequently used for solving TSP to improve speed in finding the solution and closeness to the optimal solution.

2. Heuristics for the TSP.

Generally, TSP heuristics can be classified into two groups: tour construction heuristics and tour improvement heuristics. The former creates tours from scratch, based on including points in the tours (usually one at a time) until a complete tour is developed (Kim et al., 1998), while the latter uses simple local search heuristics to improve existing tours by transposing two or more points in the initial tour (Hahsler & Hornik, 2007). This report analyzes the state-of-art approaches for both of these classifications.

2.1. Tour construction heuristics

Nearest neighbors algorithm selects each node's closest neighbour until all nodes have been visited, then finish the tour by linking back up with the initial node (Rosenkrantz et al., 1977). The worst-case performance of this algorithm results in an O(n2) runtime, which is obviously inefficient because it gives no consideration to the final relinking step and may lead to a local solution with a very long edge to the depot (Mahéo, 2017). Even though this is often regarded as a poor running time for programming problems, it is still faster than the factorial runtime from the Brute-force method.

2.2. Tour improvement heuristics

State-of-the-art TSP solvers are built on local search, with the exception of the optimal ones (Mariescu-Istodor & Fränti, 2021). The idea is to have an initial solution and then apply a trial-and-error manner by using local operators to see if the cost decreases. The operator determines how to alter the existing tour to produce a new optimal solution. For instance, the most prevalent operator, 2-opt (Croes, 1958), identifies two links and swaps them between the nodes. This approach can enhance the greedy algorithm's minimum tour length and is significantly faster than the BB algorithm. Other advantages of the 2-opt algorithm include the feasibility of all 2-opt moves when considering a complete graph, the edge selection without requiring complex selective heuristics, the simplicity of the swap operation on tours, and the ease with which it can be determined whether a move will enhance a tour. Furthermore, a hybrid algorithm of 2-opt and another exact method for smaller size TSP ($n<50$) is reported to perform with high optimality and little computational time (Vijaykumar et al., 2014). All of these advantages can be obtained with relatively modest implementation costs.

II. Instances for the TSP

The number of given nodes in TSP code has increased significantly, from Dantzig, Fulkerson, and Johnson's solution of a 49-city problem in 1954 to the solution of an 85,900-point problem in 2006 from TSPLIB. This report only focuses on solving 3 datasets using heuristic algorithm.

Table 1 summarize the datasets for three specific problem, imported from Reinelt's TSPLIB.

Table 1. Datasets summary

Item	**Problem 1**	**Problem 2**	**Problem 3**
Name	Bays29	Berlin52	Gr120
Number of nodes	29 nodes	52 nodes	120 nodes
Minimal tour length	2020	7542	6942
Sources	Groetschel, Juenger, Reinelt	Martin Groetschel	Martin Groetschel
Algorithm applied	NA	NA	Linear programming cutting plane
CPU time given by Concorde Benchmarks (from 1999)	0.13 seconds	0.29 seconds	2.23 seconds

III. Construction heuristic

1. Pseudo-code.

Quasi Traveling Salesman Problem (QTSP) is a variations of TSP, adding one condition: only one customer maybe ignored. The 2-opt method similar to other heuristic search algorithms is sensitive to its initial point in the search space (Pedram, 2021[3]). To minimize this sensitivity, different randomized initial routes is tested by running 10 permutations and selected the best result among them. Consequently, the outcome can vary in each iteration. The proposed pseudo-code can be described as follow:

Input: A distance matrix with diagonal filled with zeros and a list of original number of n nodes

Output: A 2-opt optimal route through n-1 nodes

Solver (Pedram, 2021[2]): 2-opt swapping operator

repeat

for i $\in$ n-1 cities eligible to be swapped do

for j $\in$ n-1 cities eligible to be swapped such that y > x

```
                initialize new_route
                    add route[1] to route[x-1]
                    add route[x] to route[y] in reverse order to new_route
                    add route[y+1] to end in order to new_route
                    calculate new_distance
                if
                    new_distance < best_distance
                then
                    best_route = new_route
                end if
            end for
        end for
until no improvement is made
```

RouteFinder (Pedram, 2021[1]): selected the best initial 2-opt route from 10 different randomized route

```
repeat 10 iterations:
    initialise initial_route (a randomized list start with node 1 to n-1 nodes)
    for each initial_route
        if new_distance < best_distance:
            best_distance = new_distance
            best_route = new_route
        end if
    end for
```

2. Coding results reporting

According to Appendix 2, the minimal tour length obtained by running the new code (Appendix 1) is obviously less than the reported cases for the berlin52 and bays29 instances, considering the elimination of one node. With a limited number of cities, the 2-opt heuristics can provide a better tour. However, for the gr120 dataset, the minimal distance exceeds the minimal tour length solved by the linear programming cutting plane approach. This aligns with the previous literature review on the heuristics approach: they can provide the acceptable tour length within an acceptable amount of time; however, the minimality of the answer is not guaranteed. Time complexity is also compared to the optimal approaches using these three instances. It is noticeable that the new code's time complexity is only around 7.69%, 17.24%, and 13% respectively. In further development, there are various methods to improve the optimality of the 2-opt algorithm without considerably increasing the time complexity. One instance is the hybrid method between nearest neighbor and 2-opt algorithm instead of random permutations to find a starting tour. While the nearest neighbor approach is fast and straightforward for maintaining a better initial tour, the 2-opt algorithm can improve the tour effectively (Nuraiman et al., 2018), thus escaping the local optimum.

Word count: 1436

REFERENCES

Abdulkarim, H., Alshammari, I., (2015). Comparison of Algorithms for Solving Traveling Salesman Problem. *International Journal of Engineering and Advanced Technology* 4(6).

Chatting, M. (2019) *A comparison of exact and heuristic algorithms to solve the travelling salesman problem*, *PEARL Home*. University of Plymouth. Available at: https://pearl.plymouth.ac.uk/handle/10026.1/14184 (Accessed: January 4, 2023).

Concorde Benchmarks (from 1999) (1999) *Concorde TSP benchmarks*. Available at: https://www.math.uwaterloo.ca/tsp/concorde/benchmarks/bench99.html (Accessed: January 2, 2023).

Croes, G.A. (1958) “A method for solving traveling-salesman problems,” *Operations Research*, 6(6), pp. 791–812. Available at: https://doi.org/10.1287/opre.6.6.791.

Cummings, N. (2000) *A brief History of the Travelling Salesman Problem*, *The OR Society*. Available at: https://www.theorsociety.com/about-or/or-methods/heuristics/a-brief-history-of-the-travelling-salesman-problem/ (Accessed: January 2, 2023).

Hahsler, M. and Hornik, K. (2007) “TSP—Infrastructure for the Traveling Salesperson Problem”, *Journal of Statistical Software*, 23(2), pp. 1–21. doi: 10.18637/jss.v023.i02.

Hayes, G. (2022) *Solving travelling salesperson problems with python*, *Medium*. Towards Data Science. Available at: https://towardsdatascience.com/solving-travelling-salesperson-problems-with-python-5de7e883d847 (Accessed: January 1, 2023).

Kim, B.I.,Shim, J.I, Zhang, M. (1998). *Comparison of TSP Algorithms*. Available at: https://pja.mykhi.org/4sem/NAI/rozne/Comparison%20of%20TSP%20Algorithms/Comparison%20of%20TSP%20Algorithms.PDF

Mahéo, A. (2017) *Local TSP heuristics in Python*, *Yet another optimisation blog*. Available at: https://arthur.maheo.net/python-local-tsp-heuristics/ (Accessed: January 1, 2023).

Mariescu-Istodor, R. and Fränti, P. (2021) "Solving the large-scale TSP problem in 1 H: Santa Claus Challenge 2020," *Frontiers in Robotics and AI*, 8. Available at: https://doi.org/10.3389/frobt.2021.689908.

Neoh, A. *et al.* (1970) *An evaluation of the traveling salesman problem*, *ScholarWorks*. California State Polytechnic University, Pomona. Available at: http://hdl.handle.net/20.500.12680/8g84mp499 (Accessed: January 2, 2023).

Nuraiman, D. *et al.* (2018) "A new hybrid method based on nearest neighbor algorithm and 2-opt algorithm for traveling salesman problem," *2018 4th International Conference on Wireless and Telematics (ICWT)* [Preprint]. Available at: https://doi.org/10.1109/icwt.2018.8527878.

Pedram, A. (2021) *Py2opt/routefinder.py at master · PDRM83/py2opt*, *GitHub*. Available at: https://github.com/pdrm83/py2opt/blob/master/py2opt/routefinder.py (Accessed: January 1, 2023).

Pedram, A. (2021) *Py2opt/solver.py at master · PDRM83/py2opt*, *GitHub*. Available at: https://github.com/pdrm83/py2opt/blob/master/py2opt/solver.py (Accessed: January 9, 2023).

Pedram, A. (2021) *Py2opt: How to solve the traveling salesman problem with the 2-opt algorithm.*, *PyPI*. Available at: https://pypi.org/project/py2opt/ (Accessed: January 1, 2023).

Rajarajeswari, P. and Maheswari, D. (2020) "Travelling salesman problem using branch and bound technique," *International Journal of Mathematics Trends and Technology*, 66(5), pp. 202–206. Available at: https://doi.org/10.14445/22315373/ijmtt-v66i5p528.

Reinelt, G. (1991) "TSPLIB—a traveling salesman problem library," *ORSA Journal on Computing*, 3(4), pp. 376–384. Available at: https://doi.org/10.1287/ijoc.3.4.376.

Rosenkrantz, D.J., Stearns, R.E. and Lewis, II, P.M. (1977) "An analysis of several heuristics for the traveling salesman problem," *SIAM Journal on Computing*, 6(3), pp. 563–581. Available at: https://doi.org/10.1137/0206041.

Sathya, N. and Muthukumaravel, A. (2015) "A review of the optimization algorithms on traveling salesman problem," *Indian Journal of Science and Technology*, 8(29). Available at: https://doi.org/10.17485/ijst/2015/v8i1/84652.

Vijaykumar, S., Shetty, S. and Menezes, S. (2014) "Comparative Analysis of Various Algorithms Used in Travelling Salesman Problem," *International Journal of Innovative Research in Computer and Communication Engineering*, 2(5).

APPENDICES

Appendix 1. Python code

```
import numpy as np

import tsplib95

import random2

import time

start = time.time()

# Use tsplib95 package to load data and create a complete distance matrix

def tsplib_distance_matrix(tsplib_file: str) -> np.ndarray:


    tsp_problem = tsplib95.load(tsplib_file)

    distance_matrix_flattened = np.array(

        [tsp_problem.get_weight(*edge) for edge in tsp_problem.get_edges()]

    )

    distance_matrix = np.reshape(

        distance_matrix_flattened,

        (tsp_problem.dimension, tsp_problem.dimension),

    )

    # A diagonal filled with zeros

    np.fill_diagonal(distance_matrix, 0)

    return distance_matrix
```

```
#2-opt Solver
class Solver:
    def __init__(self, distance_matrix, initial_route):
        self.distance_matrix = distance_matrix
        self.num_cities = len(self.distance_matrix)
        self.initial_route = initial_route
        self.best_route = []
        self.best_distance = 0
        self.distances = []

    def update(self, new_route, new_distance):
        self.best_distance = new_distance
        self.best_route = new_route
        return self.best_distance, self.best_route

    @staticmethod
    #calculates the total distance between the first city to the last city
in the given path.
    def calculate_path_dist(distance_matrix, path):
        path_distance = 0
        for ind in range(len(path) - 1):
            path_distance += distance_matrix[path[ind]][path[ind + 1]]
        return float("{0:.2f}".format(path_distance))

    @staticmethod
    def swap(path, swap_first, swap_last):
        path_updated = np.concatenate((path[0:swap_first],
                                       path[swap_last:-len(path) +
swap_first - 1:-1],
                                       path[swap_last + 1:len(path)]))
```

```
        return path_updated.tolist()



#2-opt function
    def two_opt(self, improvement_threshold=0.01):
        self.best_route = self.initial_route
        self.best_distance = self.calculate_path_dist(self.distance_matrix, self.best_route)
        improvement_factor = 1

        while improvement_factor > improvement_threshold:
            previous_best = self.best_distance
            for swap_first in range(1, self.num_cities - 3):
                for swap_last in range(swap_first + 1, self.num_cities - 2):
                    before_start = self.best_route[swap_first - 1]
                    start = self.best_route[swap_first]
                    end = self.best_route[swap_last]
                    after_end = self.best_route[swap_last+1]
                    before = self.distance_matrix[before_start][start] + self.distance_matrix[end][after_end]
                    after = self.distance_matrix[before_start][end] + self.distance_matrix[start][after_end]
                    if after < before:
                        new_route = self.swap(self.best_route, swap_first, swap_last)
                        new_distance = self.calculate_path_dist(self.distance_matrix, new_route)
                        self.update(new_route, new_distance)

            improvement_factor = 1 - self.best_distance/previous_best
        return self.best_route, self.best_distance, self.distances
```

```
# selected the best result from 10 different randomized initial points
class RouteFinder:
    def __init__(self, distance_matrix, cities_names, iterations=5,
writer_flag=False, method='py2opt'):
        self.distance_matrix = distance_matrix
        self.iterations = iterations
        self.writer_flag = writer_flag
        self.cities_names = cities_names

    def solve(self):

        iteration = 0
        best_distance = 0
        best_route = []

        while iteration < self.iterations:
            num_cities = len(self.distance_matrix)
            initial_route = [0] + random2.sample(range(1, num_cities-1),
num_cities - 2)
            tsp = Solver(self.distance_matrix, initial_route)
            new_route, new_distance, distances = tsp.two_opt()

            if iteration == 0:
                best_distance = new_distance
                best_route = new_route
            else:
                pass

            if new_distance < best_distance:
```

```
                best_distance = new_distance
                best_route = new_route

            iteration += 1

        if self.writer_flag:
            self.writer(best_route, best_distance, self.cities_names)

        if self.cities_names:
            best_route = [self.cities_names[i] for i in best_route]
            return best_distance, best_route
        else:
            return best_distance, best_route

#Input data: tsplib95 file, cities list, distance matrix to solve the problem
tsp_problem = tsplib95.load('Input file location')
cities_names = list(tsp_problem.get_nodes())
dist_mat = tsplib_distance_matrix('Input file location')
route_finder = RouteFinder(dist_mat, cities_names, iterations=10)
best_distance, best_route = route_finder.solve()

print('Minimal tour length: ' ,best_distance)
print('Best tour order: ', best_route)
end = time.time()
print('Time complexity: ',end-start)
```

Appendix 2. Results given by new algorithm

Item	Problem 1	Problem 2	Problem 3

Name	Bays29	Berlin52	Gr120
Number of nodes	29 nodes	52 nodes	120 nodes
Reported minimal tour length	2020	7542	6942
New minimal tour length	1892	7266	7084
Compared to optimal tour	93.66%	96.34%	102.05%
CPU time given by Concorde Benchmarks (from 1999)	0.13 seconds	0.29 seconds	2.23 seconds
CPU time with new code	0.01 seconds	0.05 seconds	0.29 seconds
Compared to optimal tour	7.69%	17.24%	13.00%