University of Southampton

Faculty of Engineering and Physical Sciences

Electronics and Computer Science

A new Hybrid method based on Hopfield Network and 2-opt to solve TSP

by

RunYing Jiang

February 2022

Supervisor: Richard Watson
Second Examiner: Shiyan Hu

A dissertation submitted in partial fulfilment of the degree
of MSC Artificial Intelligence

University of Southampton

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
ELECTRONICS AND COMPUTER SCIENCE

Master of Science

by RunYing Jiang

The travelling salesman problem is a famous mathematical combinatorial optimization problem and it is a classic NP-complete problem. For large-scale TSP problems, researchers are still unable to find a perfect solution that ensures to find the optimal solution. In addition, most search methodology can only be used with a specific type of TSP problem, which does not have a universal application value. This paper proposed a Hybrid Method combining the Hopfield Network and 2-opt Hill Climbing together. Experimental results show that compared with solely applying Hopfield Network, 2-opt Hill-Climbing and Genetic Algorithm, the hybrid method based on Hopfield network and 2-opt Hill-Climbing algorithm has a slight improvement in solving the TSP problem. The new hybrid method can be more guaranteed to get the global optima in solving the TSP problem. Furthermore, the hybrid method has the potential in solving other variations of TSP problems which is more similar to the real-life problem.

Keywords : travelling salesman problem, Hopfield Network, 2-opt, Genetic Algorithm, Hybrid Method

Acknowledgements

First of all, I would like to thank my supervisor, Richard Watson. Richard made instructive comments and suggestions on the research direction of my thesis, gave me careful guidance on the difficulties and doubts I encountered in the process of research. I am especially grateful for the weekly meetings, which gave me the encouragement to finish this project during the special period of the epidemic.

In addition, I would also thank my friends and family for their great support and help on all my decisions. During the project, I felt very lonely and powerless, my friends always stand with me and gave me company. Special thanks for Naixing Wang, Tianhao Ye, Liqi Gu and Abdullah for the support through this year.

Also, thanks to the teachers who taught me over the past year and the authors of the references, and thank them for their academic efforts, which gave me inspiration and help in completing this paper.

Finally, I would thank all the readers and examiners on this paper, thank you for your time, hope you enjoy it and have fun!

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

I have not used any resources produced by anyone else.

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else.

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

ECS Statement of Originality Template, updated August 2018, Alex Weddell aiofficer@ecs.soton.ac.uk

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

The origins of the travelling salesman problem are unclear. A handbook of Voigt for travelling salesmen from 1832 mentioned the problem and included example tours through Germany and Switzerland, but did not contain any mathematical treatment.

The travelling salesman problem (also called the travelling salesperson problem or TSP) is a classic NP problem in combinatorial optimization. Because of its extensive practical application background in engineering, it has always been concerned by the majority of scientific researchers.

The travelling salesman problem asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?". In short, there are several cities on a map, the travelling salesman need to find a route which will visit all of the cities and only once for each city. TSP problem is for finding the shortest routes. It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research.

In recent years, researchers have tried to use various methods to solve TSP, but with the deepening of the understanding of TSP characteristics, attempts to use an exact algorithm to solve TSP research disappeared, replaced by various heuristic methods; The attempts to use a single method to solve the TSP problem were also decreasing, while the combination of multiple methods has gradually occupied the mainstream of research.

1.2 Project Aims and Objectives

The main aim of the project is to find whether the Hybrid method based on 2-opt with optimization of deep learning outperforms other methods in solving travelling salesman problems.

The following questions has been raised to explore further on this topic:

- How much the Hybrid method can improve in optimization?
- Why Hybrid method?
- What is the Limitation and Advantages of the Hybrid Method?
- What is the research value in the application of the Hybrid Method?

The Hybrid Method was designed and implemented to solve the Travelling Salesman problem (TSP), which was a special case of vehicle routing problem when m=1 and $Q = +\infty$, because in practice, TSP was much less difficult to solve than the VRP problem of the same size. The N-City TSP problem could be an n by n square matrix. To get a solution to the TSP problem, the problem was described by an energy function, where the lowest distance to all cities corresponds to the best path.

The Hybrid Method was a system that combined Hopfield Network and 2-OPT Hill Climbing Algorithm. The 2-opt Hill Climbing Algorithm was used to find the minimal energy of VRP, which was a local optimization of constraints, while the Hopfield Network was used to develop associated memory with the potential to optimize traditional methods for repetitive vehicle routing problems. Although Modares et al. (2006) suggested that deep learning did not provide strong optimizations for solving traditional VRP problems, it was very likely to perform well for repetitive vehicle routing problems, since the routing problems we were facing today were very similar to the problems of Yesterday which had been solved, we did not need to start from scratch.

In addition, to better demonstrate the advantages and disadvantages of the Hybrid Method, the performance of the Hybrid Method has been analyzed and compared to other approaches including the 2-opt Hill climbing algorithm only, Hopfield Network only, and evolutionary algorithms.

1.3 Structure of Report

The remaining part of the paper is organized as follow: **Chapter 2: Literature Review** introduces some variations of the TSP problem and the related work of methodology to solve these problems. The project mainly focuses on the metaheuristic methodology. **Chapter 3: Methodology** describes the methods I implemented in this project: 2-opt Hill Climbing, Genetic Algorithm, Hopfield Network and the Hybrid Method. **Chapter 4: Implementation and Analysis** presents the experiment results of the above methods and the performance has been compared and discussed in **Chapter 5: Discussion**. **Chapter 6** is the conclusion part of this paper.

1.4 Project Management

The Gantt chart in **Figure 1.1** show my timeline to work on this project.

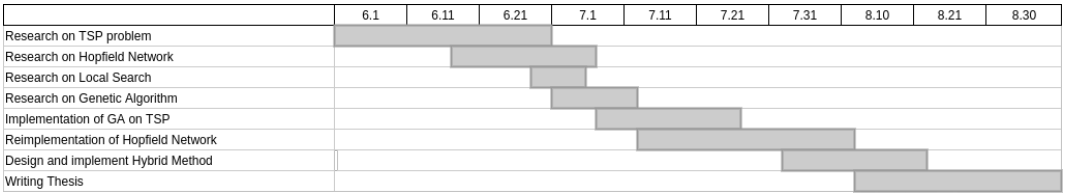


FIGURE 1.1: Gantt chart of Project

The initial idea of this project was to combine the evolutionary algorithm with deep learning methods together. However, from the experimental result, I found the evolutionary algorithm was not that effective as I expected in solving the TSP problems and I saw the possibility of the optimization on a local search method, so I designed the Hybrid Method based on the deep learning method and local search instead. Later, after getting in touch with more literature, I became interested in hybrid methods and dig deeper into the methodology and implementation of Hopfield Network. Most of the time of this project was spent on understanding the implementation and optimizing the algorithms.

Chapter 2

Literature Review

This section introduces the literature review from the following two fields: Extension of TSP problem and Related Work.

2.1 Extension of TSP problem

2.1.1 Multiple travelling salesman problem

TSP problem is a special case of multiple travelling salesman problem (mVRP). With some special additional constraints, it can be evolved into some practical problems, so it has higher theoretical research value. In the multi-travel salesman problem, a task is jointly completed by multiple travel agents, which is more difficult to solve than the classical travel salesman problem. The methods and strategies of the classical travelling salesman problem can not be simply applied to the solution of multiple traveling salesman problem.

Multiple travelling salesman problem can be divided into four types according to Qu et al. (2007):

- Travel agents travel from the same city to visit a certain number of cities. There is only one city center and each city must be visited by a certain travel agent. All the city can only be visited once and the travel agent should return to the original start.
- Travel agents start from different cities to visit a certain number of cities, so that each city must be visited by a certain travel agent only once, and finally back to their respective starting city.

- Travel agents start from the same city and visit a certain number of cities, so that each city must be visited by a certain travel agent and can only be visited once, and finally reach different cities.
- Travel agents start from different cities to visit a certain number of cities, so that each city must be visited by a certain travel agent and can only be visited once, and finally reach the same city.

The common mTSP is the first two kinds, for example, the delivery problem of express company is the first kind, and the delivery problem of national chain stores is the second kind. On the basis of general mTSP, some constraints will be added to evolve a variant mTSP.

The research on mTSP started in the 1970s, and at first it mainly used accurate algorithm to solve the problem. However, with the expansion of the problem scale, the spatial expansion speed of the problem solution has an exponential relationship with the problem size, which has become the main bottleneck restricting the performance of the accurate algorithm to solve the mTSP. Most research solutions are to first transform mTSP into TSP, and then solve the TSP based on the TSP solution algorithm.

2.2 Vehicle routing problem

The vehicle routing problem is a problem that asks "What is the best set of routes for a fleet to traverse in order to deliver to a given set of customers?" , which was first raised in G. B. Dantzig (1959). Thousands of distributors around the world face this issue every day, and it has important economic implications. Common examples are delivering milk, groceries, and newspapers. The problem was first raised 50 years ago, and many types of studies have been carried out to find the best way to solve the problem.

In real life vehicles, routes are much more complex, with many limitations, such as capacity and scheduling. If you want to handle more than 5,000 cars and 100,000 visits in a limited amount of time in a very crowded city, it's almost impossible to find a global optimal solution. Like Uber and Amazon has a very sophisticated way of doing this, which involves a complex combination of methods. Any small optimization can save a lot of time and manpower. In addition, the method to solve the vehicle routing problem can also be applied to distributed resource allocation fields such as grid. Or cloud computing. Our research is to see if there is a feasible way to optimize the traditional method, so as to save a lot of resources in real life, with the great progress of technology.

2.2.1 Capacitated vehicle routing problem

The capacitated vehicle routing problem in Ralphs et al. (2003), in which a fixed fleet of delivery vehicles of uniform capacity must service known customer demands for a single commodity from a common depot at minimum transit cost. This difficult combinatorial problem contains both the Bin Packing Problem and the Traveling Salesman Problem (TSP) as special cases and conceptually lies at the intersection of these two well-studied problems. The capacity constraints of the integer programming formulation of this routing model provide the link between the underlying routing and packing structures.

The capacitated vehicle routing problem consists of designing optimal delivery or collection routes from a central depot to a set of geographically scattered customers, subject to various constraints, such as vehicle capacity, route length, time windows, precedence relation between customers, etc. It can be represented by a complete weighted directed graph G=(V,A,d), where V is a set of vertices and A(vi,vj) is a set of arcs. The vertex v0 denotes the central depot and the other vertices of V represent customers. The problem consists of determining a set of m vehicle routes (1) starting and ending at the depot, and such that (2) each customer is visited by exactly one vehicle (3) the total demand of any route does not exceed Q. The objective of this problem is to find minimum cost vehicle routes with some specific constraints.

2.2.2 Repeated vehicle routing problem

The repeated vehicle routing problem is an extension of the Vehicle routing problem (VRP), which repeatedly builds solutions to minimize the energy of the system. Vehicle routing issues include designing optimal delivery or collection routes from a central warehouse to a geographically dispersed set of customers, subject to various constraints such as vehicle capacity, route length, time Windows, priority relationships among customers, and so on. Each day is a different VRP issue, but it shows that today's issues are very similar to yesterday's issues, such as road layout, number of vehicles, and some delivery points.

2.3 Related Work

The algorithm to solve travelling salesman problem can be divided into exact algorithm and heuristic algorithm and metaheuristic algorithm.

2.3.1 Exact algorithm

The exhaustive method is the exact algorithm that lists all the possible solution and compares them to find the best solution as the global optima. In the TSP problem, as the number of city states increases a little bit, computational complexity will grow exponentially. For example, to solve a TSP problem with 20 cities, there are $(n-1)!/2 = 1.2 * 10^{18}$. Even if it only takes 0.000000001 second to get a valid solution, it will take 350 years to get all the solutions.

There are some other exact algorithms like branch-and-bound algorithm and branch-and-cut algorithm in Laporte and Nobert (1983). Compared to the original exact algorithm, they do make a huge improvement in computing complexity, but still it's not possible for TSP problem with more cities to find the global optima in reasonable time.

2.3.2 Heuristic algorithm

Unlike the exact algorithm, the heuristic algorithm does not force to get the optimal solution, instead, they try to get the relative better solution. There are two typical algorithms in classic heuristics: 1) Path construction algorithm, which is based on certain rules to add cities one by one in the path, after all the cities have been added, a circuit path will be obtained. Different algorithms have different rules on adding cities to the path. 2) Path optimization algorithm, which generates an initial path first and then changes the sequence order of the original path to optimize and by running more times, probably a local optima solution will be obtained.

2.3.3 Metaheuristic algorithm

Significant progress has been witnessed in the development of intelligent method. Metaheuristic algorithm can be grouped into 1) local search 2) population search and 3) learning mechanisms.

Local search, also known as neighbourhood search or hill climbing, is the basis of many heuristic approaches for combinatorial optimization problems. To find a good approximate solution, the method is repeated many times until it can't find any better solution. The main idea behind it is to make a trial, fail and try again. Voudouris and Tsang (1999) presented how the techniques of combining Guided Local Search (GLS) and Fast Local Search (FLS) together to solve the TSP problem. The combination of Global Local Search and Fast Local Search is studied with TSP local search heuristics of different efficiency and effectiveness.

Genetic algorithm is a tyical population search method based on evolutionary thinking, natural selection and inheritance. which evolves a population of solutions encoded

as bitstrings. The basic concept of Genetic algorithms aims to simulate the processes necessary for evolution in natural systems, especially those processes that follow the survival of the fittest principle first proposed by Charles Darwin. Genetic algorithms have been successfully applied to many problems of business, engineering, and science. Pelikan and Goldberg (2010) showed that genetic algorithms were now playing an increasingly important role in computational optimization and operations research since the operational simplicity and wide applicability of the evolutionary algorithm.

Learning mechanism includes DL-based method, simulated annealing and ant colony optimization. Bookstaber (1997) first explained the details of an implementation of a Simulated Annealing (SA) algorithm to find optimal solutions to the TSP problem. Yang et al. (2008) succeeded in extending the ant colony optimization method from TSP to a generalized TSP problem which was a very simple but practical extension of TSP. Moreover, the latest study of Kool et al. (2019) proposed a model based on attention layers with benefits over the Pointer Network and the mode was trained with reinforcement learning with a simple baseline based on a deterministic greedy roll out. The paper suggested that the model has significantly improved over recent learned heuristics for the Travelling Salesman Problem (TSP), getting close to optimal results for problems up to 100 nodes.

The common feature of this type of method is that some feasible solutions are randomly generated first, and then new feasible solutions are generated from these initial feasible solutions through certain substitution rules. This is repeated until the predetermined stopping criterion is met, and the final feasible solution is obtained. The solution is the approximate solution of the optimal solution of the problem.

Chapter 3

Methodology

In this section, three traditional intelligent algorithms are introduced and presented to solve the traveling salesman problem problem : 2-opt Hill Climbing, Genetic Algorithm and Hopfield Network. Meanwhile, the Hybrid Method that combines Hopfield Network and 2-opt Hill Climbing is introduced, the potential ability can be seen from the framework in solving real-life problems.

3.1 2-Opt Hill Climbing

2-opt Hill Climbing algorithm is a random-restart hill climbing with 2-opt local search applied to Travelling Salesman Problem was first introduced by Croes in 1958 Croes (1958). The main idea behind 2-opt local search is the swapping mechanism that crossovers itself and reorders the route so that the route can meet the requirements. A complete 2-opt Hill Climbing algorithm will select the optimal solution from all of the possible routes.

The advantages of the 2-opt search method is that it can be implemented easily and fast. 2- opt Hill Climbing is mainly used when there are good heuristics. Just as other local search algorithms, there is no need to maintain and process the search tree or graph, because it only retains a single current state. However, 2-opt hill climbing will be easily trapped in the local optimums and it does not have any mechanism to jump out of them. Therefore, it is very sensitive to the initial configurations.

Algorithm 1 2-opt Hill Climbing

0: *minimum_Distance* ← *initial_Distance_of_Sequence*
  for all the number of nodes eligible to be swapped **do**
    i ← *start_index_of_reverse_order*
    k ← *end_index_of_reverse_order*
    add Node[0] to Node[i] to new route
    add Node[i] to Node[k] in reverse order to new route
    add Node[]k+1] to end in order to new route
    new_distance ← *Distance_of_new_Route*
    if *new_distance* < *minimum_Distance* **then**
      update *minimum_Distance*
  end for=0

Figure: 3.1 The following figure presents an example of how 2-opt algorithm solving the TSP problem. The initial order of the route is a →b →e →d →c →f →g if $i = 1$ $k = 4$. Node[a] and Node[b] were added to the new route, the new route was set up as: a →b. Then the route from Node[e] to Node[c] was reversed and added to the new route: a →b →c →d →e. Finally, the rest of the route was added to the new route a →b →c →d →e →f →g.

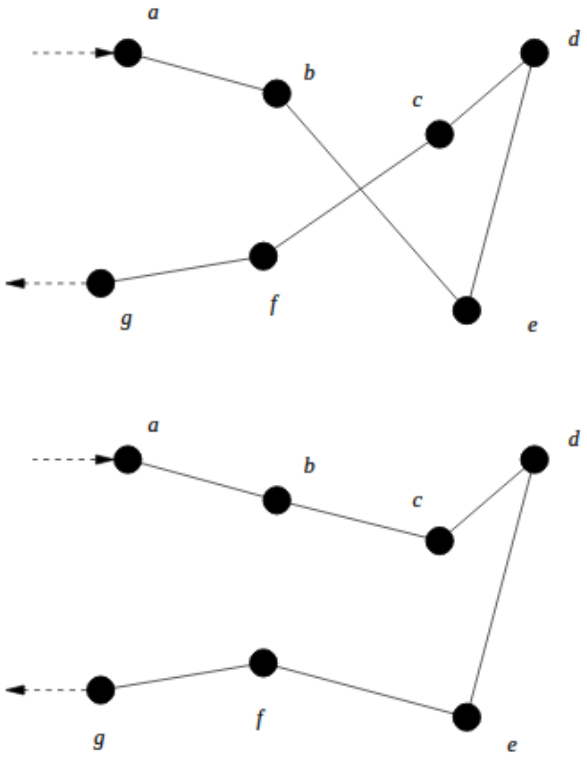


FIGURE 3.1: 2-opt hill Climbing Algorithm

3.2 Hopfield Network

Hopfield Network was first introduced by Hopfield, J. J. in 1985 Tank and Hopfield (1985). The Hopfield network is a recurrent neural network with highly- interconnected neurons, which means that each neuron is connected to other neurons. The main idea of the Hopfield network is to map the problem to the inter-connected neurons, evolve the equilibrium of the state of the network through the dynamic update of the neurons and search for the local optimal solution of the problem. However, there is no guarantee to find the global optimal solution.

By updating the weights and inputs of neurons recursively to minimize the energy function, Hopfield Network can solve the specific optimization problem and achieve a desired state. The rules for updating neurons are based on the differential equation of the energy function, which ensures that Hopfield Network can get the local optimal.

There are two types of Hopfield Network: The Discrete Hopfield Network (DHNN) and the Continuous Hopfield Network (CHNN). The difference is the value of neuron. In DHNN model, the values of the neuron can only be binary values like {0, 1} or {-1, 1}. The value of the neuron is continuous and non-linear.

Figure: 3.2 of Hopfield Network shows the structure of the hopfield network : The neurons are connected in bi-direction which gives a symmetric weight metrix: $W_{ij} = W_{ji}$; The neurons are not self-connected which means $W_{ii} = 0$;

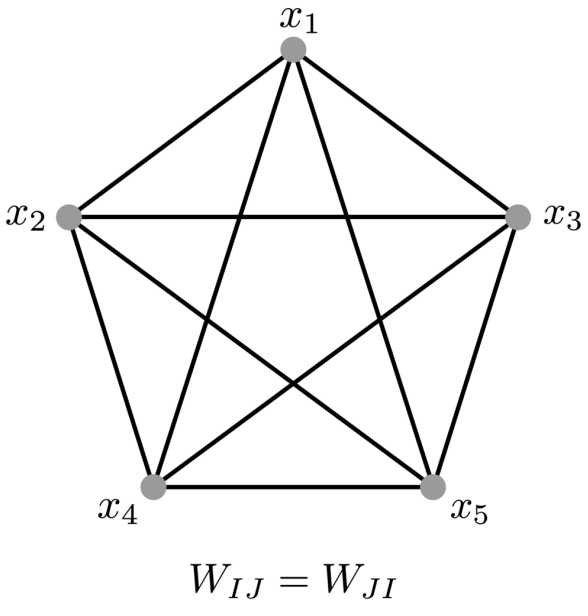


FIGURE 3.2: The connectivity diagram of the fully-connected Hopfield Network consisting of five neurons.

3.2.1 Continuous Hopfield Network

In 1984, Hopfield implemented Hopfield network by using analog electronic circuit. The activation function of the neurons in this network is a continuous function, so this network is also called the continuous Hopfield network. In the continuous Hopfield network, the input and output of the network are analog quantities, and each neuron works in synchronous mode. Hopfield applied the network to solve the optimization problem, and successfully solved the TSP problem.

Figure: 3.5 shows the structure of continuous Hopfield Network : Each neuron in a Hopfield neural network is composed of an operational amplifier and its associated circuits.

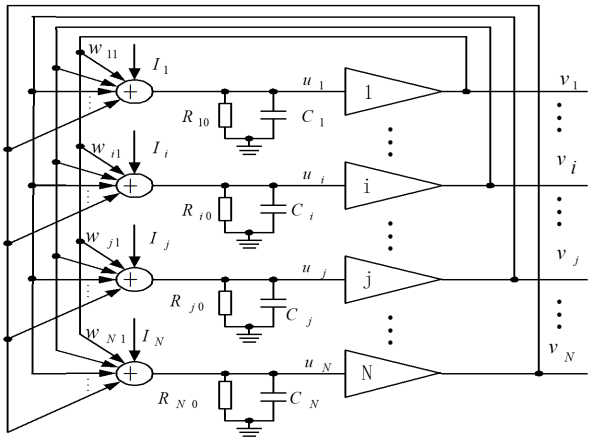


FIGURE 3.3: The structure of continuous Hopfield Network

Each operational amplifier I (or neuron I) has two sets of inputs: The first is a constant external input, denoted by I_i, which corresponds to the current input of the amplifier; The second set is the feedback connection from other operational amplifiers, such as: another arbitrary operational amplifier J (or neuron J), represented by W_{ij}, which corresponds to the connection weight between neuron I and neuron J. u_i represents the input voltage of an operational amplifier while V_i represents the output voltage of an operational amplifier. The activation function is as follow:

$$V_i = f(u_i) = \tanh\frac{a_i u}{2}) = \frac{1-\exp -a_i u_i}{1+\exp -a_i u_i} \tag{3.1}$$

The inverse of the activation function can be derived as follow:

$$u_i = f^{-1}(v_i) = -\frac{1}{a_i}\log(\frac{1-v_i}{1+v_i}) \tag{3.2}$$

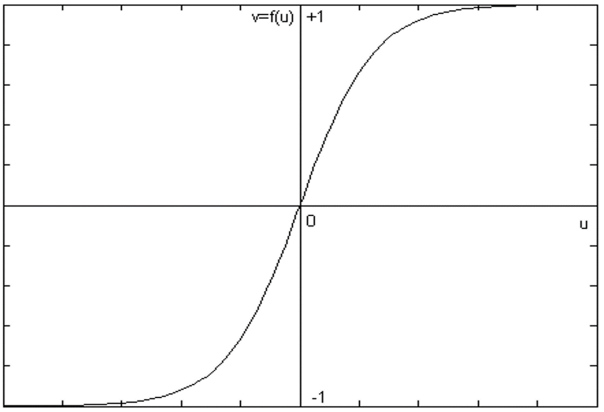


FIGURE 3.4: The mapping of **Equation 3.1**

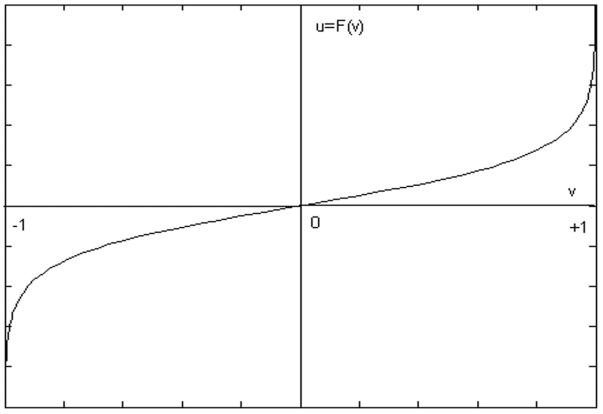


FIGURE 3.5: The mapping of **Equation3.2**

According to the Kirchhoff Circuit Law:

$$C_i \frac{du_i}{dt} + \frac{u_i}{R_{i0}} = \sum_{j=1}^{N} \frac{1}{R_{ij}} (V_j - u_i) + I_i \tag{3.3}$$

$$C_i \frac{du_i}{dt} = \sum_{j=1}^{N} w_{ij} (V_j - u_i) + I_i - \frac{u_i}{R_{i0}} \tag{3.4}$$

$$C_i \frac{du_i}{dt} = \sum_{j=1}^{N} w_{ij} V_j + I_i - \frac{u_i}{R_i} \tag{3.5}$$

The energy function can be defined as follow:

$$E = -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} V_i V_j + \sum_{i=1}^{N} \frac{1}{R_i} \int_0^{u_i} f^{-1}(V_i)\, dV_i - \sum_{i=1}^{N} I_i V_i \tag{3.6}$$

The continuous Hopfield network model highlights the main characteristics of neural computing in biological systems, such as:

- The neurons in the continuous Hopfield network have Sigmoid transmission characteristics for the I/O transformation.
- Hopfield Network has the function of space-time integration.
- There are many connections between excitement and inhibition between neurons, mainly through feedback.
- Hopfield Network has neurons that represent action potentials, as well as neurons that work in a progressive manner. Therefore, the network retains the two most important computational characteristics: dynamics and nonlinearity.

3.2.2 Associative Memory

Hopfield Network is a neurodynamic system with a stable equilibrium state, that is, there is an attractor which is a close subset of the state, so that Hopfield Network has the associative memory. Associative memory can also return a storage pattern similar to the current pattern, so that noise input can also be recognized.

The process of generating associative memory can be divided into two stages: one is the memory stage, also known as the storage stage or the learning stage; The second is the association stage, also known as the recovery stage or recall stage. The memory phase is to design or learn the weights of the network so that the network has several stable equilibrium states called attractors. The associative phase is the process in which the associative memory network reaches a stable state through the dynamic evolution, that is, converges to the attractor and recalls the stored mode.

Hopfield Network with an associative memory works as follow:

- Set the memory mode and encode the memory pattern to the values of {-1, 1}:

$$U_k = [u_1^k, u_2^k, \cdots u_i^k, \cdots, u_n^k]$$

- Set the weights by simply implementing Hebbian Learning:

$$f(x) = \begin{cases} \frac{1}{N} \sum_{\mu=1}^{M} u_i^k u_i^k & (j \neq i) \\ 0 & (j = i) \end{cases} \tag{3.7}$$

- Initialization: set the memory mode as the initial state of the Hopfield Network :

$$V_i(t) = u_i \ (t = 0) \tag{3.8}$$

- Update the state of the neurons:

$$V_i(t+1) = \mathrm{Sgn}[\sum_{j=1}^{N} w_{ij} X_j(n)] \tag{3.9}$$

- Converge: Iterate repeatedly until the state of all neurons in the network does not change

- Output the steady state of the Hopfield Network:

$$y = V_i(T) \tag{3.10}$$

There are two constraints to implement a Hopfield Network with associative memory successfully: The network needs to have a large memory capacity; The network needs to have a certain fault tolerance.

The memory capacity of Hopfield associative memory network is the maximum number of non-interference sequences stored in the network under a certain tolerance of associative error probability. The memory capacity α reflects the relationship between the remembered pattern M and the number of neurons N: $\alpha = m/N$

The number of neurons required to remember m patterns is $N = m/\alpha$, and the number of connection weights is $(m/\alpha)2$. If α is doubled, the number of connection weights is reduced to 1/4 of the original. Experimental and theoretical studies show that the upper limit of memory capacity of Hopfield associative memory network is $0.15n$ Abu-Mostafa and St. Jacques (1985)

3.3 Genetic Algorithm

Genetic Algorithm (GA) is a computational model that simulates the biological evolution process of natural selection and genetic mechanism of Darwin's biological evolution theory. Also it is a method of searching for the optimal solution by simulating the natural evolution process.

The Genetic Algorithm takes all individuals in a group as the population, and uses randomization technology to guide an efficient search for a better population. A Genetic Algorithm contains five phrases: Initialization, Fitness Function, Selection Strategy, Crossover and Mutation:

- Initialization: The genetic algorithm starts with an initial population of the possible solutions which can be referred to as chromosomes. In the TSP problem, the population is a set of Route Sequences. The chromosomes represent the possible route sequence while the genes represent each city-state.

- Fitness Function: A fitness function is set up to measure the quality of the population which can be used to distinguish the good individuals from the bad ones. Once an offspring population is created, each of the individuals will be evaluated by the fitness function.
- Selection Strategy: Selection Strategy is to select the better individuals with a better fitness value out of the worse ones. Individuals with a higher fitness score are more likely to be selected to breed. Many of the selection methods has been proposed: routette-wheel selection, stochastic universal selection, elite selection, tournament selection, scaling window selection and so on.
- Crossover: Crossover is to recombine the genes of two or more randomly selected parents. The generated new offspring combines the features of both parents which is probably a a better solution. Generally speaking, after each round of reproduction, the fitness score of the population will naturally increase, although this is not guaranteed. Many crossover method has been designed such as one-point crossover, two-point crossover and uniform crossover.
- Mutation: The motivation of the mutation is that if lucky, the offspring of the mutation may have better fitness value than their parents (but in most cases, the mutation is bad) making it more likely to be selected for crossover in the next stage, and it good genes are deeply ingrained in the population. Some of the offspring will be affected (low probability) random mutations at each intersection, which means that some of their features randomly mutate and is not inherited from the parents. Thus randomness is introduced into this algorithm and with the method of mutation, the algorithm may not be stuck on the local minimum.

Algorithm 2 Genetic Algorithm

```
0: solution = 0
0: Population_of_solution = randomly initialized population
   while termination criterion = false do
     solution = solution + 1
     for each round do
       select P(solution) from P(solution − 1) based on fitness value
       apply genetic operators (crossover and mutation) to P(solution)
       evaluate each individual in P(solution)
     end for
   end while=0
```

The main advantages of the Genetic Algorithm are as follow: Genetic Algorithm is not restricted by the derivation and continuity of a function; Genetic algorithm has the inherent ability to evolve better solutions and obtain relatively better global optimization solution; Genetic Algorithm can automatically obtain and guide the optimized search space without definite rules, and adjust the search direction.

3.4 Hybrid Method

The idea of the improved methodology which combines the Hopfield Network with 2-opt hill climbing together is from previous research Watson et al. (2011). The methodology in the previous research is a complex adaptive system combines deep learning and evolutionary algorithm together to solve solving the random and modular network constraint problems which is equivalent to graph coloring and distributed task allocation problems. The evolutionary algorithm is applied to find the energy minimization which is the local optimizations of the constraints while the Hopfield Network is used to develop an associative memory that amplifies a subset of its own attractor states so that it has a potential to solve repeated problems. The previous research has also suggested that the Hybrid method could find the globally optimal resolutions of constraints more reliably and more quickly in random and modular network constraint problems, though Hebbian learning did not provide a strong optimization.

Because genetic algorithm, ant colony algorithm and simulated annealing algorithm have strong group search ability, but also have the possibility of falling into local optimal, researchers usually combine other search algorithms with these algorithms to construct more efficient hybrid algorithms. Wang and Guo (2010) suggested that A hybrid algorithm (HA) integrated genetic algorithm (GA) with ant colony optimization (ACO) for solving Traveling Salesman Problems(TSP) was studied to get better optimization performance than each single algorithm. Gülcü et al. (2018) also presented a parallel cooperative hybrid algorithm which based on ant colony optimization and 3-Opt algorithm for solving traveling salesman problem. Although heuristic approaches and the hybrid methods could also obtained good results in solving the TSP, though they cannot successfully avoid getting stuck to local optima.

Since Neural network and self-organizing graph have the ability of self-learning, associative storage and telling to find the optimal solution. Using them Hybrid with other methods also provides the possibility for algorithm optimization of TSP problem. Instead of combining the Hopfield Network with evolutionary algorithm together which was introduced in the previous research, I made an attempt to combine Hopfield Network and 2-opt Hill Climbing algorithm together since compared with the traditional model of genetic algorithm, 2-opt Hill Climbing has advantages in computing resources and efficiency.

Chapter 4

Implementation and Analysis

In this section, the Tank and Hopfield (1985) was re-implemented to solve the TSP problem. The details are show in **Section 4.2** Based on that, a self-modeling framework was carried out in **Section 4.3** and in **Section 4.3**. Hybrid Method was compared with Hopfield network, 2-opt hill climbing algorithms and genetic algorithm.

4.1 Problem Description

TSP problem is to find a closed path that passes through each city only once, returning to the start city and the path length must be the shortest. The Traveling Salesman Problem (TSP) is one of the most famous combinatorial optimization problems. This problem is very easy to explain, but very complicated to solve – even for instances with a small number of cities.

Parameters: Assume a set of N cities was given and the distance between them was:

d_{ij} : distance between city i and city j (Notice that $d_{ij} = d_{ij}$)

$x_{ij} = \{0, 1\}$: 0 means the city i and city j is not connected ;1 means they are connected

Objective Function: To minimize the total distance of route :

$$min \sum_{i=1}^{N} d_{ij} x_{ij} \tag{4.1}$$

Constraints: The valid X matrix of five cites:

$$X = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

- The city can be visited only once: For each row of matrix X, there is only one value equals to one.
- There is only one city waiting to be visited: For each columns of matrix X, there is only one value equals to one.
- All the city need to be visited : $\sum x_{ij} = N$

For the TSP problem of N cities, there are $2n$ routes which are equal in length because the length of each route has nothing to do with the start city and the direction of the route. Also, there are $\frac{n!}{2n}$ travel routes with different length for the TSP problem of N cities.

4.2 Hopfield Network

In this section, I re-implemented the Hopfield Network based on the paper published by Tank, D. W. and Hopfield, J. J. Tank and Hopfield (1985) and tested it on a TSP problem with 10 cities.

4.2.1 Energy Function

Solving the TSP problem with the Hopfield network requires choosing an adequate energy function, in the original paper E was defined as:

$$E = E1 + E2 \tag{4.2}$$

Where

$$E1 = \frac{A}{2}\sum_{x=1}^{N}\sum_{i=1}^{N}\sum_{j\neq i}^{N} V_{xi}V_{xj} + \frac{B}{2}\sum_{i=1}^{N}\sum_{x=1}^{N}\sum_{y\neq x}^{N} V_{xi}V_{yi} + \frac{C}{2}(\sum_{x=1}^{N}\sum_{i=1}^{N} V_{xi} - N) \tag{4.3}$$

$$E2 = \frac{D}{2}\sum_{x=1}^{N}\sum_{y\neq x}^{N}\sum_{i=1}^{N} d_{xy}V_{xi}(V_{yi+1} + V_{yi-1}) \tag{4.4}$$

The energy function of a TSP problem should meets the following requirements:

- The energy function should be suitable for the stable state as an output in the form of permutation matrix
- In all of the valid routes, the energy function is beneficial to the shortest route.

E1 **Equation: 4.3** is the penalty function which keeps the output of the network valid.

- $\frac{A}{2}\sum_{x=1}^{N}\sum_{i=1}^{N}\sum_{j\neq i}^{N} V_{xi}V_{xj}$ works as the first penalty to keep all the rows of the output matrix should contain only one value of **1**.
- $\frac{B}{2}\sum_{i=1}^{N}\sum_{x=1}^{N}\sum_{y\neq x}^{N} V_{xi}V_{yi}$ works as the second penalty to keep all the columns of the output matrix should contain only one value of **1**.
- $\frac{C}{2}(\sum_{x=1}^{N}\sum_{i=1}^{N} V_{xi} - N)$ works as the third penalty to keep the output matrix contain N of **1**s.

E2 **Equation: 4.4** is the function to calculate the sum of route distance which is beneficial to the shortest route.**Figure: 4.1**shows the landscape of a Hopfield Network including the basin of attraction in green, the current state, the converged attractors. The energy of the Hopfield network will always decrease once the algorithm began.

A	500
B	500
C	200
D	500

TABLE 4.1: Parameters of Energy Function

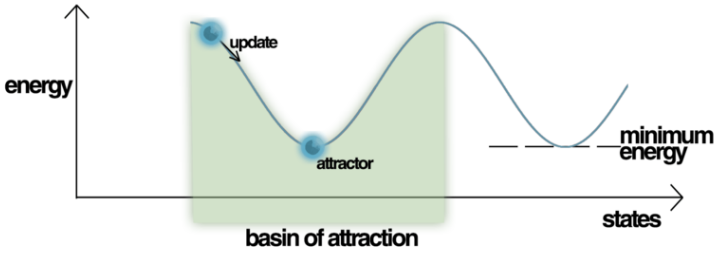


FIGURE 4.1: The energy landscape of the Hopfield network

4.2.2 Map of Cities

In the original paper, the location of the cities are randomly initialized with a uniform probability density. For better simulation and comparison, the X-coordinates and Y-coordinates were as close as possible to the value of the original cities. **Table: 4.2** shows the parameters of the cities in **Figure: 4.2**

City	X-coordinate	Y-coordinate
A	0.25	0.16
B	0.85	0.35
C	0.65	0.24
D	0.7	0.5
E	0.15	0.22
F	0.25	0.78
G	0.4	0.45
H	0.9	0.65
I	0.55	0.9
J	0.6	0.28

TABLE 4.2: The coordinates of Cities

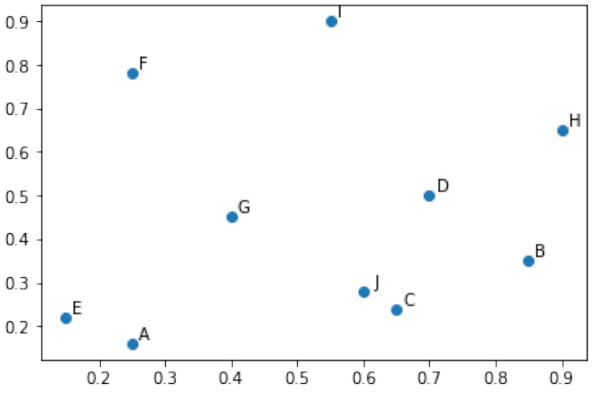
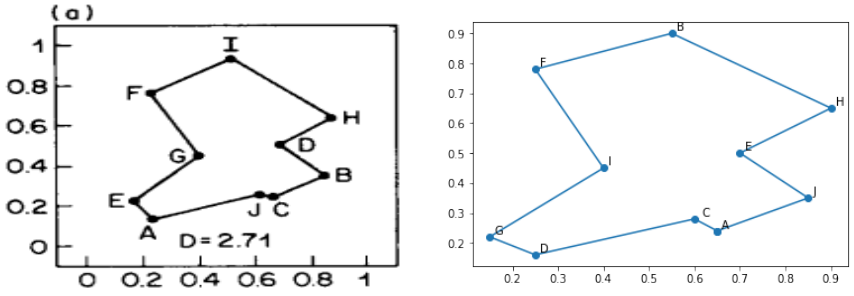


FIGURE 4.2: Map of the cities

Figure 4.3 shows a compatison of map in original paper and my simulation. The total distance in the orignal paper is 2.71cm and in my re-implementation is 2.69cm, also it is the shortest path of ten cities.



(a) The original map (b) My simulated map

FIGURE 4.3: Comparison of map in original paper and my re-implementation

4.2.3 Experiment Results

The analog network for the TSP problem generated the following equations of motion:

$$\frac{du_{xi}}{dt} = -\frac{u_{xi}}{\tau} - A\sum_{j\neq i} V_{xj} - B\sum_{Y\neq X} V_{Yi} - C\Big(\sum_X \sum_j V_{xj} - n\Big) \quad - D\sum_Y d_{XY}(V_{Y,i+1} + V_{Y,i-1}) \tag{4.5}$$

$$V_{xi} = g(u_{xi}) = \frac{1}{2}(1 + tanh(\frac{u_{xi}}{u0}) \tag{4.6}$$

The deriviation of the above equation can be found in the earlier section **Equation: 3.3, 3.4 and 3.5** Note that in the implementation, τ was set as 1 which meant there was no loss of generality. The initial value of symmetry breaking in ordering contraints was set as $-0.1u_0 \leq u_{xi} \leq 0.1u_0 \quad (u_0 = 0.02)$

Figure : 4.4 and 4.5 is the comparison of distribution of all the possible path length and the distribution of Hopfield Network. **Figure : 4.4** shows a histogram of the number of different paths length for the TSP with 10 cities. There are $\frac{10!}{20} = 181400$ total distinct paths for a TSP problem with 10 cities and the average of all the possible path length is 5 while most of the path length of routes generated by the Hopfield network in **Figure: 4.5** was much more better than the average path length also it is much more easier to converge to the shortest routes.

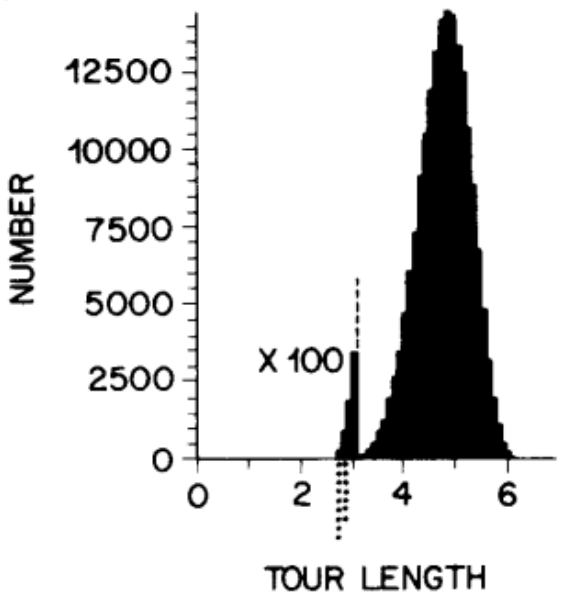


FIGURE 4.4: Distribution of all the possible path length of 10 cities

Figure: 4.6 compares the path route generated by random selection and Hopfield Network. **Figure: 4.6(a), 4.6(b)** is the average path found by the Hopfield Network

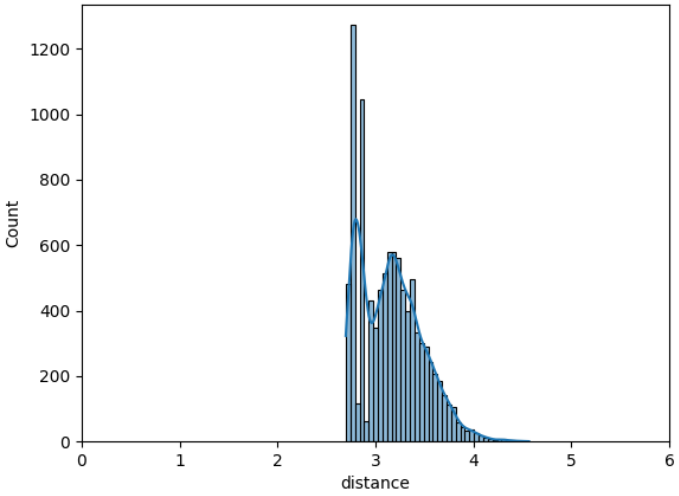
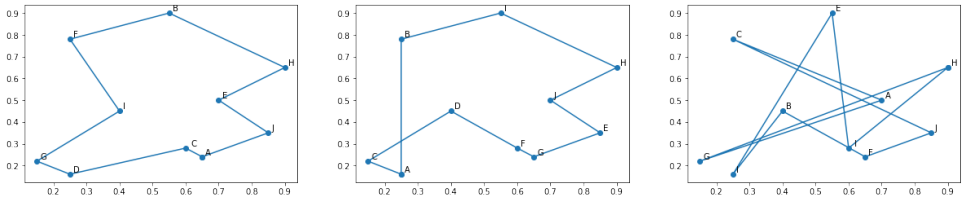


FIGURE 4.5: Distribution of Hopfield Network results of 100000 trials

with a seperate route length of 2.69 and 2.84 while **Figure 4.6(c)** is the average path found by a random selection with a distance of 5.52.



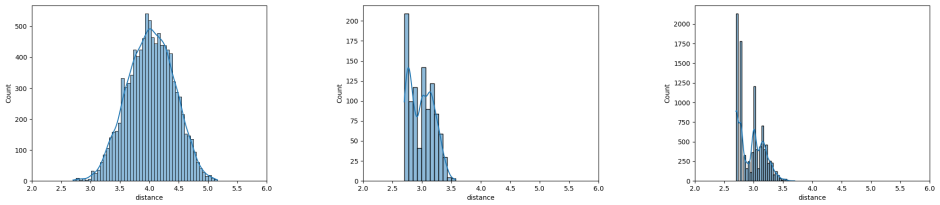
(a) Optima route (b) Converged route (c) Random selected route

FIGURE 4.6: Comparison of route path

4.3 2-opt Hill Climbing

In this section, the 2-opt Hill-Climbing algorithm has been implemented and tested with different iterations to solve the same TSP problem. To get more reliable results, I run 2-opt 100 iterations in with different random starting points and the best distribution result is in **Figure 4.7(c)**.

In terms of optimization, 2-opt Hill Climbing algorithm is a simple local search algorithm with a special exchange mechanism, which is very suitable for solving the traveling salesman problem. From **Figure: 4.7**, we can see that the 2-opt hill climbing gets converges very quickly. However, the algorithm is sensitive to the initial point of search, that is, the final result will change with different initial points. Also, just the same as the classic Hill Climbing methods, it can be stuck in the local optima solution quite easily.



(a) two opt with only one iteration (b) two opt with five iterations (c) two opt with one hundred iterations

FIGURE 4.7: Distribution of two opt algorithm with different iterations

4.4 Genetic Algorithm

In this section, the traditional Genetic Algorithm was implemented , the parameters of traditional GA model are in the following **Table: 4.3** and also there is a change on the crossover operator to fix the TSP problem.

4.4.1 Design of Genetic Algorithm

The main phrases of Genetic Algorithm is Initialization, Evaluation, Selection, Crossover and Mutation, the implementation details are as follow:

- Initialization: The initialization of the TSP problem including setting the *size*, the crossover rate P_c, the mutation rate P_m and setting the iteration round to zero.
- Evaluation: The fitness value in our implementation is set as the opposite of route length of each of the route individual in the population.
- Selection: The selection is based on the scaling window technique of W (width of the window) . The scaling window is set as follow: f_{min} = the least value of the $f(x)$ in the last W generation. For example: if $W = 0$. $f(x)$ is set to the minimum fitness value of last generation, and in the next round of selection, any individual with a fitness value less than $f(x)$ won't be selected in the next generation.
- Crossover: The crossover rate P_c is set as the frequency of when the crossover operator will be applied. The higher the crossover rate is, the structure of the generation changes more quickly. If the crossover rate is set to be too high, some good and stable structure will be discarded as well.
  The Crossover operator has been changed a little bit to solve the TSP problem: Traditional Way of a One-point Crossover: For example: two individuals are randomly selected for the new individual: Sequence of Parent A : $4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 5$; Sequence of Parent B : $3 \rightarrow 2 \rightarrow 5 \rightarrow 1 \rightarrow 4$. By doing crossover at the second position of city, $4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 4$ is what we got. The first 3 states from Parent A and the last 2 states from Parent B. However, it is not valid since

we need to visit all the cities and visit each of them only once.
Improved Crossover method: We keep the same part of parent A but we sort the rest cities in the order of parent B, we will get $4 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 1$ as the result.

- Mutation: The mutation rate P_m is the operator which is for increasing the variability of the population. If the mutation rate is too high, the performance of the genetic algorithm will be similar to random search. In our implementation, the mutation is set as the random exchange of the position of the city.

P_c	0.6
P_m	0.1
size	10

TABLE 4.3: Parameters of Genetic Algorithm

4.4.2 Experiment Result

Figure 4.8 shows the distribution of 10000 trials of the GA model, the performance is quite similar with the random search. To balance the time complexity and space complexity, I choose a relatively small population size. Compared with the performance of 2-opt hill climbing in **Figure 4.7**, it can be seen that traditional Genetic Algorithm is not good enough in solving TSP problem which gives us a strong reason to build a self-modeling framework with 2-opt Hill Climbing method instead.

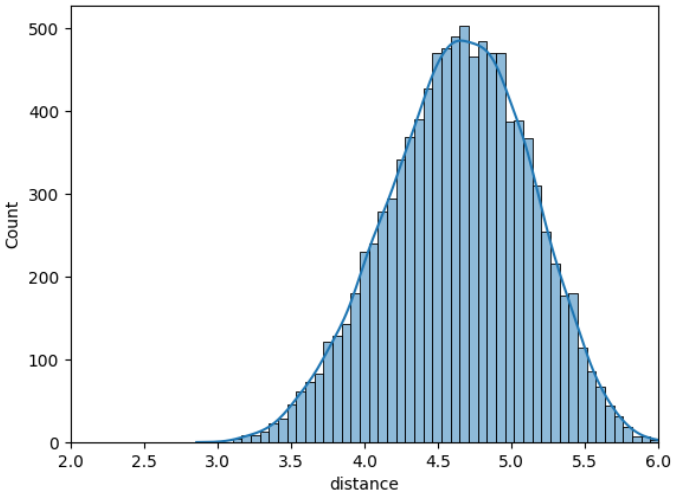


FIGURE 4.8: Distribution of Genetic Algorithm

4.5 Hybrid Method

In this section, Hybrid method based Hopfield Network and 2-opt was implemented. The parameters of Hopfield Network are the same as the re-implementation of original paper Tank and Hopfield (1985).

4.5.1 Design of Hybrid Method

The design of the system **Figure: 4.9** can be divided into two parts: Training the Hopfield Network to get an initial solution and Applying 2-opt hill-climbing to solve the problem. The initial solution is the optimal solution got from yesterday's problem which probably has a similar pattern with today's problem so that we can see potential in solving repeated problems.

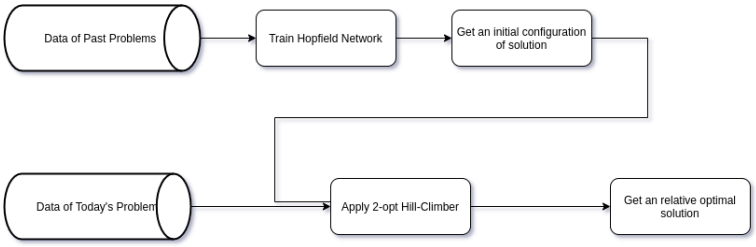


FIGURE 4.9: Design of Hybrid Method

4.5.2 Experiment Result

Figure 4.10 shows the distribution of 10000 trials of Hybrid Methods on TSP problem. One out of four of the results can achieve the optimal solution.

4.6 Comparison

4.6.1 Comparison of Time Complexity

The time complexity is the computational complexity that describes the amount of computer time it takes to run the algorithm. n stands for the number of cities. T stands for the outer rounds of iteration.

- Time Complexity of Genetic Algorithm is $O(Tpn^2)$ (p stands for the number of population) : Initialization phrase: $O(pn)$, Evaluation phrase: $O(pn)$, Selection phrase: $O(pn)$, crossover and mutation operator: $O(pn^2)$.
- Time Complexity of 2-opt Hill Climbing: $O(Tn^2)$

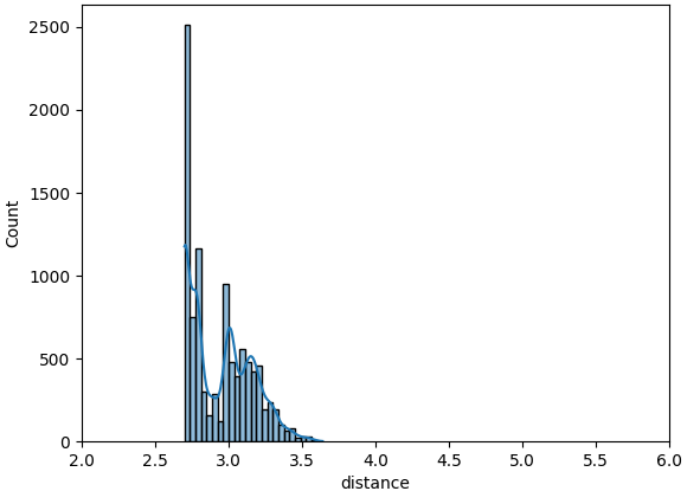


FIGURE 4.10: Distribution of Hybrid Method

- Time Complexity of Hopfield Network:$O(TT_1n^2)$ (T_1 stands for the number of inner loop iteration)
- Time Complexity of Hybrid Method:$O(TT_1n^2) + O(Tn^2)$

Normally, the time complexity of Genetic Algorithm, 2-opt Hill Climbing, Hopfield Network and Hybrid Method is $O(Tpn^2)$>$O(TT_1n^2) + O(Tn^2)$>$O(TT_1n^2)$>$0(Tn^2)$. Genetic algorithm is the slowest because, in order to get good results, it is always necessary to set a relatively large population.

To ensure the fairness of the experiment in terms of time, every iteration is run in 2 seconds for all the algorithm above and the comparison of performance is in **Figure 4.15**

4.6.2 Comparison of Space Complexity

The space complexity of an algorithm or a computer program is the amount of memory space required to solve an instance of the computational problem as a function of characteristics of the input. All of the four algorithm need space to save the distance matrix which is $O(n^2)$

- Space Complexity of Genetic Algorithm is $O(pn) + O(n^2)$ (p stands for the population)
- Space Complexity of 2-opt Hill Climbing: $O(n^2)$

- Space Complexity of Hopfield Network: $O(2n^2) + O(n^2)$ since the v matrix and u matrix need to be stored.
- Space Complexity of Hybrid Method: $O(3n^2)$

4.6.3 Comparison of Result

Figure 4.15 suggests that compared to the rest of the methods, the Hybrid Method is more able to guarantee the optima solutions for the TSP problem. The distribution is the result of 10,000 runs from all algorithms. The Distribution of Hybrid method in **Figure 4.9** is with an average of **2.93**, which is the best overall. Almost one out of four solutions can reach a global optima. Genetic Algorithm in **Figure 4.12** is with an average of **4.63**. The performance of Genetic Algorithm is just a little bit better than the normal distribution of all the possible routes. The Distribution of 2-opt Hill Climbing Algorithm in **Figure 4.13** is with an average of **2.96**. 2-opt Hill Climbing is a simple and efficient local search algorithm for the TSP problem. More than one out of five solutions are the global optima. However, from the distribution, it can be seen that the algorithm is quite easier to stuck on the local optima, nearly one out of five solutions are quite near the best solution. The Distribution of Hopfield Network in **Figure 4.14** is with an average of **3.85**, which is worse than the result from 2-opt Hill Climbing but still works.

Algorithm	Path Length
Hybrid Method	2.93
2-OPT	2.96
Hopfield Network	3.85
Genetic Algorithm	4.63

Table 4.4: Average Path Length of the algorithms

The comparison of performance between the 2-opt Hill Climbing method and classic genetic algorithm also gives me a strong reason to combine the 2-opt with Hopfield Network in the Hybrid Method instead of the Genetic Algorithm.

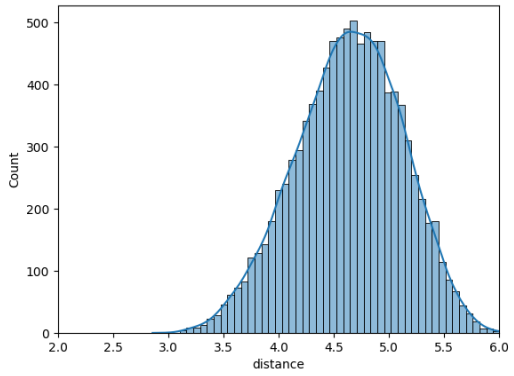


FIGURE 4.12: Distribution of Genetic Algorithm

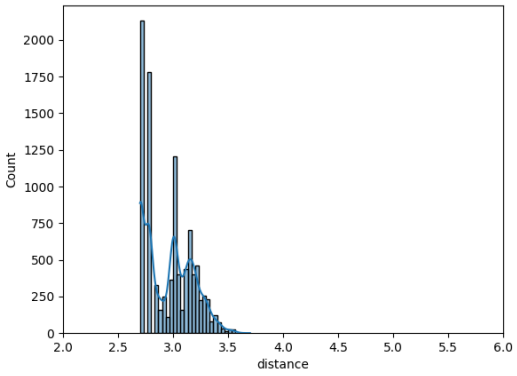


FIGURE 4.13: Distribution of 2-opt Hill Climbing Algorithm

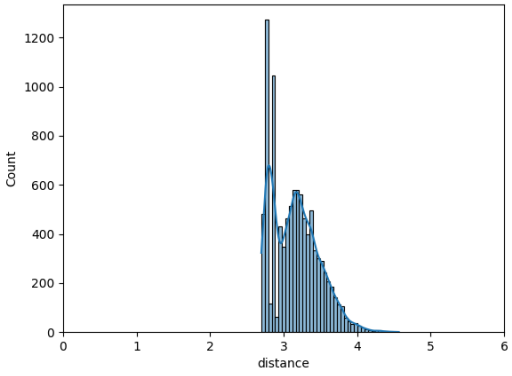


FIGURE 4.14: Distribution of Hopfield Network

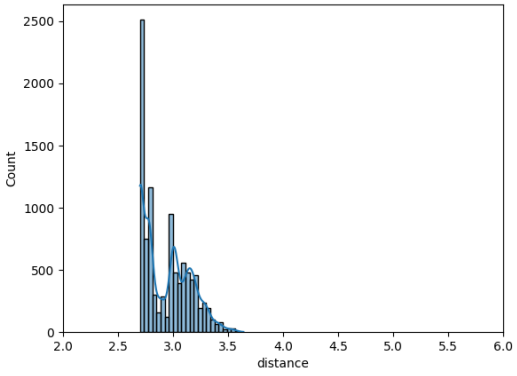


FIGURE 4.15: Distribution of Hybrid method

FIGURE 4.15: Comparison of Methods

Chapter 5

Discussion

In this section, the limitations and advantages of Hybrid Method will be discussed based on experimental results and recent research.

5.1 Advantages and Limitations

The Hybrid Method proposes that combines local search and intelligent algorithms to solve the TSP problem which makes certain optimization on either single method. The most obvious limitation of 2-opt hill-climbing algorithms is that they are local search algorithms. Therefore, they usually only find local optima. Therefore, if any of these algorithms converges to a local optima, and there is a better solution that is close to the found solution in both the solution and the search space, then none of these algorithms will find that solution. Better solutions. They would basically be trapped. Therefore, in order to overcome their limitations, the combination of Hopfield Network and 2-opt Hill-climbing algorithm is applied to the TSP problem, which provides the possibility to find global optima.

In the comparison of the experiment results, more cutting-edge methods should be added. Only being compared with the traditional GA model, 2-opt Hill Climbing and Hopfield Network can't prove the new Hybrid Method can work as the advanced method in solving TSP. In addition, The new Hybrid Method can be tested on TSP problem with more cities to see how it performs in solving large-scale TSP problem.

5.2 Future Work

The future exploration of TSP problem and Hybrid Method can be divided into the following three directions: Optimization on existing algorithm, Optimization on Hybrid algorithms and trials on more vaiations of TSP.

5.2.1 Optimization on existing algorithm

The optimization on existing algorithms to seek breakthroughs in method theory, so as to improve the existing algorithms and get a better solution to the TSP problem. For example, the main disadvantage of the continuous Hopfield network when used to the traveling salesman problem, is the infeasibility of the obtained solution and the process of setting the values of the model parameters by trial and error. Tan et al. (2005) presented some stability criteria that ensure the convergence of valid solutions and suppression of infeasible solutions on the parameter settings of Hopfield networks applied to traveling salesman problems. There are also some ways to optimize 2-opt Hill Climbing, Rocki and Suda (2012) provided the acceleration of 2-opt and 3-opt local search using GPU in the travelling salesman problem.

5.2.2 Optimization on Hybrid algorithms

Since all algorithms have their own advantages and disadvantages, researchers combine the advantages of one algorithm with other algorithms to avoid the disadvantages of the algorithm, and thus a variety of hybrid optimization algorithms appear. In our experiment, we can try to combine more optimization algorithms and compare the experiment results, such as genetic algorithm and simulated annealing, genetic algorithm and ant colony algorithm, Hopfield Network and simulated annealing or other local search and etc.

5.2.3 Trials on more variations of TSP

More experiments and trials on different variation of TSP problems to see whether the current approach can improve a particular TSP problem. Hopfield Network is a DL-based algorithm with associative memory, we can see there is a potential in solving other variations of the TSP problem such as the Repeated Vehicle Routing Problem. We can also test whether the Hybrid method can solve the Repeated Vehicle Routing Problem well by changing the location of the city and adding more constraints.

Chapter 6

Conclusions

The TSP problem was first proposed in 1930 and is one of the most in-depth research problems in optimization. With the deepening of research, more technologies are applied to solve this problem. The benchmark for these problems is usually an optimization formula whose cost surface is non-convex and has many local minima, so finding the global minima is a difficult task. Due to the non-convexity of the cost surface, many traditional methods such as gradient descent method will fall into poor local minimums due to different initial values. One of the most straightforward remedies is to choose multiple initial values and the lowest achievable cost value as the potential global minimum. The structure of our new Hybrid Method is to obtain the initial configuration from the Hopfield Network and use the 2-opt Hill Climbing optimization.

This paper represents the methodology and implementation of four algorithms in detail, namely, 2-opt Hill Climbing, classic genetic algorithm, classic Hopfield Network and Hybrid Method. Then all these four algorithms are compared in time complexity, space complexity, performance on solving TSP problems, limitations and advantages. The results show that the hybrid method based on Hopfield network and 2-opt hill-climbing algorithm is slightly improved in solving the TSP problem.

Due to the wide practical application background of TSP, it is believed that TSP will still be a hot issue in algorithm research for a long time in the future. Unless there is a new algorithm framework for solving combinatorial optimization problems, various intelligent algorithms combined with local optimization algorithms will still be the focus of research. To design appropriate operators and local optimization strategies to construct hybrid algorithms will still be an important way to solve the TSP problem and the variations of the TSP problem.

Bibliography

Y. Abu-Mostafa and J. St. Jacques. Information capacity of the hopfield model. *IEEE Transactions on Information Theory*, 31(4):461–464, 1985.

David Bookstaber. Simulated annealing for traveling salesman problem. *SAREPORT. nb*, 1997.

G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958. ISSN 0030364X, 15265463.

J. H. Ramser G. B. Dantzig. The truck dispatching problem. *The Truck Dispatching Problem. Management Science*, 6:80–91, 08 1959.

Şaban Gülcü, Mostafa Mahi, Ömer Kaan Baykan, and Halife Kodaz. A parallel cooperative hybrid method based on ant colony optimization and 3-opt algorithm for solving traveling salesman problem. *Soft Computing*, 22(5):1669–1685, 2018.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

Gilbert Laporte and Yves Nobert. A branch and bound algorithm for the capacitated vehicle routing problem. *Operations-Research-Spektrum*, 5(2):77–85, 1983.

A. Modares, Samerkae Somhom, and T. Enkawa. A self-organizing neural network approach for multiple traveling salesman and vehicle routing problems. *International Transactions in Operational Research*, 6:591 – 606, 08 2006.

Martin Pelikan and David E Goldberg. Genetic algorithms. *MEDAL Report*, (2010007): 1–28, 2010.

Hong Qu, Zhang Yi, and HuaJin Tang. A columnar competitive model for solving multi-traveling salesman problem. *Chaos, Solitons & Fractals*, 31(4):1009–1019, 2007.

Ted Ralphs, L. Kopman, William Pulleyblank, and L.E. Trotter. On the capacitated vehicle routing problem. *Mathematical Programming*, 94:343–359, 01 2003.

Kamil Rocki and Reiji Suda. Accelerating 2-opt and 3-opt local search using gpu in the travelling salesman problem. In *2012 International Conference on High Performance Computing Simulation (HPCS)*, pages 489–495, 2012.

K.C. Tan, Huajin Tang, and S.S. Ge. On parameter settings of hopfield networks applied to traveling salesman problems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 52(5):994–1002, 2005.

D. W. Tank and J. J. Hopfield. "neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52(3):142–152, 1985.

Ilmenau : Voigt. Der handlungsreisende wie er sein soll und was er zu thun hat, um aufträge zu erhalten und eines glücklichen erfolgs in seinen geschäften gewiß zu sein ; mit e. titelkupf. / von einem alten commis-voyageur. -. `https://zs.thulb.uni-jena.de/receive/jportal_jparticle_00248075`. 1832.

Christos Voudouris and Edward Tsang. Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, 113(2):469–499, 1999. ISSN 0377-2217.

Chunxiang Wang and Xiaoni Guo. A hybrid algorithm based on genetic algorithm and ant colony optimization for traveling salesman problems. In *The 2nd International Conference on Information Science and Engineering*, pages 4257–4260, 2010.

Richard Watson, C. Buckley, and Rob Mills. Optimization in "self-modeling" complex adaptive systems. *Complexity*, 16:17–26, 05 2011.

Jinhui Yang, Xiaohu Shi, Maurizio Marchese, and Yanchun Liang. An ant colony optimization method for generalized tsp problem. *Progress in Natural Science*, 18(11): 1417–1422, 2008. ISSN 1002-0071.

Appendix A

Code

Genetic Algorithm on Tsp:

```
import numpy as np
import random

import numpy.random
from tqdm import tqdm
import pandas as pd
from two_opt_TSP import total_distance


class Route:
    def __init__(self, sequence_list):
        self.route_seq = sequence_list

    def __getSeq__(self):
        return self.route_seq

    def __getValue__(self):
        distance = 0

        for point in range(len(self.route_seq) - 1):
            distance = distance + (
                        (map.iloc[self.route_seq[point + 1], 0] -
                        map.iloc[self.route_seq[point], 1]) ** 2 + (
                            map.iloc[self.route_seq[point + 1], 0] - map.iloc[self.route_seq[poi
        distance = distance + ((map.iloc[0, 0] - map.iloc[self.route_seq[-1], 0]) ** 2 + (
                    map.iloc[0, 1] - map.iloc[self.route_seq[-1], 1]) ** 2) ** 0.5
        return distance


def init_pop(population_size, point_num):
    pop_list = []
    while len(pop_list) < population_size:
        sequence_list = list(range(point_num))
        random.shuffle(sequence_list)
        new_indi = Route(sequence_list)
        pop_list.append(new_indi)
```

```
    return pop_list


def get_random(key_list):
    L = len(key_list)
    i = np.random.randint(0, L)
    return key_list[i]


def crossover_mute(list, crossover_rate):
    individual_A = get_random(list)
    individual_B = get_random(list)
    individual_C = get_random(list)
    individual_D = get_random(list)
    parent_A = individual_A if individual_A.__getValue__() < individual_B.__getValue__() else individua
    parent_B = individual_C if individual_C.__getValue__() < individual_D.__getValue__() else individua

    if random.uniform(0, 1) < crossover_rate:
        for i in range(len(parent_A.route_seq)):
            seq_len = len(parent_A.route_seq)
            crossover_point = random.randint(0, seq_len)
            child_seq = parent_A.route_seq[0:crossover_point]
            for state in parent_B.route_seq:
                if state not in child_seq:
                    child_seq.append(state)
            child = Route(child_seq)
        return child
    else:
        child = parent_A if parent_A.__getValue__() < parent_B.__getValue__() else parent_B
        return child


def scaling_window(fitness_list, pop_list):
    if sum(fitness_list) != 0:
        proportion = fitness_list / sum(fitness_list) * len(fitness_list)
        proportion_list = np.ndarray.round(proportion)
    else:
        proportion_list = np.ones(len(fitness_list))
        # pure strategy select
    select_1 = np.where(proportion_list == 1)[0]
    select_2 = np.where(proportion_list > 1)[0]
    if len(select_2) != 0:
        pop_list = 2 * [pop_list[i] for i in select_2] + [pop_list[i] for i in select_1]
    else:
        pop_list = [pop_list[i] for i in select_1]
    return pop_list[:10]


def ga_model(pop_list, epoch):
    eval_list = [state.__getValue__() for state in pop_list]
    fitness_max = [np.min(eval_list)]
    fitness_min = [np.max(eval_list)]

    for i in range(epoch):
        # applying scaling window
        gen_list = [pop_list[np.nanargmin(eval_list)]]
        fmin = fitness_min[0] if i < 6 else fitness_min[i - 5]
        fitness_list = fmin - eval_list
        pop_list = scaling_window(fitness_list, pop_list)
```

```
        while len(gen_list) < len(init_list):
            # one-point crossover
            child = crossover_mute(pop_list, 0.6)
            gen_list.append(child)
        # re-evaluate
        eval_list = [state.__getValue__() for state in pop_list]
        fitness_max.append(np.min(eval_list))
        fitness_min.append(np.max(eval_list))
        pop_list = gen_list

    best_route = [pop_list[np.nanargmin(eval_list)]][0].route_seq
    v = np.zeros([point_num, point_num])
    seq = best_route
    j = 0
    for el in seq:
        v[el, j] = 1
        j += 1
    td, cityx_final, cityy_final = total_distance(v, map, point_num)
    return td, cityx_final, cityy_final


if __name__ == '__main__':
    max_value = 50
    min_value = 0
    point_num = 10
    map = pd.read_csv('state_list.csv')
    DistList_ga = []
    for i in tqdm(range(10000)):
        init_list = init_pop(20, point_num)
        td, cityx_final, cityy_final = ga_model(init_list, 10)
        DistList_ga.append([td, cityx_final, cityy_final])
    df = pd.DataFrame(data=DistList_ga, columns=['distance', 'x', 'y'])
    df.to_csv('DistList_ga.csv')
```

Hopfield Network on TSP:

```
import numpy as np
import pandas as pd
from py2opt.routefinder import RouteFinder
import random
from tqdm import tqdm


# Calculate distance matrix
def distance_cal(df):
    cityx = df['0'].values
    cityy = df['1'].values
    n = len(df)
    d_mat = np.zeros([n, n])
    for i in range(n):
        for j in range(n):
            d_mat[i, j] = np.sqrt((cityx[i] - cityx[j]) ** 2 + (cityy[i] - cityy[j]) ** 2)
    return d_mat


def check_valid(v_mat):
```

```
    # testing whether solution is legal
    t1, t2, t3 = 0, 0, 0
    N = len(v_mat)

    for vx in range(N):
        for vi in range(N):
            t1 += v_mat[vx, vi]

    for x in range(N):
        for i in range(N - 1):
            for j in range(i + 1, N):
                t2 += np.multiply(v_mat[x, i], v_mat[x, j])

    for i in range(N):
        for x in range(N - 1):
            for y in range(x + 1, N):
                t3 += np.multiply(v_mat[x, i], v_mat[y, i])

    if t1 == N and t2 == 0 and t3 == 0:
        return True
    else:
        return False


def init_sequence(N):
    sequence_list = list(range(N))
    random.shuffle(sequence_list)
    return sequence_list


def mask(list):
    m = np.zeros((len(list), len(list)))
    list.append(list[0])
    for i in range(0, len(list) - 2):
        # set mask of Link
        m[i][i + 1] = 1
        m[i + 1][i] = 1
    list.pop()  # delete the last element
    return m


def initial_v(N):
    v = np.zeros([N, N])
    sequence = init_sequence(N)
    j = 0
    for el in sequence:
        v[el, j] = 1
        j += 1
    return v


def hopfield():
    u0 = 0.02
    udao = np.zeros([N, N])
    ctr = 0
    end = 0
    v = initial_v(N)
    while end == 0:
        ctr += 1
```

```
        # v = np.random.rand(N, N)
        v = initial_v(N)
        u = np.ones([N, N]) * (-u0 * np.log(N - 1) / 2)
        u += u * 0.91
        for _ in range(1000):
            for vx in range(N):
                for vi in range(N):
                    j1, j2, j3, j4 = 0, 0, 0, 0
                    # derivative 1 (sum over columns j!=vi)
                    for j in range(N):
                        if j != vi:
                            j1 += v[vx, j]
                            # print(j, vi, j1)
                    j1 *= -A
                    # derivative 2 (sum over rows y!=x)
                    for y in range(N):
                        if y != vx:
                            j2 += v[y, vi]
                    j2 *= -B
                    # derivative 3 (overall sum)
                    j3 = np.sum(v)
                    j3 = -C * (j3 - N)
                    for y in range(N):
                        if y != vx:
                            if vi == 0:
                                j4 += dist_mat[vx, y] * (v[y, vi + 1] + v[y, N - 1])
                            elif vi == N - 1:
                                j4 += dist_mat[vx, y] * (v[y, vi - 1] + v[y, 0])
                            else:
                                j4 += dist_mat[vx, y] * (v[y, vi + 1] + v[y, vi - 1])
                    j4 *= -D
                    udao[vx, vi] = -u[vx, vi] + j1 + j2 + j3 + j4

            # update status and derivatives
            u = u + alpha * udao
            # calculate node value from input potential u
            v = (1 + np.tanh(u / u0)) / 2
            # threshold
            for vx in range(N):
                for vi in range(N):
                    if v[vx, vi] < 0.5:
                        v[vx, vi] = 0
                    if (v[vx, vi] >= 0.5):
                        v[vx, vi] = 1
            # print(v)
            if check_valid(v):
                end = 1
                break
    return v, ctr


def set_route(v, N):
    res_v = np.ones([N, N])
    for i in range(N):
        max_index = np.argmax(v[:, i])
        res_v[max_index, i] = 1
    return res_v
```

```
def total_distance(v):
    cityx_final = np.zeros([N + 1])
    cityy_final = np.zeros([N + 1])
    for j in range(N):
        for i in range(N):
            if v[i, j] == 1:
                cityx_final[j] = cityx[i]
                cityy_final[j] = cityy[i]

    cityx_final[N] = cityx_final[0]
    cityy_final[N] = cityy_final[0]
    # calculate the total distance
    td = 0
    for i in range(N - 1):
        td += np.sqrt((cityx_final[i] - cityx_final[i + 1]) ** 2
                      + (cityy_final[i] - cityy_final[i + 1]) ** 2)
    td += np.sqrt((cityx_final[N - 1] - cityx_final[0]) ** 2
                  + (cityy_final[N - 1] - cityy_final[0]) ** 2)
    return td, cityx_final, cityy_final



if __name__ == '__main__':
    raw_data = pd.read_csv("state_list.csv")

    # get Optimization result from HillClimber
    city_names = list(raw_data.index.array)
    dist_mat = distance_cal(raw_data)

    route_finder = RouteFinder(dist_mat, city_names, iterations=5)
    best_distance, best_route = route_finder.solve()
    print(best_distance)
    print(best_route)
    # Number of cities
    N = len(raw_data)
    # City distances
    cityx = raw_data['0'].values
    cityy = raw_data['1'].values

    A = 500;
    B = 500;
    C = 1000;
    D = 200;
    alpha = 0.0001
    Dist_List = []

    v_ideal = np.zeros([N, N])
    seq = best_route
    j = 0
    for el in seq:
        v_ideal[el, j] = 1
        j += 1

    v = np.zeros([N, N])
    # desired total distance
    ct = 0
    optimal_list = []
    v = initial_v(N)
    td, city_x, city_y = total_distance(v)
```

```
    print([td, city_x, city_y])
    Dist_List.append([td, city_x, city_y])
    for i in tqdm(range(10000)):
        ct += 1
        v, steps = hopfield()
        td, city_x, city_y = total_distance(v)
        Dist_List.append([td, city_x, city_y])
    df = pd.DataFrame(data=Dist_List, columns=['distance', 'x', 'y'])
    df.to_csv('DistList_1.csv')
```

Two Opt on TSP:

```
import pandas as pd
import numpy as np
import random

from tqdm import tqdm


def distance_cal(df):
    cityx = df['0'].values
    cityy = df['1'].values
    n = len(df)
    d_mat = np.zeros([n, n])
    for i in range(n):
        for j in range(n):
            d_mat[i, j] = np.sqrt((cityx[i] - cityx[j]) ** 2 + (cityy[i] - cityy[j]) ** 2)
    return d_mat


def total_distance(v, df, N):
    cityx = df['0'].values
    cityy = df['1'].values
    cityx_final = np.zeros([N + 1])
    cityy_final = np.zeros([N + 1])
    for j in range(N):
        for i in range(N):
            if v[i, j] == 1:
                cityx_final[j] = cityx[i]
                cityy_final[j] = cityy[i]

    cityx_final[N] = cityx_final[0]
    cityy_final[N] = cityy_final[0]
    # calculate the total distance
    td = 0
    for i in range(N - 1):
        td += np.sqrt((cityx_final[i] - cityx_final[i + 1]) ** 2
                      + (cityy_final[i] - cityy_final[i + 1]) ** 2)
    td += np.sqrt((cityx_final[N - 1] - cityx_final[0]) ** 2
                  + (cityy_final[N - 1] - cityy_final[0]) ** 2)
    return td, cityx_final, cityy_final


def two_opt(init_route, N, dist_mat):
    best_distance = calculate_path_dist(dist_mat, init_route)
    best_route = init_route
    for i in range(100):
        for swap_first in range(1, N - 2):
```

```
            for swap_last in range(swap_first + 1, N - 1):
                before_start = init_route[swap_first - 1]
                start = init_route[swap_first]
                end = init_route[swap_last]
                after_end = init_route[swap_last + 1]
                before = dist_mat[before_start][start] + dist_mat[end][after_end]
                after = dist_mat[before_start][end] + dist_mat[start][after_end]
                if after < before:
                    route = swap(init_route, swap_first, swap_last)
                    distance = calculate_path_dist(dist_mat, route)
                    if distance < best_distance:
                        best_route = route
                        best_distance = distance
        init_route = best_route

    j = 0
    v = np.zeros([N, N])
    for el in best_route:
        v[el, j] = 1
        j += 1
    cityx_final = np.zeros([N + 1])
    cityy_final = np.zeros([N + 1])
    for j in range(N):
        for i in range(N):
            if v[i, j] == 1:
                cityx_final[j] = cityx[i]
                cityy_final[j] = cityy[i]

    cityx_final[N] = cityx_final[0]
    cityy_final[N] = cityy_final[0]
    return best_distance, cityx_final, cityy_final


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()


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]]
    path_distance += distance_matrix[path[len(path) - 1]][path[0]]
    return float("{0:.2f}".format(path_distance))


def init_sequence(N):
    sequence_list = list(range(N))
    random.shuffle(sequence_list)
    return sequence_list


def set_route(v, N):
    res_v = np.ones([N, N])
    for i in range(N):
        max_index = np.argmax(v[:, i])
        res_v[max_index, i] = 1
```

```
    return res_v


if __name__ == '__main__':
    raw_data = pd.read_csv("state_list.csv")
    city_names = list(raw_data.index.array)
    dist_mat = distance_cal(raw_data)
    cityx = raw_data['0'].values
    cityy = raw_data['1'].values
    # route_finder = RouteFinder(dist_mat, city_names, iterations=1)
    # _, best_route = route_finder.solve()
    two_opt_list = []
    for i in tqdm(range(10000)):
        N = len(dist_mat)
        init_route = init_sequence(N)
        best_dist, city_x, city_y = two_opt(init_route, N, dist_mat)
        two_opt_list.append([best_dist, city_x, city_y])
    df = pd.DataFrame(data=two_opt_list, columns=['distance', 'x', 'y'])
    df.to_csv('DistList_two_opt_100.csv')
```

Hybrid Methods on TSP:

```
from tqdm import tqdm

from TSP_hopfield import initial_v, check_valid
from two_opt_TSP import distance_cal, init_sequence, calculate_path_dist, swap
import numpy as np
import pandas as pd


def hopfield():
    u0 = 0.02
    udao = np.zeros([N, N])
    ctr = 0
    end = 0
    while end == 0:
        ctr += 1
        # v = np.random.rand(N, N)
        v = initial_v(N)
        u = np.ones([N, N]) * (-u0 * np.log(N - 1) / 2)
        u += u * 0.91
        for _ in range(1000):
            for vx in range(N):
                for vi in range(N):
                    j1, j2, j3, j4 = 0, 0, 0, 0
                    # derivative 1 (sum over columns j!=vi)
                    for j in range(N):
                        if j != vi:
                            j1 += v[vx, j]
                            # print(j, vi, j1)
                    j1 *= -A
                    # derivative 2 (sum over rows y!=x)
                    for y in range(N):
                        if y != vx:
                            j2 += v[y, vi]
                    j2 *= -B
```

```
                    # derivative 3 (overall sum)
                    j3 = np.sum(v)
                    j3 = -C * (j3 - N)
                    for y in range(N):
                        if y != vx:
                            if vi == 0:
                                j4 += dist_mat[vx, y] * (v[y, vi + 1] + v[y, N - 1])
                            elif vi == N - 1:
                                j4 += dist_mat[vx, y] * (v[y, vi - 1] + v[y, 0])
                            else:
                                j4 += dist_mat[vx, y] * (v[y, vi + 1] + v[y, vi - 1])
                    j4 *= -D
                    udao[vx, vi] = -u[vx, vi] + j1 + j2 + j3 + j4

            # update status and derivatives
            u = u + alpha * udao
            # calculate node value from input potential u
            v = (1 + np.tanh(u / u0)) / 2
            # threshold
            for vx in range(N):
                for vi in range(N):
                    if v[vx, vi] < 0.5:
                        v[vx, vi] = 0
                    if (v[vx, vi] >= 0.5):
                        v[vx, vi] = 1
            # print(v)
            if check_valid(v):
                end = 1
                break
    return v, ctr

def two_opt(init_route, N, dist_mat):
    best_distance = calculate_path_dist(dist_mat, init_route)
    best_route = init_route
    for i in range(10):
        for swap_first in range(1, N - 2):
            for swap_last in range(swap_first + 1, N - 1):
                before_start = init_route[swap_first - 1]
                start = init_route[swap_first]
                end = init_route[swap_last]
                after_end = init_route[swap_last + 1]
                before = dist_mat[before_start][start] + dist_mat[end][after_end]
                after = dist_mat[before_start][end] + dist_mat[start][after_end]
                if after < before:
                    route = swap(init_route, swap_first, swap_last)
                    distance = calculate_path_dist(dist_mat, route)
                    if distance < best_distance:
                        best_route = route
                        best_distance = distance
        init_route = best_route

    j = 0
    v = np.zeros([N, N])
    for el in best_route:
        v[el, j] = 1
        j += 1
    cityx_final = np.zeros([N + 1])
    cityy_final = np.zeros([N + 1])
    for j in range(N):
```

```
        for i in range(N):
            if v[i, j] == 1:
                cityx_final[j] = cityx[i]
                cityy_final[j] = cityy[i]

    cityx_final[N] = cityx_final[0]
    cityy_final[N] = cityy_final[0]
    return best_distance, cityx_final, cityy_final

def get_route(v):
    route = []
    for j in range(v.shape[1]):
        route.append(np.argmax(v[:, j]))
    return route


if __name__ == '__main__':
    raw_data = pd.read_csv("state_list.csv")
    city_names = list(raw_data.index.array)
    dist_mat = distance_cal(raw_data)
    cityx = raw_data['0'].values
    cityy = raw_data['1'].values

    N = len(dist_mat)


    A = 500;
    B = 500;
    C = 1000;
    D = 200;
    alpha = 0.0001
    self_dist = []
    init_route = init_sequence(N)
    best_distance = calculate_path_dist(dist_mat, init_route)
    best_route = init_route
    print(best_distance)

    for i in tqdm(range(10000)):
        v, _ = hopfield()
        best_dist, city_x, city_y = two_opt(get_route(v), N, dist_mat)
        self_dist.append([best_dist, city_x, city_y])
    df = pd.DataFrame(data=self_dist, columns=['distance', 'x', 'y'])
    df.to_csv('self_dist_advanced.csv')
```