Combining Neural Networks and Linear Programming for the Travelings Salesman Problem

Jonathan Bogerd

414304

June 1, 2021

Abstract

The use of neural networks has increased rapidly in the previous decades. However, for many operations research problems, they have found moderate success at best, because current algorithms and heuristics are of a high quality and neural networks are not capable of exceeding this. Combining neural networks and standard algorithms has hardly been investigated so far. This thesis will combine both methods in order to find solutions for the traveling salesman problem. The neural network is used for clustering the cities and conventional methods are used to obtain tours from this. This provides better solutions than using standard clustering techniques. Also a way is found to use the neural network to remove edges from the TSP instance without increasing its optimal tour length with more than 2.5%.

Erasmus University Rotterdam

Erasmus School of Economics

Master Thesis Quantitative Logistics and Operations Research

Name of Supervisor: D. Pecin

Name of Co-reader: P. Bouman

Contents

1 Introduction **3**

2 Literature Review **4**

3 Methodology **7**

3.1 Overview 7

3.2 Clustering 7

3.2.1 Design of the ANN 8

3.2.2 Initialization 12

3.2.3 Optimization of weights 13

3.2.4 Conventional Clustering Methods 19

3.3 Finding tours within a cluster 20

3.4 Connecting clusters 21

4 Data **22**

5 Computational Implementation **23**

5.1 Subtours 23

5.2 Multithreading and Network Calculations 24

5.3 Memory 25

5.4 Lin-Kernighan 25

6 Parameter Settings **26**

6.1 CPLEX Settings 26

6.2 Network Settings 26

6.3 Training Algorithm Settings 27

7 Results **27**

7.1 Standard setup 28

7.1.1 Normalization 28

7.1.2 Dimensions 28

7.1.3 Other results 30

7.2 Other setups 30

8 Conclusion **31**

9 Appendix **37**

1 Introduction

The traveling salesman problem (TSP) is one of the most studied combinatorial optimization problems. Many attempts have been made to find efficient solution methods and heuristics. The problem can be summarized in the following way: Given N cities and their relative distances, find the smallest tour that visits all cities exactly once. A detailed description of this very well known problem can for instance be found in Dantzig, Fulkerson, and S. Johnson (1954). This problem arises in many applications. Examples are postal delivery, manufacturing of circuit boards, genome sequencing, finding optimal bus routes, astronomy, optimal production planning etc. On top of these applications, the TSP is also often used as a benchmark for new heuristics and exact solution methods (K. Smith 1996). This thesis focuses on the same principle, because it proposes a new way in which conventional solution methods can be accelerated using neural networks.

In the last decades, the use of neural networks has been increasing rapidly. Several companies and governments use neural networks for many different problems, such as natural language processing, finding patterns in consumer data, predicting stock market prices etc. In the literature, neural networks, in several forms, are also used for solving traditional problems such as the traveling salesman problem. This thesis also uses a neural network for the traveling salesman problem. However, the goal of this paper is to evaluate the possibility of using a neural network together with conventional algorithms and heuristics to solve these problems. It is found that combining neural networks and conventional methods is possible and it yields promising results. A way is found to find tours within 2.5% of the optimal tour length. Clustering by means of neural networks also provides better results compared to for instance k-means.

The setup of this thesis is as follows. First, a description of the current literature on the traveling salesman problem and neural networks is provided and the main concept of reinforcement learning is studied by means of the literature. Then, the methods used in this thesis are provided. This section can be divided in three parts. The first part contains methods on clustering techniques, including neural networks, by far the largest part. This section also provides details of the design of the neural network and the steps of reinforcement learning. The second part explains the ways in which the TSP within the cluster is solved. The last part described how the separate tours can be combined to one tour. Data is the topic of the next section of the thesis. This chapter contains information about the TSP instances that are

used. The fifth section focuses on the implementation of the methods in Java. It provides ways to accelerate the proposed methods. Then the settings and parameters of the experiments are provided. The next chapter contains information about the results, some summary tables and figures and investigates the quality of the proposed methods. Last, a conclusion is provided. In the Appendix, more elaborate results can be found.

2 Literature Review

The traveling salesman problem (TSP) has interested many researchers and has been the subject of many papers. With the introduction of the computer, algorithms were developed to solve larger instances of the problem. One of the most notable papers on this topic is written by Dantzig, Fulkerson, and S. Johnson (1954). In this paper, a branch-and-cut algorithm was proposed and tested on instances of the TSP. This formed a starting point for other researchers. Currently, the best performing exact algorithm is still based on their original paper (Applegate et al. 2006). Next to the exact algorithms, many heuristics were proposed for the TSP. One of these heuristics uses elements from graph theory to find tours that are at most 1.5 times the optimal tour length. It requires finding a minimum spanning tree and perfect matching. If this is done, the tree and matching are combined to form a graph with a Hamiltonian cycle and the TSP is solved (Christofides 1976). Another well-known heuristic is provided by Lin and Kernighan (1973). It involves swapping edges in a tour and is a form of local search. It is still one of the most effective heuristics for solving the TSP (Karapetyan and Gutin 2011). A more recent approach can be found in both Dorigo and Gambardella (1997) and Yuren Zhou (2009). It uses ant colony optimization to find tours. The main idea is to increase the probability of an edge to be used, if the previous shortest tours used the same edge. Then, over several iterations, the algorithm 'learns' shorter tours and thus provides a solution for the TSP. A disadvantage of this heuristic, and many other 'learning' algorithms, is the local optimum trap (Hlaing and Khine 2011). Another solution approach for the TSP involves the use of neural networks.

Artificial neural networks (ANNs) have been around since the 1950's. Early versions consisted of one or two neurons and were only able to learn and solve very basic problems (Daniel 2013). Since then, the networks have improved much in quality and have increased in size. Their main advantage is the promise of solving difficult problems with easy computation. Other advantages include their general usability and the possibilities of parallel computation (Wasserman and Schwartz 1988). Early versions are for instance perceptrons that consist of one layer,

but incorporate the main ideas of weights and activation functions (Rosenblatt 1958). Later on, when weight setting schemes were introduced, larger and deeper networks were introduced. Especially backpropagation, introduced in Rumelhart, Hinton, and Williams (1986), opened new possibilities for larger networks. It requires partial derivatives to calculate the direction and magnitude of weight change to find better weights in each iteration of the training data (Daniel 2013). Of particular interest for this paper is the introduction of Hopfield networks in Hopfield (1982) and the application to the TSP in Hopfield and Tank (1985). In the latter work, a neural network is constructed that is able to provide solutions to the TSP. After this paper, controversy arose, because the obtained results could not immediately be replicated by other researchers (Daniel 2013). However, a.o. Behzad Kamgar-Parsi and Behrooz Kamgar-Parsi (1990) showed the reasons for the discrepancies and the Hopfield network has attracted much attention since then. Numerous improvements have been made since then, for instance by Talaván and Yáñez (2002).

However, the Hopfield approach has several drawbacks in general and specifically for the TSP. For the TSP, K. Smith (1996) argues that the Hopfield approach is limited, because it requires the use of a quadratic formulation of the TSP, which is weaker than for instance the formulation provided by Dantzig, Fulkerson, and S. Johnson (1954). General drawbacks involve often finding infeasible solutions, frequent tuning of parameters and finding local minima, some of which were solved by Takahashi (1997) and Takahashi (1999). Despite efforts from Budinich (1996) and Leung, Jin, and Z.-B. Xu (2004) to improve the Hopfield network and its application to the TSP, no major breakthroughs have been achieved. An interesting idea was used by X. Xu and Tsai (1991) to combine the Hopfield network with known heuristics. This provides solutions of high quality and is an inspiration for this thesis. Other approaches involving neural networks have been tried since then. An attempt was made to solve large TSP instances with a neural network by H. Wang, N. Zhang, and Créput (2017) with reasonable results. The network was able to find solutions fast, however the quality of the solutions was limited. More promising was the attempt by Pasti and Castro (2006) using a neuro-immune network. Neural networks are also used for solving the decision version of the TSP, an NP-hard problem of its own. An example of this can be found in Prates et al. (2019). The training regime that is used in this thesis is reinforcement learning. It has been extensively used in for instance neural networks that play chess with much success. An interesting read on the principles of reinforcement and its use in chess can be found in Silver et al. (2017) and Lai (2015). The last paper contains a description of a chess engine solely build on the concept of reinforcement learning. Apart from

chess, it has also been used in many other contexts such as elevator performance by Crites, Barto, et al. (1996), helicopter flight in Abbeel et al. (2007), medicine in Jonsson (2019) and job-scheduling in W. Zhang and Dietterich (1995).

In this paper, clustering is performed by using a neural network to evaluate a non-standard error function. Clustering can be done in many ways, of which an overview can be found in for instance Du (2010). In this overview, several neural network approaches are shown, along with more conventional methods, such as k-means and hierarchical clustering. These last two mentioned clustering techniques are used as a benchmark for the proposed neural network approach. An example of clustering by a neural network can be found in J. Wang and Yalan Zhou (2009). In this paper, the previously mentioned Hopfield network is used for clustering. Another neural network design that is well suited for clustering is the Kohonen self-organizing map (SOM) (Daniel 2013 and Ali and K. A. Smith 2006). Many examples of the SOM can be found in the literature, for instance in Afolabi and Olude (2007).

3 Methodology

In this section, the methodology of this thesis is discussed. First, an overview of the proposed methods is provided to explain the general ideas of this paper. Then, the three main parts of the methodology are discussed in greater detail.

3.1 Overview

The goal of this paper is to find minimum length tours for TSP instances. It is difficult to find the optimal tour for large TSPs. Therefore, the proposed methods focus on determining tours that are close to optimal. A three-phase approach is used in this paper to find solutions to the TSP. The first part of the solution method is to assign a cluster to all cities of the TSP instance. This is the most important step and the main focus of this paper. An artificial neural network (ANN) is trained with a genetic algorithm to assign a cluster to the cities in such a way that the next two steps will provide a close to optimal tour. The methods to construct the neural network and how to assign a cluster to the cities, will be described in detail in the following sections. The second stage determines the optimal tour within the provided clusters. For this step, well-known methods will be used, specifically as proposed in Dantzig, Fulkerson, and S. Johnson (1954) and a greedy algorithm. The last step connects the tours of the clusters to one tour that visits all cities. An advantage of the proposed method is that the first two steps can be executed in parallel to reduce the computation time drastically. The main goal is to find the shortest tour using an ANN for clustering and conventional solution methods for determining the within-cluster tour and the total tour.

3.2 Clustering

Standard clustering techniques can be very good at minimizing certain error functions. The most common clustering error function is (Du 2010):

$$\frac{1}{N}\sum_{k=1}^{K}\sum_{x_n \in C_k} ||x_n - c_k||^2 \tag{1}$$

where N is the total number of cities, K the total number of clusters, C_k a set of all cities in cluster k, x_n represents a city and c_k is the center of cluster C_k. However, the goal here is not to find clusters such that the average distance to the center is minimal, but to find clusters that yield a tour with a close to optimal length. Therefore, it might be possible to train an artificial neural network to identify criteria to create clusters that are better suited for this

purpose. In order to determine the quality of a certain clustering, a tour has to be constructed, and its length has to be calculated. Next to minimizing the tours length, it is also important to evenly distribute the cities over the clusters, because setting all cities to the same cluster will provide high quality tours, however it will take a very long time to compute these tours. In the following sections, first, the design of the ANN will be discussed. In the next section, the initialization of the network will be considered. Afterwards, a training regime or optimization strategy will be proposed. Lastly, some conventional methods for clustering will be discussed briefly, because the performance of the neural network will be compared to these conventional methods.

3.2.1 Design of the ANN

In general, an artificial neural network consists of nodes, called neurons, and connections between the nodes. The nodes are organized in layers. The connections in the neural network determine which nodes communicate with each other and the weights assigned to the connections determine how strongly a node will react to an input from another node. More information on the structure and concepts of neural networks can be found in Nandy and Biswas (2018) and Kohli, Miglani, and Rapariya (2014).

The goal of the neural network, which will be discussed in this section, is to assign a cluster to all cities. This is done by feeding the neural network all cities sequentially. This means that the neural network will process one city at a time. At the end of processing one city, the neural network will determine a cluster for this city, the city is assigned to that cluster and the network continues with the next city. The main idea here is that there are features of a city that determine in which cluster it belongs. If these features can be determined, they can help to identify the best cluster to which a city can be assigned, such that solving the resulting TSPs will yield close to optimal results.

Setup of the ANN An artificial neural network consists of an input layer, one or more hidden layers and an output layer. The input layer of the neural network has to consist of all features that can help to determine how to assign the city to one of the K clusters. In the next section, the features that are used in this paper are described. It is important to note here, that, in order to keep the network efficient, inputs in the network should not be computationally involved. Another important point is that the order in which the inputs are arranged within the neural network is irrelevant, i.e. the first node is not necessarily more important than the

last node.

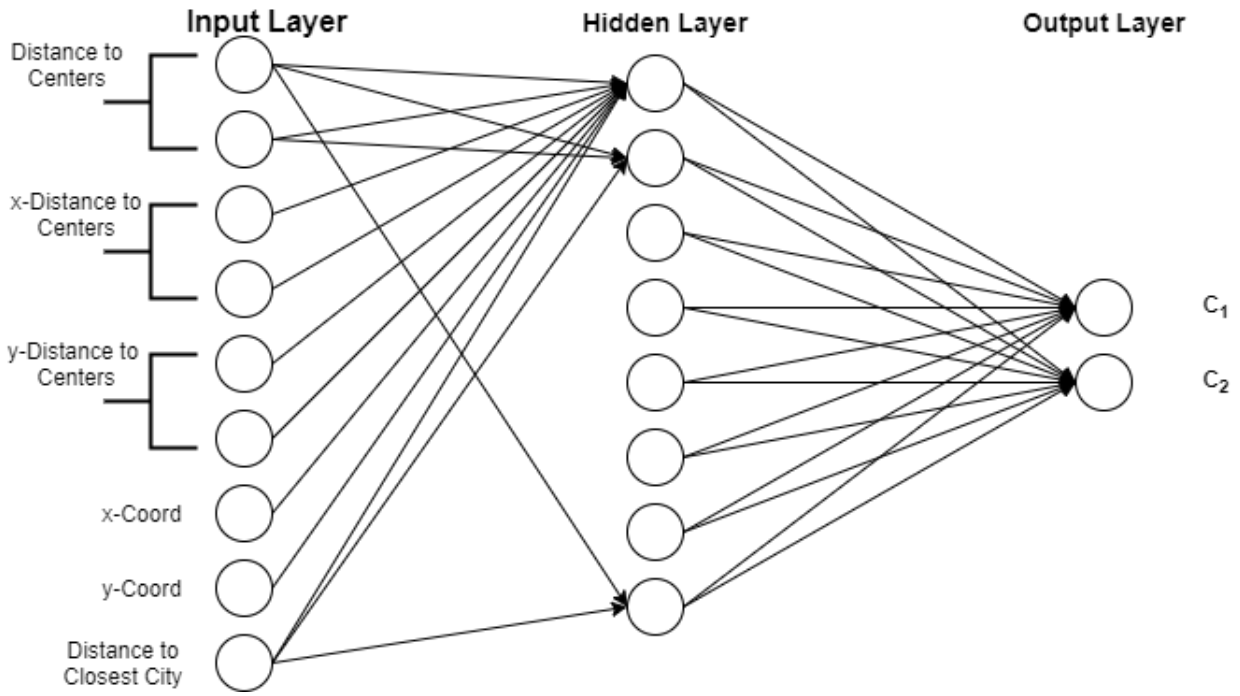
Following Svozil, Kvasnicka, and Pospichal (1997), one hidden layer is used at first. If the network is not able to provide useful clusters, this will be changed to construct a deeper neural network. The number of neurons in the hidden layer will be determined by means of testing, as is described in the literature, e.g. Svozil, Kvasnicka, and Pospichal (1997). In advance, there is no clear way of determining a good starting point for the number of layers or number of nodes within these layers. However, several types of layers exists, all with different properties, advantages and disadvantaged. For instance, pooling or convolution layers could be used to scale down the data, without losing the most important information. Examples of this can be found in Goldberg (2017) and Raghavendra et al. (2018). In this paper, standard hidden layers are used, because the main idea is to test the feasibility of this approach.

The output layer consists of K neurons all corresponding to a cluster. The neuron in the output layer with the highest value is the winning neuron. That is, the city is assigned to the cluster that corresponds to the neuron with the highest output. This is known as the winner-take-all principle. In case two or more output neurons all have the same, highest output, a winner among these is determined at random. In Figure 1 an example of the design of the neural network can be seen. Note that in this case, K, the number of clusters, equals 2. Not all inputs are shown, and only the connection to the first neuron in the hidden layer is depicted, because the figure would otherwise be to crowded.

Inputs The input layer of the neural network has to consist of all features that can help to determine how to assign the city to one of the K clusters. In the end, the researcher has to decide which features will be included. Too few features will result in a poorly performing network because not all relevant information is taken into account. Too many features will result in a very large network, that is impractical to use and difficult to optimize.

The first set of input neurons consider the distance to the center of each cluster. As a first distance measure, the Euclidean distance between the city and the center of all K clusters is used. Next to this distance measure, also the x and y distance to the center of the K clusters is used. So in total, $3K$ input neurons are based on the distance to the center of the clusters. The next 2 input neurons are for the coordinates of the city, one for the x, the other for the y component of the coordinates. For the next 4 inputs, the distance to the closest city of each

Figure 1: Design of the Neural Network



cluster is used. Then the size of the current cluster is considered by the network. The next 4 inputs are again the Euclidean distances between the city and the center of all K clusters, however, this time they are squared. Lastly, a variable number p of closest cities is used as an input. For this, the distances to the closest p cities of that TSP instance are inputted. For most experiments, $p = 0$, however, it will be tested whether or not this variable has any contribution to the quality of the network and the resulting tours.

In order to make the network usable for all TSP instances, the mentioned inputs need to be scaled. As is common in neural networks, see Nandy and Biswas (2018) for instance, the inputs will be scaled between 0 and 1. For this scaling, first the $xMax$, $xMin$, $yMax$ and $yMin$ of the TSP instance is set to the highest or lowest x or y value of all city coordinates. The maximum Euclidean distance (MED) then is defined as:

$$\sqrt{(xMax - xMin)^2 + (yMax - yMin)^2} \tag{2}$$

The scaling of all Euclidean distances is done by dividing by this MED. Scaling of x and y distances is done by dividing by the maximum difference of these distances. Coordinates are scaled by:

$$\frac{t - tMin}{tMax - tMin} \tag{3}$$

with $t \in \{x, y\}$.

The cluster size is scaled by dividing the current cluster size by the allowed value, which is based on the parameter d, which will be discussed in more detail later on in this paper.

Processing within the Network Once the inputs for a certain city are determined, the network needs to process these inputs in order to get the outputs. This processing occurs within the neurons of the network. Within a neuron, two steps are performed. Firstly, the so-called activation is performed. This is the weighted sum of all inputs and bias, which is a constant (Agatonovic-Kustrin and Beresford 2000). So, the activation value of a neuron can be calculated by:

$$v_{ji} = \sum_{t=1}^{T} w_{(j-1)t} x_{(j-1)t} + b_{ji} \tag{4}$$

in which j is the index of the layer, i the index of the neuron within layer j and T the number of neurons in layer $j-1$. v_{ij} then, is the activation value for neuron i in layer j, $w_{(j-1)t}$ the weight of the connection from neuron t in layer $j-1$ to neuron i in layer j. $x_{(j-1)t}$ is the respective activation value, or input. b_{ij} is the bias for neuron i in layer j. The second step is the transfer function of the neuron. In this paper, the Sigmoid function is used, as it is most common in the literature, as can be seen in for instance K. A. Smith (1999) and Svozil, Kvasnicka, and Pospichal (1997). The output value of the neuron is (Daniel 2013):

$$y_{ji} = \frac{1}{1 + \exp(-v_{ji})} = f(v_{ji}) \tag{5}$$

The output y_{ji} then forms the input of the neurons in the next layer. The ANN 'learns' by finding weights that minimize some error function. To calculate the error, first all cities have to be assigned to a cluster by the neural network. Then, the remainder of the algorithm determines the length of a tour. The error function has to take into account the length of the tour that is found, but it also has to evaluate the clustering. If for instance all cities are put in the same cluster, this will result in a close to optimal tour, however the computation time will be very long. Therefore, the network will be penalized if too many cities are within the same cluster. The error function is defined as:

$$E = \frac{R - R_{opt}}{R_{opt}} * 100 + \sum_{k=1}^{K} \max(N_k - \frac{d * N}{K}, 0) \tag{6}$$

in which R is the length of the obtained tour, R_{opt} the length of smallest known tour, K the total number of clusters, N_k the number of cities assigned to cluster k, d a constant and N

the total number of cities in the problem instance. The first part of the error function adds the difference between the obtained tour and the best known tour. The second part penalizes having too large clusters. The constant d defines how large clusters are allowed to be, before a penalty is incurred.

3.2.2 Initialization

Before the network can be used, two components need to initialized. Firstly, the centers, that are required for most inputs, need to be set. Secondly, the weights and biases for the network have to be set in order to have a network that can find close to optimal tours.

Initialization of Centers The k-means algorithm sets the centers at random by picking the required number of cities at random at the beginning of the algorithm, see Du (2010). This would not be a good start for the network approach, because for instance the coordinates inputs require a more fixed center. Therefore, the centers are set in the following way:

$$c_{kt} = v * (tMax - tMin) + tMin \tag{7}$$

where c_{kt} is the coordinate t of the center of cluster k, $k \in K$ and $t \in \{x, y\}$. v is set such that the centers of the clusters are the middle of four, equal quadrants of a rectangle containing all cities of the TSP. Therefore, $v \in \{0.33, 0.67\}$.

Initialization of Weights and Biases To set the weights and biases of the artificial neural network, several options are investigated. The literature suggest to use relatively small weights (Montana and Davis 1989, Daniel 2013 and Svozil, Kvasnicka, and Pospichal 1997). The first option is to set the weights and biases randomly, using a uniform distribution. The second options is similar, however, a $t-$distribution with 5 degrees of freedom is used, as was proposed in Svozil, Kvasnicka, and Pospichal (1997) or Daniel (2013). This centers more values around 0, but allows for values to exceed the boundaries of the proposed uniform distribution. In these two, random initialization strategies, no prior knowledge on which input could be important is used and the network tries to figure this out by itself. The expectation therefore is, that it will take longer to find a good network, based on it's output, than if prior knowledge is used. However, it might be able to find settings, that were not known beforehand, and that provide

good solutions.

In this paper, including the random and knowledge-based initializations, 18 network settings for the weights, and 6 network settings for the biases are used. The description of the settings can be found in Table 1 and Table 2. The mentioned process of normalization will be discussed in the next section, because it also plays in important role in optimizing the network. Note that also a kmeans like clustering approach can be implemented using the neural network. The only difference between the standard kmeans and the network implementation is that the clusters and their initial centers are fixed in the described way, as opposed to being randomly selected. In general, if the network consists of at least one hidden layer, the first node of the hidden layer will be regarded as a neuron for cluster 1, the second node of the hidden layer for cluster 2 and so on. For all remaining hidden layers, the first node of that hidden layer is connected to the previous hidden layers first node, with weight 1. Note that the order of these connection are without loss of generality. From this setup, it is clear that prior knowledge-based initializations, can be beneficial for networks with no or at most one hidden layer, because it does not really use the benefits of multiple hidden layers. It is therefore expected that for deeper networks, if the solution method can find close to optimal tours, the prior knowledge-based settings will be outperformed by a completely different setting, which was found by the optimization strategy that will be discussed in the next section.

3.2.3 Optimization of weights

Within deep learning, several learning methods exist. Supervised learning is used if the model is trained on data for which the output is already labeled. Examples and a discussion of this can be found in Cunningham, Cord, and Delany (2008). In this case, the clusters are not labeled, because the best cluster is not known in advance. Therefore, supervised learning is not possible. Unsupervised learning is a method in which the labels do not exist in advance, however the model tries to identify patterns itself (Ghahramani 2003). Combinations of these also exist and are discussed in detail in the literature. In this paper, a different learning method is used, called reinforcement learning. In reinforcement learning, the model is trained based on rewards (Sutton and Barto 2018). In this case, the rewards are awarded by the error function, which calculates the performance of a network and gives this score to the network as a reward. In this section, a detailed description of the learning process is given. First, training and test data is discussed, then the genetic algorithm is introduced and all components are described.

Table 1: Description Network Initializations Weights

Setting	Description	Type
0	zero weights	Constant
1	random weights uniform	Random
2	random weights uniform, normalized	Random
3	random weights t-distribution	Random
4	random weights t-distribution, normalized	Random
5	constant weights, normalized	Constant
6	kmeans weights: only center distances is set to 1	Knowledge
7	kmeans and cluster size weights: only center distances and cluster sizes weights are set to 1	Knowledge
8	cluster size weights: only cluster size weights are set to 1	Knowledge
9	squared weights: only squared center distances is set to 1	Knowledge
10	squared weights and cluster size weights: only squared center distance and cluster size weights are set to 1	Knowledge
11	kmeans, squared and cluster size	Knowledge
12	only x and y distance	Knowledge
13	only x and y distance + cluster size	Knowledge
14	only distance to nearest city	Knowledge
15	only distance to nearest city + cluster size	Knowledge
16	kmeans,cluster size, x and y distance, distance to nearest city, squared weights	Knowledge
17	xy coords, kmeans,cluster size, x and y distance, distance to nearest city, squared weights	Knowledge

Table 2: Description Network Initializations Biases

Setting	Description	Type
0	zero biases	Constant
1	random biases uniform	Random
2	random biases uniform, normalized	Random
3	random biases t-distribution	Random
4	random biases t-distribution, normalized	Random
5	constant biases, normalized	Random

Afterwards normalization, a key part in the learning and initialization strategies is explained and lastly a picture of the entire process is provided.

Test and Training Data In order to train the neural network, the data is split in two parts, the training data and the test data, which is the standard in machine learning. An example of this can for instance be found in Kohli, Miglani, and Rapariya (2014). The test data is used in the end of the training process to calculate the performance of the network. The network is trained on the training data set. In this way, overfitting is less of an issue and a less biased result is obtained (Daniel 2013). The training data is also split up in parts, in order to speed up the algorithm in the next section. So instead of running the algorithm on the entire training set, a subset of this training set is used to evaluate the performance of the network.

Genetic Algorithm A genetic algorithm is an algorithm based on the theory of evolution. The main concept is to select good instances of the neural network to create a new set of instances and to discard bad performing instances. This very much resembles natural selection within evolution. In the literature, genetic algorithms are often used to solve optimization problems. A well-known example is ant colony optimization for the TSP in for instance Dorigo and Gambardella (1997).

In each iteration of the algorithm the performance of a set of networks is evaluated. The set of networks is often called a generation. In order to generate the next generation, a parent pool is selected, which is described in the following section. These parents then are combined or transformed into children, which form the next generation. This transformation or combination is often referred to as mutation. For this mutation, there is a trade-off between detailed local search and the risk of getting stuck in a local optimum, see Montana and Davis (1989). This, and ways to deal with this issue, are discussed in the section on Mutations. To improve the quality of the networks and to avoid ever increasing weights, normalization is also introduced.

Parent Pool After an iteration of the genetic algorithm is performed, a parent pool is created in order to create next generation of networks. The maximum size of the parent pool is $2P$. In each iteration the $P/2$ best performing networks are added to the parent pool. Then, the parent pool is trimmed to be at most the maximum size of $2P$, by removing the worst performing networks from the parent pool. In order to improve the algorithm, the maximum size of the parent pool is dynamically increased during the algorithm. The exact specifications

for this can be found in the section Parameter Settings.

To create the children for the next generation, a random subset from the parent pool is created, where the probability to select a network as parent is higher if the performance of the network is better. If a small difference in performance increases the probability to be selected to a large extend, the pool is narrowed down more, than if the probability is only increased slightly. However, it is expected that the genetic algorithm convergences faster if the probability is increased quickly. This again is part of the trade-off in many local search like algorithms. In this paper, the probability that a member of the generation is selected as a parent for the next generation is given by:

$$(\frac{1}{e_t})/(\frac{1}{\sum_p^P e_p})^2 \tag{8}$$

in which e_t is the error of network t and P is the set of all networks currently in the parent pool. This probability setting ensures that good performing networks are more likely to become a parent. In order to avoid too narrow of a parent pool, the initial networks, described in Tables 1 and 2 are never removed from the parent pool. This means that they can always be selected as a parent, however, the same probability calculation applies, resulting in bad settings to be selected infrequently.

Children Children are created in the genetic algorithm by mutating parents, such that traits of the parents are inherited by the children. In the context of networks, possible traits can be the weights and the biases for each neuron. Standard mutation strategies are used in this paper, such as crossovers, in which each trait is randomly selected from one of the two parents. However, for networks, the relationship between the weights is very relevant. For instance, if a network in one hidden layer is used and both parents have a different ordering of hidden layer neurons in relation to the output layer, a crossover of the weights between input and hidden layer from one parent and a crossover from between hidden layer and output layer of the other parent, will in general not provide a network of similar quality to the parents. Therefore, in the section on Mutation, more ordered versions of mutations are described as well as randomized crossovers.

The starting networks of the genetic algorithm are provided with a list of mutation strategies. The first parent always passes this list on two the children and the two parents generate a child for each mutation strategy attached to the first parent. This means that, if for instance the

Table 3: Mutation Strategies

	Name	Description
1	Crossover	Random from two parents
2	Organized Crossover	Random from two parents, based on grouping
3	Large Random	Random value from standard normal
4	Small Random I	Random value from standard normal, logarithmically scaled with iterations
5	Small Random II	Random value from standard normal, scaled with iterations
6	Zero Entries	0.5 or -0.5 for zero entries
7	Non Zero Entries	0.5 or -0.5 for non-zero entries

parent has two crossover mutations related to it, two children are created, both with the two crossover mutations related to them as well.

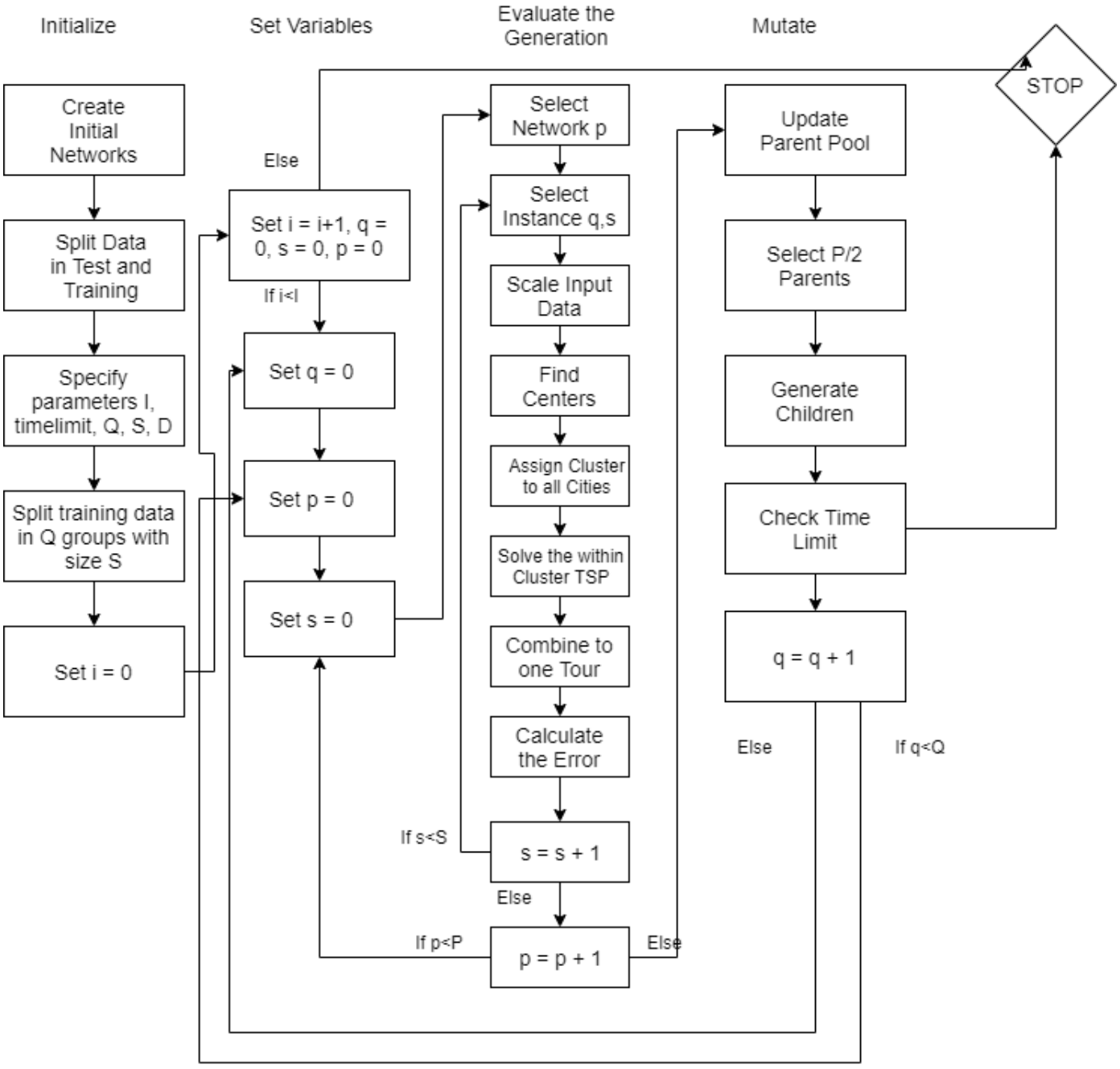
Mutation In this paper 7 different mutation strategies are used, that can be split into two groups: crossover mutations and random mutations. For the crossover mutations there is a standard crossover mutation strategy and an organized crossover mutation strategy. The standard crossover takes a weight or bias with equal probability from the two parents. The organized crossover tries to capture the relations within the network as well by grouping the inputs that belong together and choosing all weights and biases from that group from one of the two parents. For instance the inputs regarding Euclidean distance are combined into one group.

There are three random mutation strategies that are very similar. All take into account the structure of the networks, in the same way as the organized crossover. The large random mutation sets the weights with probability h to a random value from the standard normal distribution. The other two mutation strategies divide a random value from the standard normal by either the iteration count or $\log(iteration + 2)$. The idea between these last mutations is to create large deviations in the beginning of the algorithm, but to deviate less and less during the algorithm runs. This idea is borrowed from local search and gradient descent and can be found in for instance Michiels, Aarts, and Korst (2007). The last two random mutations are centered around zero or non zero values. The first sets a weight to -0.5 or 0.5 with probability h, if the weights was zero, the other does the same for values that are non zero. In Table 3 the mutation strategies are summarized.

Normalization Networks are very sensitive to the scale of the weights. If two parents have entirely differently scaled weights and they are combined, it is expected that the resulting network will perform very poorly. However, in the Results section, this hypothesis will be tested and the results will be discussed. If normalization is used, the following technique is used. Firstly, for the weights, the norm, that is the absolute value, of all inputs to one neuron is summed. If this is non-zero, all weights for this neuron are divided by this sum of norms, such that the new summed norm is equal to 1. For the biases, the same strategy is used.

Overview Picture In Figure 2 the training setup of this paper is visualized. Firstly, some initial networks are created, using the settings in Tables 1 and 2. Then, after the data is split in training and test date, the training data is split in Q groups. Then, iteratively, the network assigns a cluster to each city, the within cluster TSP is solved, see one of the next sections, one tour is created and the error is calculated. Then after all instances of that group are processed, the resulting average error is saved, all cities are removed from the cluster and the next network starts the process. After all networks of that generation are evaluated, the parent pool is updated by adding them all to the parent pool and removing the worst performing networks until the parent pool has the maximum size of $2P$. Then, parents are selected, children generated and the algorithm starts again, until the required number of iterations are performed or until a certain amount of time has transpired.

Figure 2: Algorithm Overview



3.2.4 Conventional Clustering Methods

Many clustering algorithms exists in the literature. In this thesis, these are only used as a comparison to the provided neural network. The two methods that are used are k-means and hierarchical clustering. In k-means, the centers are set at random cities at the start of the algorithm. Then, over all remaining cities, the closest city to a cluster is added to that cluster, the center of that cluster is recalculated and the next iterations starts, until all cities are assigned to a cluster. For hierarchical clustering, at the start, all cities are treated as centers of their own cluster. Afterwards, the two most similar clusters are identified and merged and

the next iterations starts, until one cluster remains. For implementations of these clustering algorithms, see for instance Du (2010), Kanungo et al. (2002) or S. C. Johnson (1967).

3.3 Finding tours within a cluster

After the clustering is performed, sub-TSPs arise. Within each cluster, a TSP has to be solved to find a good or optimal tour within the cluster. To do this, standard TSP algorithms or heuristics can be used. In this paper, as the number of the cities within a cluster will remain relatively small, an exact algorithm is used to solve the TSP as well as a heuristic. CPLEX in combination with Java is used to solve the following formulation given by Dantzig, Fulkerson, and S. Johnson (1954):

$$\min \sum_{i=1}^{N} \sum_{j=1, j\neq i} c_{ij} x_{ij} : \tag{9}$$

$$0 \leq x_{ij} \leq 1 \quad i, j = 1, ..., N \tag{10}$$

$$\sum_{i=1, i\neq j}^{N} x_{ij} = 1 \quad j = 1, ..., N \tag{11}$$

$$\sum_{j=1, j\neq i}^{N} x_{ij} = 1 \quad i = 1, ..., N \tag{12}$$

$$\sum_{i\in Q} \sum_{j\neq i, j\in Q} x_{ij} \leq |Q| - 1 \quad \forall Q \subseteq \{1, ..., N\}, |Q| \geq 2 \tag{13}$$

in which the last set of constraints are the so-called subtour elimination constraints. There are exponentially many subtour constraints, so they are added as lazy constraints within CPLEX. For clusters with less than 100 cities, CPLEX is usually able to find the optimal solution within a few seconds. The CPLEX implementation of the TSP formulation is preferred here to a specific TSP solver, because several techniques can be used then to speed up the training of the network. This is possible because, instead of solving a TSP once, in order to train it is required to solve similar TSPs hundreds of times. In the section Computational Implementation, details on how this can be used to speed up the algorithm are discussed.

The heuristic that will be used in this paper is a standard, nearest neighbour algorithm. In essence, it starts a random city and then connects the closest, not yet connected neighbour. This process is much faster than the TSP formulation, however, the results are typically around 15% off from the best possible solution. More details on the nearest neighbour algorithm and a discussion on the performance can be found in Kizilateş and Nuriyeva (2013) and Gutin, Yeo, and Zverovich (2002).

3.4 Connecting clusters

If all cities are added to a cluster and the optimal tour is obtained within all clusters, the last remaining task is to combine the tours of the clusters to one tour covering all cities. Note that this step can influence the quality of a solution to a large extend. Therefore, this is not a trivial step. One way to solve this problem is proposed by Phienthrakul (2014). In this paper, first the centroids of all clusters are calculated. Then, the distance between the pair of clusters is calculated. For the closest pair of clusters, the distance of the centroid of cluster 1 and all points in cluster 2 are calculated. The closest point is selected, denoted by A. The same is done in reversed order, denoted by B. These two points will be connected. Next, for cluster 1, the closest point in the obtained tour to A will be linked, together with the remaining tour. The same is done for cluster 2. Then, the process is repeated for all clusters, until 2 endpoints remain. These will also be connected, and a solution to the TSP is found.

Another way to connect the clusters, is to evaluate all distances between any two points in two clusters, after which you connect them in a similar way as in Phienthrakul (2014). A disadvantage of this method, is the large number of computations to be performed, compared to the previous method. Even though the number of clusters will be fairly limited, this does not have to be a problem. This method will be more likely to find better solutions, because no approximation of distance is made by means of the centroid.

The last, more experimental, idea is to remove all within cluster edges from the TSP that are not used in the within cluster tours. Then, solve the initial TSP, without the within cluster edges and using the already obtained tour as initial solution for the CPLEX solver and by inheriting the still relevant subtour elimination constraints that were already found. Note that this method does not guarantee finding the optimal solution, because a sub-problem of the original problem is solved in the last step.

All three presented methods to combine the clusters yield one global tour that visits all cities. Many methods to improve the quality of a tour are known, most notably by Lin and Kernighan (1973). This could be a good addition to improve the quality of the obtained tour, however, it can also hurt the provided algorithm. The neural network is trained by comparing its tour to the best tour known. The Lin-Kernighan heuristic can make the link between the clusters and the obtained tour less clear and hence 'confuse' the network. Therefore, both

options are investigated in this paper.

4 Data

For this research, a data set of TSP instances is required. Several sources are available, in this case the TSPLIB is used, the most common test data set for the TSP (Reinelt 1991). This dataset consist of over a 100 TSP instances, varying in size from 14 to 85900 cities. In order to train a network, the number of centers for the network is fixed. This also limits the range of number of cities a specific network can handle, in order to avoid a long computation time. Therefore, the data set is split in groups with similar city size. A summary is shown in Table (4).

Table 4: Summary TSPLIB

	4 Centers	6 Centers	8 Centers	10 Centers
Number of Cities	50-200	200-400	400-600	600-1000
Number of Available Data Sets	35	10	12	7

In order to train a neural network, a larger data set is required. However, adding new, random data sets might not be a good strategy, because several authors mention that an instance from TSPLIB is easier to solve than a random data set, due to the underlying structure (Fischer et al. 2005 a.o.). Therefore, the three largest data sets from TSPLIB, rl11849, usa13509 and pla85900, are used to generate more data sets of a given size, by randomly selecting the specified number of cities from the data sets. In total, 1000 TSP instances will be required to train the network. A further 200 problem instances will be used for testing the quality of the proposed solution approach, based on TSP instances for 4 centers, the main focus of this paper. Therefore TSP instances had to be created, Table 5 describes the number of random TSP instances created out of rl11849, usa13509 and pla85900.

In most papers on the Traveling Salesman Problem, all distances between cities are rounded to be integer, either rounded down, or rounded to the nearest integer. To follow the literature,

Table 5: Generated Instances of TSP

Instance	Created Instances
rl11849	265
usa13509	400
pla85900	500

this is also done in this paper, where all distances are always rounded to the nearest integer. Another important point is the optimal route for all the 1200 instances. To test the quality of a newly found solution, by the network or any other algorithm, it can be compared to the optimal solution. Therefore, the optimal solution of all instances needs to be determined. This is done by CPLEX using the aforementioned formulation of Dantzig, Fulkerson, and S. Johnson (1954). The subtour elimination constraints are added as lazy constraints. Once a solution is found, an algorithm runs to check for any subtours in the solution. If any is present, the specific constraint is added and CPLEX continues solving the Linear Program.

5 Computational Implementation

In this section, several remarks will be made on the implementation of the methods proposed in this paper. First some details regarding the CPLEX formulation of the TSP, specifically on the subtours. Then, multithreading and the network calculations are discussed. Next, several memories are discussed, designed to speed up the algorithm. Last, the exact implementation of the Lin-Kernighan algorithm is provided.

5.1 Subtours

The TSP is particularly difficult because of the subtours. In the formulation of Dantzig, Fulkerson, and S. Johnson (1954), the subtour elimination constraints are the most difficult and time-consuming, and in other algorithms, avoiding subtours can result in far from optimal solutions. In order to speed up the algorithm, it is therefore required to find a good way to handle subtours efficiently. This is done in two ways. Firstly, as a first step for finding the optimal solutions to the TSP instances, an algorithm is used to determine subtours that are very likely to occur. These subtours are then saved and always added to the subtour elimination constraints if they are relevant for that specific problem. An easy way to find subtours that are very likely to occur is by using the nearest neighbour algorithm, with a small change. Instead of prohibiting the algorithm to use a city it has already used, this is allowed. Then, if a subtour is found in this way, the nodes within the subtour are removed from the nodes to connect, the subtour is saved and the algorithm is rerun without the nodes in the subtour. This ends if a tour is found that contains all remaining cities, without that route containing a subtour. These potential subtours are known a priori, before any other algorithm is ran and are added to the list of known subtours for that instance. By means of experiment it is found in this paper, that

using the subtours found by this algorithm will speed up the Linear Program formulation of the TSP by around 40%, greatly improving the performance of the initial CPLEX run but also the performance of the training algorithm.

Above, a list of known subtours is mentioned. This is the second way to efficiently handle subtours. After solving the formulation in Dantzig, Fulkerson, and S. Johnson (1954), all subtours that are found in the process are added to the list of known subtours, together with the subtours already known a priori. Then, for the next iteration of the training algorithm, this list is used to determine all known subtours that are relevant to the particular problem. The algorithm namely solves the TSP within a certain cluster. In order then to add all relevant subtours, it iterates over the list of known subtours and adds the subtours that are entirely within that cluster. Then, before solving the within cluster TSP, it first runs the algorithm to find new subtours, specific for that cluster and adds any new subtours that were not known. In order to efficiently determine if a subtour is already in the list, it compares the hash codes instead of the entire tour, as this process is much faster. Saving all subtours in the list of known subtours can result in memory issues, however this was not a problem in this paper.

5.2 Multithreading and Network Calculations

It is important to note that normally CPLEX runs on as many cores as are available. The branch-and-bound algorithm used in CPLEX is very well adapted for this. However, when lazy constraints are used, multithreading is not longer possible, because nodes in the branch-and-bound tree are not longer independent. This is a huge disadvantage, because it will slow down the training algorithm by a factor (almost) equal to the number of cores available. This issue is resolved by running the training algorithm itself in parallel. If a network is selected in the training algorithm, it has to assign a city to a cluster for each TSP instance, then solve the within cluster TSP and then combine this all in one tour. Doing this for one TSP instance, does not the outcome of running the same steps for a different TSP instance, these steps are independent of each other. Therefore, these steps are run in parallel, by creating a specific thread for these 3 steps per instance. Creating all threads at once for the network is not recommended, so batches are used, with the size of a batch a multiple of the number of cores available. The creation and execution of the threads is done by using the java.util.concurrent package. The algorithm will wait with creating a new generation of networks until all threads are executed.

An advantage of using neural networks is the ease of calculation. Performing all calculations

on a per neuron basis is very inefficient, because it can be done by using matrix operations. Java does not contain a standard library for performing matrix operations, so the apache.commons package was used to perform these steps. In order to find the output of all neurons in a specific layer, instead of using 4 and 5, the following matrix operations can be used:

$$y_j = f(W_j * y_{j-1} + b_j) \tag{14}$$

in which, y_j is the vector of outputs of layer j, f is the Sigmoid function provided in 5, W_j is the matrix of weights of layer j. y_{j-1} is the vector of outputs of layer $j-1$ and b_j is the vector of biases of layer j.

5.3 Memory

In the section on subtours, it was mentioned that the training algorithm keeps track of the subtours in order to speed up the training algorithm. The same is true for networks and clusters. For each cluster, for which the within cluster TSP is solved, the algorithm keeps track of the solution. Then, if another network assigns cities to a cluster in a way such that the cluster is the same, the algorithm immediately can find the solution and does not have to resolve the within cluster TSP. In a similar way to subtours, clusters are found by comparing hash codes, to speed up this process. A cluster is deleted from the memory if it is not used often enough, the exact setting of this will be specified in the next section. Clusters that are very small, again the exact setting will be specified, will not be added to the memory, because solving the within cluster TSP can be done relatively fast for these clusters and if they were added, it would slow down finding the larger clusters.

5.4 Lin-Kernighan

Variations of the Lin-Kernighan algorithm are used in this paper to improve found solutions. The effect of using this algorithm on the performance of the network will be tested as it might 'confuse' the training of the neural network. Specifically, swaps between pairs, triples and quadruples of edges are used, corresponding to 2-, 3- and 4-opt. If an improvement is found, the edges are swapped, the new length of the total tour is calculated and the algorithm is restarted. Calculating the new length can be done efficiently by removing the distance of the removed edges and adding the distance of the new edges compared to recalculating the length from scratch. More details of this algorithm can be found in Lin and Kernighan (1973).

6 Parameter Settings

In this section, all parameter settings will be discussed that are used to obtain the results. First, CPLEX settings will be described. Next, network settings are mentioned and lastly the settings of the training algorithm are described.

6.1 CPLEX Settings

In this paper, the CPLEX Optimizer from IBM is used in combination with Java. In this paper the version number of CPLEX used is 12.9 which was released in 2019. The standard CPLEX solver settings are used, with the addition of a time limit and a gap. For each Linear Program instance, the time limit is set to 3600 seconds, which is an hour. For all small instances, the solving time is not even close to the hour time limit, however for the largest instances this is a valid limit. The Mixed Integer Program Gap (MIPGap) is set to 0.01%. If the instances was not solved to optimally in the available time, the best known solution at that point will be used.

6.2 Network Settings

Several settings have to be discussed regarding networks. The most important of the settings is the dimension of the network. In this paper the following dimensions are chosen: 30-4, 30-15-4 and 30-20-15-4 in which the first number corresponds to the size of the input layer, the last number corresponds to the size of the output layer and all numbers in between correspond to the size of the hidden layers. The size of the output layer is fixed, there are in total 4 centres, however, the remaining size of the network was chosen on the basis of remarks in Daniel (2013). The main idea there is to set the size of the hidden layers small initially and, if this results in sub-optimal networks, increase the size of the hidden layers gradually. The initial networks used are those described by the weights and biases in Table 1 and Table 2. This yields a total of 108 initial networks.

To these networks, mutation strategies are attached that describe how to generate the next generation. In Table 3 the possible mutation strategies can be found. In Table 6 the parameter settings for the mutation strategies can be found. For the crossover mutations, h is the probability that a weight is taken from parent 1, where $1 - h$ is the probability that a weight is taken from parent 2. For the other mutation strategies, h is the probability that a weight is randomly mutated.

Table 6: Mutation Settings

	Name	Number of this Mutation Strategy	Parameter
1	Crossover	5	$h = 0.5$
2	Organized Crossover	10	$h = 0.5$
3	Large Random	10	$h = 0.1$
4	Small Random I	10	$h = 0.1$
5	Small Random II	10	$h = 0.1$
6	Zero Entries	3	$h = 0.1$
7	Non Zero Entries	3	$h = 0.1$

6.3 Training Algorithm Settings

For the training algorithm, some parameters have to be set. The first one of these settings is the number of iterations of the training algorithm. This is set in two ways. Firstly, the number of iterations is set to a maximum of $I = 30$, with I the number of iterations. Secondly, the total time required for running the algorithm may not surpass 1200 seconds, unless otherwise specified. The instances are split in $Q = 4$ groups of equal size. The maximum size of the parent pool P is set to 50, but it will increase with:

$$P_{i+1} = ceil(P_i * (1 + i * 0.2)) \tag{15}$$

where P_{i+1} is the size of the parent pool in iteration $i + 1$, and i is the current iteration.

In order to calculate the error, using equation 6, the parameter D has to be set. This parameter specifies the allowance of large clusters. If D is increased, the penalty for having unequal sized clusters is reduced. In this paper, $D \in \{1, 1.5\}$. Furthermore, clusters are removed from the cluster memory if that cluster is used in less than 20% of the cases. Known tours are only searched in the current memory if the size of the cluster is 20 nodes or larger in order to avoid filling the memory with tours that can easily be identified.

7 Results

In this section the results of the experiments run will be presented. Firstly, the standard setup is run in order to test the result of normalization, to obtain results on networks versus standard clustering techniques and to evaluate the learning process of the training algorithm. Then other variations of the training algorithm are run, for instance with a different within cluster tour algorithm, to compare them to the standard setup.

Table 7: Results Standard Setup

Network Setup	Not Normalized Error	Normalized Error	KMeans Error	KMeans A. Error	HC Error
30-4 d=1	30,19	28,19	37.47	34.80	81.64
30-4 d=1.5	20,04	18,85	22.61	21.83	54.19
30-15-4 d=1	30,41	29,37	37.47	34.80	81.64
30-15-4 d=1.5	21,31	20,68	22.61	21.83	54.19
30-20-15-4 d=1	34,23	32,11	37.47	34.80	81.64
30-20-15-4 d=1.5	21,76	21,54	22.61	21.83	54.19

7.1 Standard setup

The standard setup contains the nearest neighbour algorithm to determine tours within clusters and the connect clusters by the method proposed in section 3.4 by Phienthrakul (2014). The full results of this experiment, the average of 10 runs of the training algorithm, can be found in Table 8 in the Appendix. The most important results are shown below in Table 7.

7.1.1 Normalization

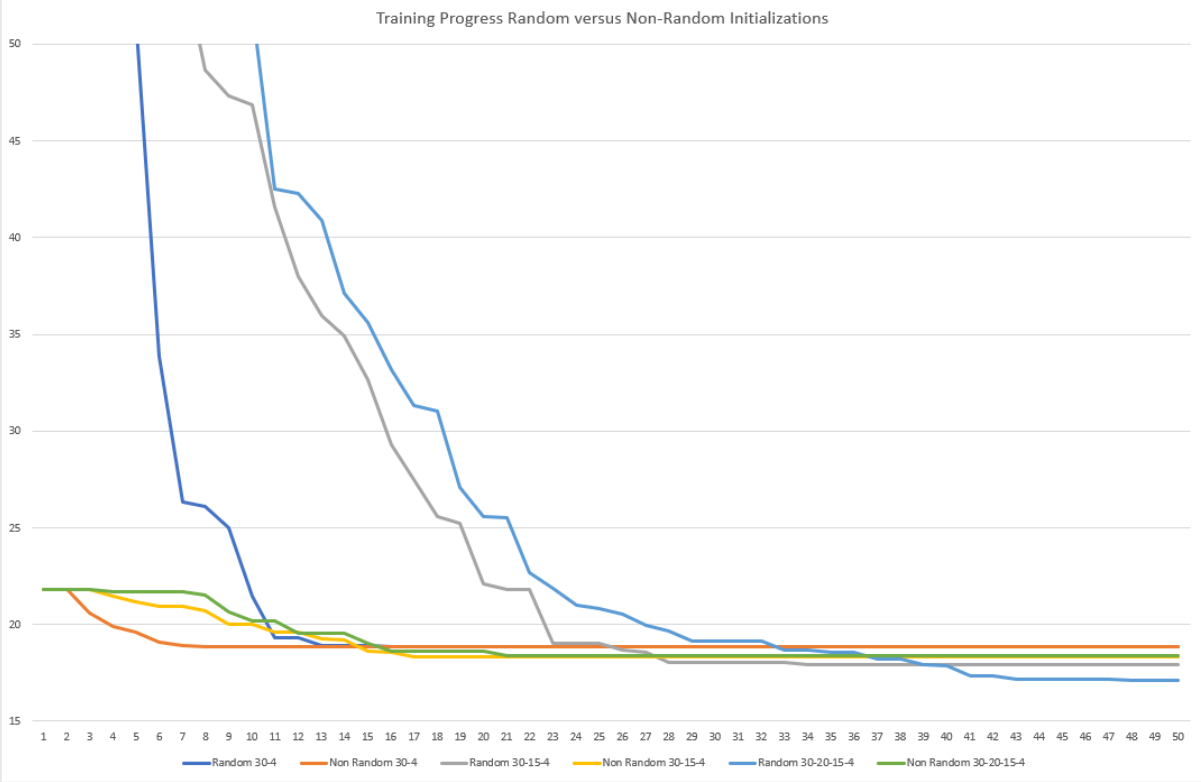
In this table it can be seen that, in general, normalizing the weights and biases during the creation of the next generation, helps in the learning process of the algorithm. For each case, the average error of networks obtained with normalizing is lower than for the non-normalized variant. The average decrease of the error is around 1.2 for all network types. Therefore, it is recommended to normalize the weights and biases, because it improves the learning behavior of the algorithm.

7.1.2 Dimensions

The topic of dimensions is not immediately clear. From the table it can be seen that in this case, no hidden layer outperforms having hidden layers. However, this does not show the full picture. With no hidden layer, the algorithm learns way faster, therefore the resulting error is lower. This is expected, because a lower number of dimensions also lowers the possible random variations. Also, the initial networks where, apart from the random ones, created in such a way, that there was no benefit from the extra hidden layers. Therefore, it is expected that with a longer time limit and using only random initialized networks, the networks with more hidden layers will outperform the networks with no hidden layers. An experiment to test this was

done in which all runs where done with 50 iterations. After each iterations, the best network is evaluated on the test data in order to have an fair comparison. Because of the computation time, the number of runs was limited from 10 to 5 and only the case of $D = 1.5$ is considered. In Figure 3 the results of this experiment are shown. A few points can be concluded from this figure. Firstly, the initial of the networks with problem specific weights outperform the random networks easily. This results in these networks filling the parent pool and even though the initials networks are added in each iteration, they perform way worse, therefore limiting the probability that these networks are selected as parents. For networks with no hidden layer however, this is not important, because it seems that the problem specific setup captures almost all relevant information. The algorithm does not find networks with a lower average error after iteration 7. With hidden layers, the algorithm finds better networks after this iteration. The main conclusion is, the more hidden layers, the slower the convergence, however, the better the quality of the final network. For the setup with two hidden layers, a random starting point will eventually outperform the problem specific initialization, because of the aforementioned reasons.

Figure 3: Iterations Random versus non-Random



7.1.3 Other results

From Table 8 more conclusions can be drawn. The choice of D, the factor that determines the number of allowed cities in a cluster before punishment, clearly changes the total error, however the remaining structure is still the same. Therefore it seems that the choice of D does not change the outcomes of this paper. However it is clear that more experiments should be done to investigate the impact and role of this variable. Another conclusion is that the networks outperform the standard clustering techniques. This in itself is not surprising, because one of the problem specific initializations is the k-means clustering. Therefore, networks should at least perform as good as this clustering technique. A separate experiment was done in which the same table was created, however now with 2-opt enabled, in order to see the effect of this. As can be seen, the general quality of the solutions has improved, which was expected. However, the networks performed comparatively worse than without 2-opt enabled. In most cases, the best tour found was exactly the same as the tour found by standard clustering techniques, at least for k-means. The Lin-Kernighan algorithm equalizes the playing field of both solution approaches. Therefore, in the remaining sections it is disabled. The last interesting point regarding the experiments in the standard setup is the role of the separate mutation strategies. In slightly over half of the runs, the best performing networks were created by first using a number of random mutation strategies and then performing organized cross over. Zero and non-zero entry mutation strategies where hardly ever used in the best performing networks, only twice out of 120 runs.

7.2 Other setups

In this section, results of other setups of the training algorithm are discussed. The within cluster TSP for instance is solved with CPLEX or combining the clusters is done by removing all within cluster unused edges and resolving the TSP from scratch. Lastly, during training the nearest neighbour algorithm is used to solve the within cluster TSP, however in test data the CPLEX solver is used. In general, the results are quite similar to the results already shown. The disadvantage of using the CPLEX solver during the training algorithm is that is will slow down the process quite a lot. After the time limit is passed, the algorithm will always finish the iteration, therefore this cannot be clearly seen in the data. However, in most cases the algorithm took over 1800 seconds to finish the fourth iteration. The resulting errors were a lot lower, less than half of the errors for the standard setup in the case of no hidden layers. Details of this can be found in Table 9.

Combining the clusters by removing unused edges and solving that TSP resulted in by far the best solution, although even slower than the previous method, especially in training. Results of this experiment can be found in Table 10. Using a faster solver, for instance Concorde by Applegate et al. (2006) could speed up this process, however it will remain a limitation. Another disadvantage is that saving for instance subtours is more difficult, if even possible, if of-the-shelf solvers are used. An interesting adaptation therefore is to use the nearest neighbour algorithm for within cluster tours during training and only in test use the CPLEX solver. This brings the advantages of the speed of the nearest neighbour algorithm, without the disadvantage of worse quality solutions for the test data. This is adapted in Table 11. The results are slightly worse than using CPLEX during training, however the training process is much faster. It is expected that the results are worse, because the network is trained for a slightly different purpose than it is used for in the end. Running this setup for the test data is faster (around 30%) then running the standard TSP, therefore it is a viable alternative.

8 Conclusion

The results of this paper can be summarized by three points. The first one is that neural networks can be used to solve the TSP together with conventional methods. In the literature, only examples in which the TSP was entirely solved by a neural network, for instance the Hopfield approach, or entirely by conventional methods. This paper shows that a combination of the methods is an alternative. The second point is that neural networks in combination with reinforcement learning are better equipped for this than standard clustering techniques, like k-means of hierarchical clustering. Neural networks can capture more information of the TSP instance than standard clustering techniques. Thirdly, using a trained neural network to determine clusters and then removing edges that are not used in the within cluster optimal tour, results in a TSP instance that can be solved quicker and without a large penalty in the tour length, around 2%.

This paper should mainly be seen as a start on the research of the integration of conventional linear program algorithms and heuristics and neural networks with reinforcement learning. Within the context of the Traveling Salesman Problem, many improvements can be made. Longer training times and more diverse network dimensions can yield better results. It could also be interesting to find initial settings that are more suited for neural networks with hidden layers. Another improvement can be to find a way to input the entire network at once, and

not per city. This could emphasize the relations between the cities more. The training can also be reduced by using faster TSP solvers, such as Concorde, if a way can be found to save the subtours that are found in the process. A last extension within the context of the TSP is to make the number of centers K flexible instead of fixing this based on the size of the TSP instance.

References

Abbeel, Pieter et al. (2007). "An application of reinforcement learning to aerobatic helicopter flight". In: *Advances in neural information processing systems* 19, p. 1.

Afolabi, Mark O and Olatoyosi Olude (2007). "Predicting stock prices using a hybrid Kohonen self organizing map (SOM)". In: *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. IEEE, pp. 48–48.

Agatonovic-Kustrin, S and R Beresford (2000). "Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research". In: *Journal of pharmaceutical and biomedical analysis* 22.5, pp. 717–727.

Ali, Shawkat and Kate A Smith (2006). "On learning algorithm selection for classification". In: *Applied Soft Computing* 6.2, pp. 119–138.

Applegate, David et al. (2006). *Concorde TSP solver*.

Budinich, Marco (1996). "A self-organizing neural network for the traveling salesman problem that is competitive with simulated annealing". In: *Neural Computation* 8.2, pp. 416–424.

Christofides, Nicos (1976). *Worst-case analysis of a new heuristic for the travelling salesman problem*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group.

Crites, Robert H, Andrew G Barto, et al. (1996). "Improving elevator performance using reinforcement learning". In: *Advances in neural information processing systems*, pp. 1017–1023.

Cunningham, Pádraig, Matthieu Cord, and Sarah Jane Delany (2008). "Supervised learning". In: *Machine learning techniques for multimedia*. Springer, pp. 21–49.

Daniel, Graupe (2013). *Principles of artificial neural networks*. Vol. 7. World Scientific.

Dantzig, George, Ray Fulkerson, and Selmer Johnson (1954). "Solution of a large-scale traveling-salesman problem". In: *Journal of the operations research society of America* 2.4, pp. 393–410.

Dorigo, Marco and Luca Maria Gambardella (1997). "Ant colonies for the travelling salesman problem". In: *biosystems* 43.2, pp. 73–81.

Du, K-L (2010). "Clustering: A neural network approach". In: *Neural networks* 23.1, pp. 89–107.

Fischer, Thomas et al. (2005). "An analysis of the hardness of TSP instances for two high performance algorithms". In: *Proceedings of the Sixth Metaheuristics International Conference*, pp. 361–367.

Ghahramani, Zoubin (2003). "Unsupervised learning". In: *Summer School on Machine Learning*. Springer, pp. 72–112.

Goldberg, Yoav (2017). "Neural network methods for natural language processing". In: *Synthesis lectures on human language technologies* 10.1, pp. 1–309.

Gutin, Gregory, Anders Yeo, and Alexey Zverovich (2002). "Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP". In: *Discrete Applied Mathematics* 117.1-3, pp. 81–86.

Hlaing, Zar Chi Su Su and May Aye Khine (2011). "Solving traveling salesman problem by using improved ant colony optimization algorithm". In: *International Journal of Information and Education Technology* 1.5, p. 404.

Hopfield, John J (1982). "Neural networks and physical systems with emergent collective computational abilities". In: *Proceedings of the national academy of sciences* 79.8, pp. 2554–2558.

Hopfield, John J and David W Tank (1985). ""Neural" computation of decisions in optimization problems". In: *Biological cybernetics* 52.3, pp. 141–152.

Johnson, Stephen C (1967). "Hierarchical clustering schemes". In: *Psychometrika* 32.3, pp. 241–254.

Jonsson, Anders (2019). "Deep reinforcement learning in medicine". In: *Kidney Diseases* 5.1, pp. 18–22.

Kamgar-Parsi, Behzad and Behrooz Kamgar-Parsi (1990). "On problem solving with Hopfield neural networks". In: *Biological Cybernetics* 62.5, pp. 415–423.

Kanungo, Tapas et al. (2002). "An efficient k-means clustering algorithm: Analysis and implementation". In: *IEEE transactions on pattern analysis and machine intelligence* 24.7, pp. 881–892.

Karapetyan, Daniel and Gregory Gutin (2011). "Lin–Kernighan heuristic adaptations for the generalized traveling salesman problem". In: *European Journal of Operational Research* 208.3, pp. 221–232.

Kizilateş, Gözde and Fidan Nuriyeva (2013). "On the nearest neighbor algorithms for the traveling salesman problem". In: *Advances in Computational Science, Engineering and Information Technology*. Springer, pp. 111–118.

Kohli, Sakshi, Surbhi Miglani, and Rahul Rapariya (2014). "Basics of artificial neural network". In: *International Journal of Computer Science and Mobile Computing* 3.9, pp. 745–751.

Lai, Matthew (2015). "Giraffe: Using deep reinforcement learning to play chess". In: *arXiv preprint arXiv:1509.01549*.

Leung, Kwong-Sak, Hui-Dong Jin, and Zong-Ben Xu (2004). "An expanding self-organizing neural network for the traveling salesman problem". In: *Neurocomputing* 62, pp. 267–292.

Lin, Shen and Brian W Kernighan (1973). "An effective heuristic algorithm for the traveling-salesman problem". In: *Operations research* 21.2, pp. 498–516.

Michiels, Wil, Emile Aarts, and Jan Korst (2007). *Theoretical aspects of local search*. Springer Science & Business Media.

Montana, David J and Lawrence Davis (1989). "Training Feedforward Neural Networks Using Genetic Algorithms." In: *IJCAI*. Vol. 89, pp. 762–767.

Nandy, Abhishek and Manisha Biswas (2018). "Neural Network Basics". In: *Neural Networks in Unity*. Springer, pp. 1–26.

Pasti, Rodrigo and L Nunes de Castro (2006). "A neuro-immune network for solving the traveling salesman problem". In: *The 2006 IEEE International Joint Conference on Neural Network Proceedings*. IEEE, pp. 3760–3766.

Phienthrakul, Tanasanee (2014). "Clustering evolutionary computation for solving travelling salesman problems". In: *International Journal of Advanced Computer Science and Information Technology (IJACSIT)* 3.3, pp. 243–262.

Prates, Marcelo et al. (2019). "Learning to solve NP-complete problems: A graph neural network for decision TSP". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33, pp. 4731–4738.

Raghavendra, U et al. (2018). "Deep convolution neural network for accurate diagnosis of glaucoma using digital fundus images". In: *Information Sciences* 441, pp. 41–49.

Reinelt, Gerhard (1991). "TSPLIB—A traveling salesman problem library". In: *ORSA journal on computing* 3.4, pp. 376–384.

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6, p. 386.

Rumelhart, DE, GE Hinton, and RJ Williams (1986). "Learning by error backpropagation". In: *parallel distributed processing*. Vol. 1. MIT press.

Silver, David et al. (2017). "Mastering chess and shogi by self-play with a general reinforcement learning algorithm". In: *arXiv preprint arXiv:1712.01815*.

Smith, Kate (1996). "An argument for abandoning the travelling salesman problem as a neural-network benchmark". In: *IEEE Transactions on Neural Networks* 7.6, pp. 1542–1544.

Smith, Kate A (1999). "Neural networks for combinatorial optimization: a review of more than a decade of research". In: *INFORMS Journal on Computing* 11.1, pp. 15–34.

Sutton, Richard S and Andrew G Barto (2018). *Reinforcement learning: An introduction*. MIT press.

Svozil, Daniel, Vladimir Kvasnicka, and Jiri Pospichal (1997). "Introduction to multi-layer feed-forward neural networks". In: *Chemometrics and intelligent laboratory systems* 39.1, pp. 43–62.

Takahashi, Yoshikane (1997). "Mathematical improvement of the Hopfield model for TSP feasible solutions by synapse dynamical systems". In: *Neurocomputing* 15.1, pp. 15–43.

— (1999). "A neural network theory for constrained optimization". In: *Neurocomputing* 24.1-3, pp. 117–161.

Talaván, Pedro M and Javier Yáñez (2002). "Parameter setting of the Hopfield network applied to TSP". In: *Neural Networks* 15.3, pp. 363–373.

Wang, Hongjian, Naiyu Zhang, and Jean-Charles Créput (2017). "A massively parallel neural network approach to large-scale Euclidean traveling salesman problems". In: *Neurocomputing* 240, pp. 137–151.

Wang, Jiahai and Yalan Zhou (2009). "Stochastic optimal competitive Hopfield network for partitional clustering". In: *Expert Systems with Applications* 36.2, pp. 2072–2080.

Wasserman, Philip D and Tom Schwartz (1988). "Neural networks. II. What are they and why is everybody so interested in them now?" In: *IEEE Expert* 3.1, pp. 10–15.

Xu, Xin and WT Tsai (1991). "Effective neural algorithms for the traveling salesman problem". In: *Neural Networks* 4.2, pp. 193–205.

Zhang, Wei and Thomas G Dietterich (1995). "A reinforcement learning approach to job-shop scheduling". In: *IJCAI*. Vol. 95. Citeseer, pp. 1114–1120.

Zhou, Yuren (2009). "Runtime analysis of an ant colony optimization algorithm for TSP instances". In: *IEEE Transactions on Evolutionary Computation* 13.5, pp. 1083–1092.

9 Appendix

This table contains the results of the experiments with the standard setup. That is, the nearest neighbour algorithm is used to solve the within cluster TSP and the networks are used to assign the cities to a cluster.

Table 8: Results Training Network

Network Setup	Not Normalized Error	Not Normalized Time	Normalized Error	Normalized Time	KMeans Error	KMeans A. Error	HC Error	TSP Time	TSP Time with Subtour Finder
30-4 d=1	30,19	1200	28,19	1200	37.47	34.80	81.64	4318	2549
30-4 d=1.5	20,04	1200	18,85	1200	22.61	21.83	54.19	4318	2549
30-15-4 d=1	30,41	1200	29,37	1200	37.47	34.80	81.64	4318	2549
30-15-4 d=1.5	21,31	1200	20,68	1200	22.61	21.83	54.19	4318	2549
30-20-15-4 d=1	34,23	1200	32,11	1200	37.47	34.80	81.64	4318	2549
30-20-15-4 d=1.5	21,76	1200	21,54	1200	22.61	21.83	54.19	4318	2549

In this table, results of the training runs for the travelings salesman problem are shown in which the within cluster TSP was solved by means of CPLEX and the formulation by Dantzig, Fulkerson, and S. Johnson (1954).

Table 9: Results Training Network with CPLEX solver

Network Setup	Normalized Error	Normalized Time	KMeans Error	KMeans A. Error	HC Error	TSP Time	TSP Time with Subtour Finder
30-4 d=1	11.46	1200	15.65	12.10	67.08	4318	2549
30-4 d=1.5	8.83	1200	11.37	9.07	41.26	4318	2549
30-15-4 d=1	12.10	1200	15.65	34.80	67.08	4318	2549
30-15-4 d=1.5	8.92	1200	11.37	9.07	41.26	4318	2549
30-20-15-4 d=1	12.10	1200	15.65	34.80	67.08	4318	2549
30-20-15-4 d=1.5	8.93	1200	11.37	9.07	41.26	4318	2549

This table contains the results of the runs in which the within cluster TSP was solved by means of CPLEX. In order to solve the original TSP instance, all unused within cluster edges are removed from the original TSP and this TSP is solved by CPLEX.

Table 10: Results Training Network with CPLEX solver, resolving TSP instance

Network Setup	Normalized Error	Normalized Time
30-4 d=1	2.29	1200
30-4 d=1.5	2.16	1200
30-15-4 d=1	2.29	1200
30-15-4 d=1.5	2.16	1200
30-20-15-4 d=1	2.29	1200
30-20-15-4 d=1.5	2.16	1200

This table contains the results of the runs in which the within cluster TSP was solved by the nearest neighbour heuristic during training and by CPLEX during the test phase. In order to solve the original TSP instance, all unused within cluster edges are removed from the original TSP and this TSP is solved by CPLEX.

Table 11: Results Training Network with nearest neighbour resolving TSP instance

Network Setup	Normalized Error	Normalized Time
30-4 d=1	3.47	1200
30-4 d=1.5	3.42	1200
30-15-4 d=1	3.98	1200
30-15-4 d=1.5	3.84	1200
30-20-15-4 d=1	3.98	1200
30-20-15-4 d=1.5	3.84	1200