Title	Reinforcement Learning Network Architecture To Solve Routing Problem With Time Window Constraint
Author(s)	朝木, 大輔
Citation	
Issue Date	2022-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/18050
Rights	
Description	Supervisor: Brain Michael Kurkoski, 先端科学技術研究科, 修士（情報科学）

Master's Thesis

Reinforcement Learning Network Architecture To Solve Routing Problem With Time Window Constraint

2030402 Asaki Daisuke

Supervisor Gregory Schwartzman

School of Information Science
Japan Advanced Institute of Science and Technology
(Master's degree)

August, 2022

Acknowledgements

This project would not have been possible without the support of many people. Many thanks to my main adviser, Prof. Schwartzman, who discussed my research with me and gave a lot of advise. He read through my thesis revisions and help me with enhancing the quality of the thesis. Also thanks to formal adviser, Prof. Kurkoski, who gave comments and advise on my thesis and facilitated the final defence. I also would like to thank to the commity members of my final defence, Prof. Nguyen, Prof. Inoue and Prof. Uehara, who gave a lot of questions and comments on my presentation.

I really appreciate to people around me who cheered me up to finish the master's degree. It wouldn't be mentally possible to finish it without their words.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The vehicle routing problem is a well-known and widely used problem in the field of operations research. In real-world business applications in the logistics industry, it is essential to find efficient routes for drivers because drivers' salaries are usually proportionate to the working hours of each driver, so if a driver delivers more parcels within the same time range, it reduces the total running cost for the delivery operation. Furthermore, it is predicted that the number of delivery drivers is not going to increase while the number of parcels increases rapidly in Japan. If a driver is able to deliver parcels more efficiently, it can resolve or prevent the problem of a lack of delivery drivers.

In the vehicle routing problem, vehicles must start from the *depot* (starting point) and return to the same depot. The goal of the vehicle routing problem is to find optimal routes for vehicles to deliver goods to a given set of customers, visiting each customer exactly once. If the number of vehicles is fixed to 1, the problem is called the traveling salesman problem (TSP). Real world applications demand additional constraints, such as a *demand capacity* constraint and a time window constraint. The capacitated vehicle routing problem (CVRP) is a VRP with a demand capacity constraint, in which vehicles have a *limited volume capacity* – they can deliver parcels such that the total volume of parcels does not exceed the vehicle's capacity. In the vehicle routing problem with a time window constraint (VRPTW), each vehicle has to visit customers within a specific time windows. The capacitated vehicle routing problem with time windows (CVRPTW) is a combination of CVRP and VRPTW, in which each vehicle has to satisfy both the capacity constraints and the time window constraints.

Although adding these constraints makes the problem more complicated, it is possible to find an approximate solution in a reasonable time by using *heuristic algorithms*. Heuristic algorithms, such as local search and guided local search, are widely used to find an approximate solution for efficient routing in real-world business settings. However, heuristic algorithms are not the ideal method because the running time required to find an optimal solution increases exponentially in the size of the input. In many real world applications, such as same-day delivery, a quick computation time is critical. Recently, there have been some attempts to apply deep learning techniques to VRP [2, 3, 4, 7, 8, 9, 10, 11, 12, 14] (detailed in the next section). The running time of deep learning models (the execution time of the trained model) which find an approximate solution is relatively fast compared to heuristic algorithms. Therefore, this approach has the potential to replace classic heuristic algorithms. However, the field of applying deep learning techniques to VRP is still in its infancy. Therefore, the goal of this study is to apply machine learning techniques to solve CVRPTW and to develop the use of deep leaning in the field of logistics.

1.2 Related work

There have been several results on the application of deep learning to VRP. Bello et al. [3] proposed using the Actor-Critic method to train a Pointer Network model [2]. Kool et al. [4] introduced the use of an attention network as a decision-maker (policy) of reinforcement learning and trained the network by the policy gradient method. The network of [4] is based on the Graph attention network [5]. These works focus on simplified problems, where instances have no time window constraints (CVRP). Zhang et al. [6] proposed to add a helper module on an attention network to learn time window constraint penalties on a traveling salesman problem instance.

There are some works on VRP with a time window constraint. Ke et al. [7] proposed to apply an attention mechanism to VRP with a soft time window constraint (VRPSTW). In their problem setting, the number of vehicles is fixed. The network successfully finds a solution for VRPSTW but the soft time window constraint is loose. It is allowed to violate the constraints if the number of vehicles is not enough to follow all the time window constraints. In our use case the number of vehicles is unbounded so it is enough to deliver all parcels while adhering to the constraints. Lin et al. [8] proposed an attention network that is able to handle the electric vehicle routing problem with time windows (EVRPTW). EVRPTW has another constraint that demands that the vehicle has to visit a charging station before it runs out of battery.

They consider time window constraints, capacity constraints and battery constraints, so their problem setting is more strict than ours. However, their model's running time (the time to execute the trained model and find a solution) is relatively long. Furthermore, the performance of their model is lacking. Although they conduct experiments on $N = 5$ to $N = 100$, they compare their results with a heuristic algorithm on instance of size $N = 5$ to $N = 30$, and it does not outperform. Joe et al. [9] studies reinforcement learning for dynamic VRP (DVRP) where the number of customers and their information is unknown when the vehicle starts. While these works show that deep reinforcement learning is a promising method for VRP with a time window constraint, our use case is more specific to CVRPTW.

For the same problem setting as our case, the following are the related works. Sultana, N.N. et al. [10] proposed a reinforcement learning architecture for both CVRPTW and the capacitated vehicle routing problem with time windows and dynamic routing (CVRP-TWDR). CVRP-TWDR is similar to CVRPTW but not all customer's information is known before the depot departure time, and new customers emerge during the vehicle travel. They use a simple fully connected neural network as a decision-maker and compare the performance with a genetic algorithm (GA). Although their problem setting is very similar to ours, they simplify it by fixing the number of vehicles. They successfully show their model outperforms the GA on CVRP-TWDR instances. However, on CVRPTW, their proposed architecture is worse than GA by more than 10% on N=25, 50, 100 instances (N is the number of nodes in the graph). Lei et al. [11] suggested a combination of reinforcement learning and a heuristic algorithm to solve CVRPTW. They show that their proposed model outperforms other heuristic algorithms. But they don't mention the running time of their algorithm, which is an important factor from a practical point of view.

Xin et al. [12] propose a combination of reinforcement learning and the Lin-Kernighan-Helsgaun algorithm [13] (a strong heuristic algorithm for the problem). Their problem setting is mainly TSP but they also extend the model for CVRPTW. The performance is good enough for practical applications, but the running time is long. It uses the deep neural network as an encoder and decoder to embed the node information and edge information, and create a solution candidate to narrow the search space. Then it uses the Lin-Kernighan-Helsgaun (LKH) algorithm to search for a solution. Since the LKH is an iterative heuristic algorithm, it takes a long time to find a solution. Jonas et al. [14] proposed an augmented attention based model. The original attention network on VRP [4] decides the next node to visit sequentially. The second route is created after the first one is created. When the second route is being created, nodes visited in the first route are not allowed to be visited. Thus, later routes are not optimized because the number of customers available is limited. Jonas et al. modifies the model so that it can create multiple routes in parallel, which means the first route, the second one and the next ones are created in turn. For instance, the second route's first customer is decided after the first route's first customer is decided, not after the first route is completed. Once a customer is visited it will be masked, so visiting one customer twice is not allowed. By doing this, each route has more options of a next node to visit, and the decision where to go next is made based on more information than the original one. They experiment on CVRPTW in various sizes ($N = 20, 50, 100$) and show that their model outperforms ORtools [16] (ORtools is widely used in the logistic field.).

1.3 Research objective

As mentioned in the previous section, there have been several works that propose an application of a neural network and reinforcement learning for the vehicle routing problem. Since this field is still in its infancy, the problem settings are too simplistic to be useful for real-world problems. Specifically, assuming a Euclidean distance between nodes, means existing works are not able to consider winding roads or traffic conditions. Therefore, we focus on the following two objectives. Some related works tackle CVRPTW but use a combination of heuristic algorithms and deep learning. Although it is a promising architecture, it has a slow run time, and the architecture itself is often complicated.

1. To extend the attention network for VRP with a time window constraint. Since it is an end-to-end model, the architecture itself is simpler, and the run time can be shorter than heuristic hybrid models.

2. To extend the attention architecture so that it can take distance matrix as input. The information in the distance matrix is arbitrary, so it can contain both traffic conditions and indirect distance.

1.4 Significance

The vehicle routing problem is widely used in the logistics industry, since it can help operators and drivers to find efficient routes for package delivery. There are several open-source software packages that use the heuristic method to solve the vehicle routing problem. These meta-heuristic methods make decisions by following some rules, which might be able to be parameterized by neural networks if these rules are seen as policies. That is, neural networks have the potential to replace existing meta-heuristic methods and it might be possible to make the logistic industry more efficient. Therefore, this work will contribute to the development of reinforcement learning for solving routing problems and, as a result, will develop the logistics industry.

Chapter 2

Problem setting and network architecture

2.1 Problem setting

Given an positively edge-weighted fully connected graph $G = (\mathbf{V}, \mathbf{E}), \mathbf{V} = \{d = v_0, v_1, ., v_i, ., v_N\}$ where N is the number of customers and d is a special depot node. Let w_e denote the weight of e. Note that the edge weights imply a symmetric distance matrix (where all entries are finite as the graph is fully connected).

$$\mathbf{D} = \begin{bmatrix} w_{(d,d)} & w_{(d,v_1)} & \cdots & w_{(d,v_N)} \\ w_{(v_1,d)} & w_{(v_1,v_1)} & \cdots & w_{(v_1,v_N)} \\ \cdots & \cdots & \cdots & \cdots \\ w_{(v_N,d)} & w_{(v_N,v_1)} & \cdots & w_{(v_N,v_N)} \end{bmatrix}$$

We often consider the case that the vertices are points in $\mathbb{R}^2$, in which case the distances are *Euclidean*.

Each v_i has the following attributes: package demand $q_i \in \mathbb{R}^+$, and a time window $[s_i, f_i] \subset \mathbb{R}^+$. Given G and a fleet of identical vehicles uniformly capacitated by $Q \in \mathbb{R}^+$, our goal is to assign delivery routes to vehicles (paths in the graph) such that each customer must be assigned to a *single* vehicle exactly once and all routes start and end in the depot. In our setting the size of the fleet is unbounded and it is a parameter which is part of the solution outputted by the algorithm (together with the routes for each vehicle). Let us denote the number of vehicles by $k \in \mathbb{N}$. Note that this does not fix the number in advance and we only use it for notation (e.g., to define our cost function). A route for each vehicle is formally defined as $r_i = (d = v_{i_1}, v_{i_2}, \dots, v_{i_m} = d)$ such that $v_{i'_j} \neq v_{i_j}$ if $j \neq j'$. Furthermore, $r_i \cap r_{i'} = \{d\}$ if $i \neq i'$ and $\cup_{i=1}^{k} r_i = \mathbf{V}$ (we slightly abuse notation and treat

r_i as a set when it is convenient).

Each customer has their own demand that a vehicle must deliver. The demand can be thought of as a volume or a weight of a package. Every vehicle has a capacity Q that is the load limitation for parcels to be delivered. So as the capacity constraint, the total customer demands of parcels that a vehicle delivers must be less than Q. Formally:

$$\sum_{v \in r_i} q_v \leq Q$$

For the time window constraint, the arrival time at each customer must be before the end of the time window. Arriving earlier than the start time would be fine, but the vehicle has to wait until the start time. For the j-th vehicle with route $r_j = (d = v_{i_1}, v_{i_2}, \ldots, v_{i_m} = d)$, the arrival time at the k-th customer is denoted as a^j_k. The arrival time is formally defined as

$$a^j_k = max(a^j_{k-1} + w_{(v_{i_{k-1}}, v_{i_k})} + c, s_{v_k}) \tag{2.1}$$

Where $a^j_0 = 0$ and c is the *service time* – the time it takes to deliver the package to the customer. We take the service time to be a constant (e.g., 10). We define $a^j = a^j_m - c$ (where m is the length of the route) as the *working time* of route j (we deduct c as there is no service time in the depot). We enforce the following constraints:

$$\forall t \in [N], f_t \geq a^k_j, v_t = v_{i_k} \in r_j$$

This constraint is called a *hard* time window constraint – it does not allow a vehicle to arrive later than the end time for each customer. A soft time window constraint allows arrival after the end time. We consider the hard time window constraint because it is more applicable to real-world business setting. The objective value which we aim to minimize is the total working hours of all drivers.

$$cost = \sum_{j=1}^{k} a^j \tag{2.2}$$

2.2 Network overview

As mentioned in the previous chapter, there have been some works that try to apply deep learning models to TSP and VRP. We focus on the attention neural network architecture because it is an end-to-end model with a fast computation time (compared to the heuristic method).

We implement two models which extend the architecture of the attention network [15] such that it satisfies our two objectives:

- (**Version 1**) An extension of the attention network such that for every node, v_i, it is able to encode a time window, $[s_i, f_i]$, a demand, q_i, and the node's XY coordinates $\mathbf{X}_i \in \mathbb{R}^2$.

- (**Version 2**) An extension of the attention network that encodes a time window $[s_i, f_i]$, a demand q_i and is able to take a distance matrix $\mathbf{D}$ as input instead of the node's XY coordinates.

The attention network architecture consists of an encoder and a decoder.

Encoder The encoder is a stack of layers whose goal is to encode the instance information, such as the constraints, the distance matrix (for Version 2), and the customer's Euclidean coordinates (for Version 1) into an *embedded vector*. That is, the instance information is packed into a vector space, called the embedded vector space. We denote the dimension of this space by dim_{emb} and for our architecture we set $\text{dim}_{emb} = 128$.

Decoder The decoder is a stack of layers whose goal is to decode the embedded vector into *meaningful values*. In our case it assigns values (we call it the *score*) to nodes, such that a greedy route construction algorithm can use these values to construct efficient vehicle routes for the problem. Once scores are assigned to nodes, the greedy algorithm picks the node with the highest score as a next node to visit.

The route construction mechanism is explained below.

1. Routes are created sequentially, that is, the route starts from the depot and picks the next nodes to visit one by one. While picking nodes, the decoder checks if unvisited nodes can be visited without violating constraints. When there is no node available, the vehicle has to return to the depot. If there are nodes which are not visited yet, a new vehicle starts traveling. Since we assume the number of vehicles is not limited, a new vehicle is generated until all nodes have been visited exactly once.

2. The route construction routine is as follows. First the decoder calculates the scores of all nodes. Second, the greedy route construction algorithm picks the node with the highest score as a next node. For instance, the decoding phase starts with the vehicle at the depot as a starting point. The decoder calculates the scores for all nodes, and the greedy algorithm picks the node with the highest score as the node to visit first. Then, the decoder calculates scores for all nodes while it puts $-\infty$ on first visited node's score. The greedy algorithm picks the next

node again by picking the node with the highest score. A construction of one route continues like this until there is no node available due to constraints or because all nodes are already visited.

3. To calculate scores, the decoder uses the output of the encoder which is the encoded graph vector and the current state. The current state is a vector which has information of the current (partial) solution, which changes during the decoding phase. The current state includes all routes constructed until now, the current (partial) route being constructed and the elapsed time. This information is critical for the decoder to be able to compute a meaningful score. The definition of the current state and how the decoder creates scores is detailed in the section 2.5.

We prepare two datasets, the training dataset and the test dataset based on the same distribution. Instances of CVRPTW are generated based on a given distribution which is explained in Chapter 3. The models are trained on the training dataset. The objective of the model is to create routes $\{r_i\}$ such that the cost (Equation 2.2) of solution is to be minimized while all vehicles follow the time window constraints and the capacity constraints.

Figure 2.1 and Figure 2.2 show an overview of the network architecture of Version 1 and Version 2. The difference between Version 1 and Version 2 is the way the encoder receives the distances between nodes. In Version 1 (Figure 2.1), the encoder takes in each node's XY coordinates as one of the encoded targets. On the other hand, the encoder in Version 2 doesn't take the node's XY coordinates but takes the distance matrix (Figure 2.2).

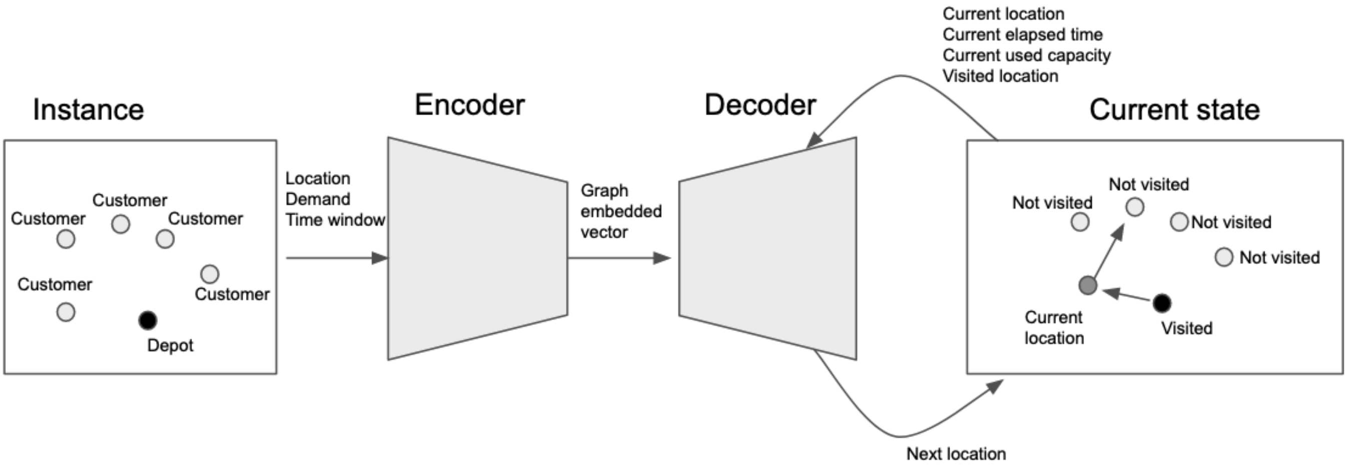


Figure 2.1: Architecture overview of Version 1.

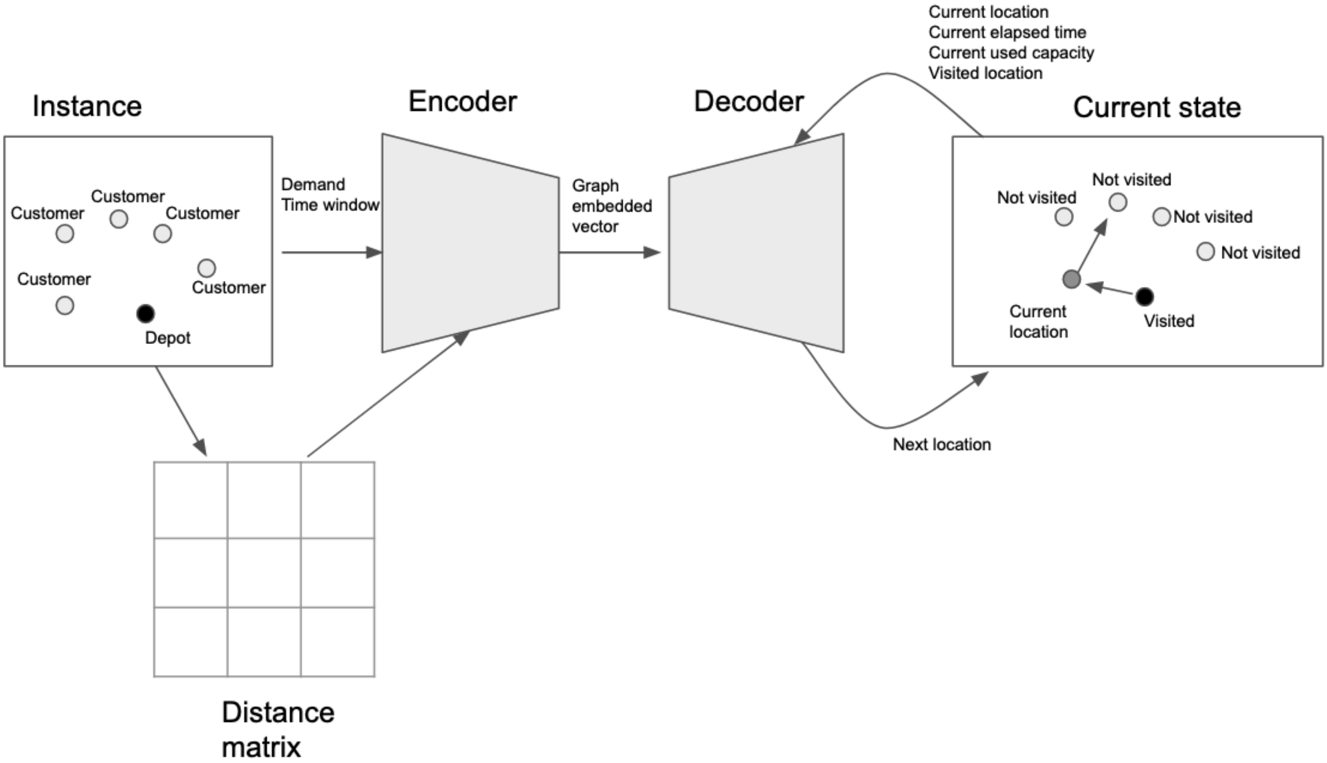


Figure 2.2: Architecture overview of Version 2.

2.3 Basic encoder architecture

We describe the general architecture for both Version 1 and Version 2, the only difference is the input of the encoder (described in this section for Version 1, and in the next section for Version 2). An overview of the encoder architecture is shown in Figure 2.3. The encoder is composed of a stack of 2 layers. Before the 2 layers, there is a simple linear projection. The two layers include a Multi-Head Attention sublayer and a simple feed-forward sublayer (we call a small component in a layer a sublayer. A layer consists of a batch normalization sublayer, a Multi-Head Attention sublayer and a feed-forward sublayer). After a Multi-Head Attention sublayer and a feed-forward sublayer a batch normalization sublayer is applied. These 2 sublayers are explained in this section. We denote an input vector of each layer by $\mathbf{z}_i^l$ where $l \in \{1, 2\}$ is the layer number and $i \in \{0, 1, \ldots, N\}$ is the index of graph nodes in an input instance (i.e, the i-th node in an instance).

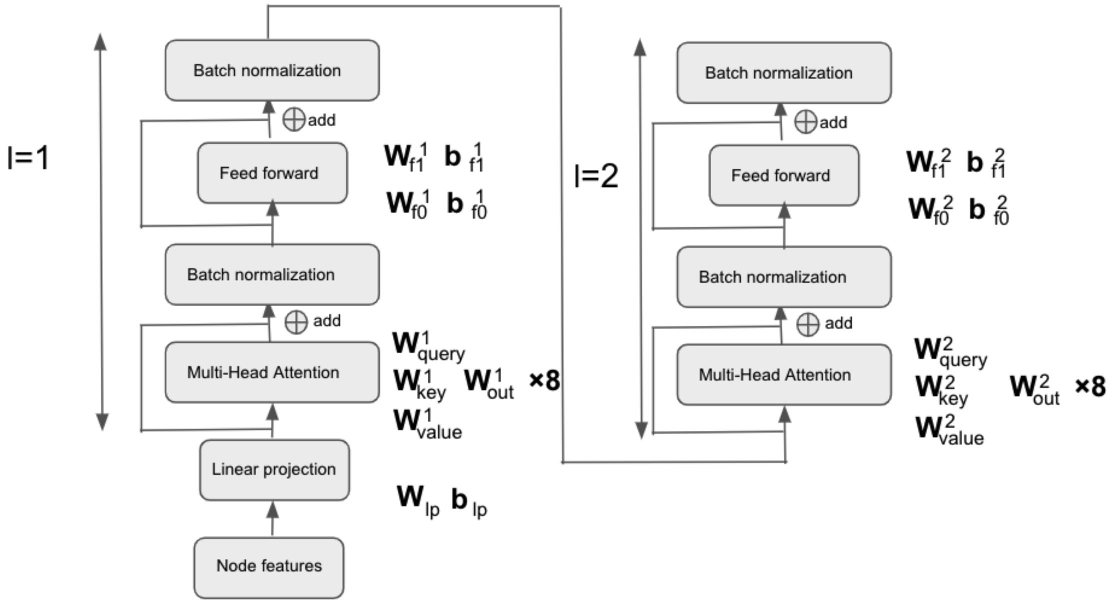


Figure 2.3: The encoder overview.
Next to every sublayer we write the notation for the weights of the layer.

The goal of the first linear projection is to project the input into a 128 dimensional vector (we follow the previous work [4] to choose 128 as the size of dimension.) by multiplying it with $\mathbf{W}_{lp}$. The weights $\mathbf{W}_{lp}$ can be learned during the network training. $\mathbf{W}_{lp}$ is of size 5×128 for Version 1 (the input dimension is 5 and output dimension is 128). The input of the encoder is $\mathbf{V} = (\mathbf{d} = \mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_N)$ where $\mathbf{d}$ denotes the depot (the depot is always the first entry, therefore the network always knows where the depot is). As mentioned in the previous section, each node $\mathbf{v}$ has several attributes. $\mathbf{z}_i^{\mathit{1}}$ denotes the i-th node's feature vector after the first linear projection. Version 1 takes 5 features, XY coordinate, demand, time window. Therefore $\mathbf{v}_n$ can be described as:

$$\mathbf{v}_i = [\mathbf{x}_i, q_i, s_i, f_i] \tag{2.3}$$
$$\mathbf{x}_i = (x_i, y_i)$$

where [,] is a horizontal concatenation. So for Version 1 the dimension of the input for each node is five.

For the linear projection we denote an intercept as $\mathbf{b}_{lp}$ with 128 dimensions. The output of the first linear projection is denoted as $\mathbf{Z}^1$ (setting $l = 1$). The output of the first projection is the input of the first layer of the encoder.

$$\mathbf{Z}^{1} = \mathbf{V}\mathbf{W}_{lp} + \mathbf{b}_{lp}$$

$$\mathbf{Z}^{1} = (\mathbf{z}_0^{1}, \mathbf{z}_1^{1}, \ldots, \mathbf{z}_N^{1})$$

$\mathbf{Z}^{1}$ is passed to the first layer. The first layer and the second layer have a multi-head attention sublayer. The attention sublayer has three parameters, $\mathbf{U}^l$, $\mathbf{E}^l$ and $\mathbf{A}^l$. These parameters are given by a linear projection of the same input, $\mathbf{z}^l$, with independent learnable weights $\mathbf{W}^l_{\text{query}}$, $\mathbf{W}^l_{\text{key}}$, $\mathbf{W}^l_{\text{value}}$ of size 128×128. The *query*, *key* and *value* can be seen as a mapping of key-value pairs and a query. The multiplication of a *query* of node A and a *key* of node B calculates the weight between node A and node B. The weight means how much node A should pay attention to node B. Finally, the *value* of B (the *value* can be seen as node B's feature vector) is added to an output vector corresponding to node A after the *value* is multiplied by the weight. When the weight is large (which means that node A's *query* and the node B's *key* is similar), node B's *value* has a strong effect on the output of node A, and vice versa. This mechanism is shown in the equation 2.7.

The parameters $\mathbf{U}^l$, $\mathbf{E}^l, \mathbf{A}^l$ are size $(N+1) \times 128$ matrices for one head in a simple attention sublayer. However, in our model, we use a multi-head attention sublayer, so these parameters are divided into eight matrices (we denote the number of heads as nhead $= 8$). Thus, a $(N+1) \times 128$ size matrix become eight matrices of size $(N+1) \times 16$. Each of these eight matrices is called a head in the context of multi-head attention sublayer.

$$\mathbf{U}^l = \mathbf{z}^l \mathbf{W}^l_{\text{query}} \tag{2.4}$$

$$\mathbf{E}^l = \mathbf{z}^l \mathbf{W}^l_{\text{key}} \tag{2.5}$$

$$\mathbf{A}^l = \mathbf{z}^l \mathbf{W}^l_{\text{value}} \tag{2.6}$$

We denote by $\text{dim}_{\mathbf{z}}$ the dimension of the input $\mathbf{z}^l$ which is 128. The superscript T over a vector denotes its transpose. The attention mechanism is as follows.

$$\text{Attention}(\mathbf{U}^l, \mathbf{E}^l, \mathbf{A}^l) = \text{softmax}(\frac{\mathbf{U}^l(\mathbf{E}^l)^T}{\sqrt{d_k}})\mathbf{A}^l \tag{2.7}$$

$$d_k = \frac{\text{dim}_{\mathbf{z}}}{\text{nhead}}$$

The multiplication of $\mathbf{U}$ and $\mathbf{E}$ is to compute *compatibility* of queries with all nodes features. We denote it as $\mathbf{c}$, defined below.

$$\mathbf{c} = \mathbf{U}\mathbf{E}^T \tag{2.8}$$

The compatibility expresses how much one node should pay attention to other nodes' vectors. $\sqrt{d_k}$ is a scaling factor. Finally, the softmax function is applied.

In our attention based model, a multi-head attention sublayer with 8 heads is used. The multi-head attention sublayer is denoted as MHA. We denote by nh a head number. $\mathbf{W}^l_{\text{nh,out}}$ is a learnable weights matrix of size 16×128 (as the dimension in one head is 16). $\mathbf{G}^l_{nh}$ is the output of the attention function of size $(N+1) \times 16$. $\mathbf{U}^l_{\text{nh}}$, $\mathbf{E}^l_{\text{nh}}$ and $\mathbf{A}^l_{\text{nh}}$ are also of size $(N+1) \times 16$. MHA is defined as follows:

$$\text{MHA}(\mathbf{Z}^l) = \sum_{nh=1}^{\text{nhead}} \mathbf{G}^l_{nh} \mathbf{W}^l_{\text{nh,out}} \tag{2.9}$$

$$\mathbf{G}^l_{nh} = \text{Attention}(\mathbf{U}^l_{\text{nh}}, \mathbf{E}^l_{\text{nh}}, \mathbf{A}^l_{\text{nh}}) \tag{2.10}$$

There is a connection that connects the input and output of MHA directly (a *skip connection*) in the attention sublayer. It is followed by a batch normalization sublayer where input data is normalized. We denote the batch normalization sublayer as BN. $l \in 1, 2$ is the number of layers in the encoder as mentioned above.):

$$\hat{\mathbf{Z}}^l = \text{BN}(\text{MHA}(\mathbf{Z}^l) + \mathbf{Z}^l)$$

$\hat{\mathbf{z}}^l_i$ denotes the output of the i-th node after the batch normalization sublayer in the l-th layer. This output is passed to a feed-forward sublayer **FF** with a ReLU activation function. This feed-forward sublayer consists of 1 hidden layer. We denote by $\mathbf{W}^l_{f0}$, $\mathbf{W}^l_{f1}$ the weights (of size 128x128) and by $\mathbf{b}^l_{f0}, \mathbf{b}^l_{f1}$ the intercepts (size 128). The feed-forward sublayer is described as follows:

$$\text{FF}(\hat{\mathbf{Z}}^l) = \text{ReLU}(\hat{\mathbf{Z}}^l \mathbf{W}^l_{f0} + \mathbf{b}^l_{f0}) \mathbf{W}^l_{f1} \cdot + \mathbf{b}^l_{f1}$$

Finally, we apply batch normalization:

$$\mathbf{Z}^{l+1} = \text{BN}(\text{FF}(\hat{\mathbf{Z}}^l) + \hat{\mathbf{Z}}^l) \tag{2.11}$$

2.4 Distance matrix extension (Version 2)

We extend the network so that it can take a distance matrix directly as input. We call this extension Version 2. In previous works, networks take location information via XY coordinates, and assume that the distance between customers is the Euclidean distance. It is fine to assume Euclidean distance to demonstrate the theoretical effectiveness of the network, however, it is essential to allow the network to take in a distance matrix as input for real-world business operations. This is because winding roads and traffic conditions may drastically affect the distance. To achieve the above objective, we implement a direct connection from the distance matrix $\mathbf{D}$ to the compatibility $\mathbf{c}$ from the equation 2.8.

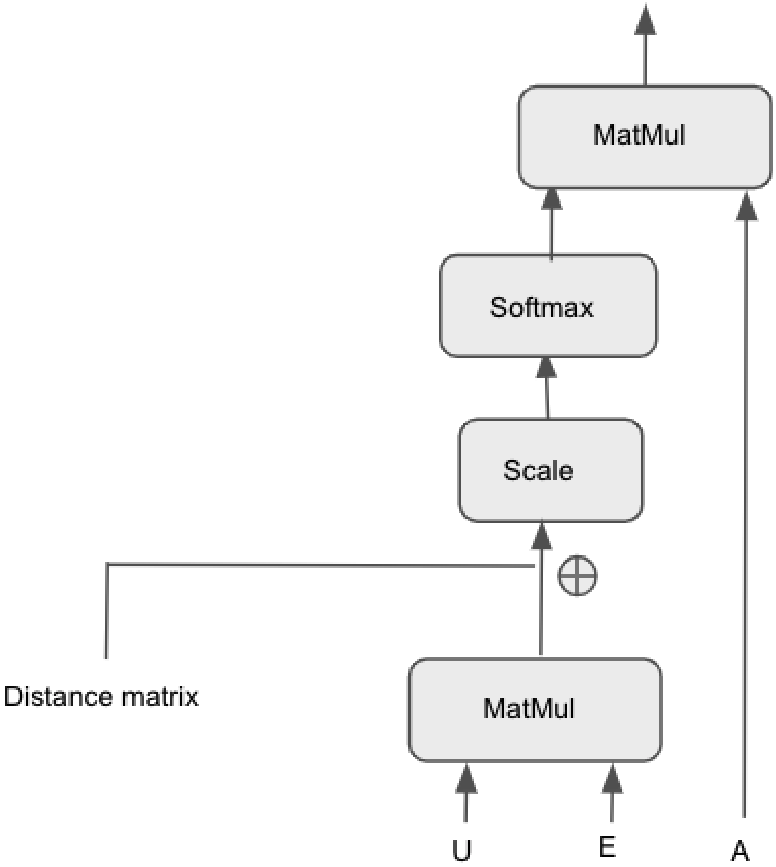


Figure 2.4: Attention layer with a distance matrix connection

As mentioned above, we use an attention layer with multiple heads. We denote the m-th head's compatibility as $\mathbf{c}_m$. The size of $\mathbf{c}_m$ is $(N+1) \times (N+1)$. We define:

$$\hat{\mathbf{c}}_m = \text{Norm}(\mathbf{c}_m) - \mathbf{D} \tag{2.12}$$

The **Norm** is defined as follows (where c_{ij} is an element of $\mathbf{c}_m$):

$$c_{\max} = \max(|c_{i,j}|), i, j \in \{0, 1, \ldots, N\}$$

$$\forall i, j, c_{i,j} = \frac{c_{i,j}}{c_{\max}}$$

The architecture is shown in figure 2.4. Version 2 doesn't encode XY coordinates at the encoding phase, instead it adds the distance matrix $\mathbf{D}$ to compatibility.

The compatibility expresses the weight of all combinations of pairs of nodes, which is used in the multiplication by $\mathbf{A}$ (in the equation 2.6) in the subsequent process. It means the compatibility value can be regarded as a weight signifying how much should a node pay attention to every other node. Since Version 2 encodes node parameters (q_n, s_n and f_n), $\mathbf{c}_m$ will have a compatibility value based on these parameters. That is, the original $\mathbf{c}_m$ has information of the capacity constraints and the time window constraints. By subtracting the normalized distance matrix value, the value of the compatibility is reduced proportionally to the distance between nodes. This means that far nodes pay less attention to far away nodes. Therefore, equation 2.12 merges the encoded information with the distance information.

The input of the first linear projection of Version 2 is different from the input of Version 1. More specifically, the node location is not passed as input:

$$\mathbf{v}_n = [q_n, s_n, f_n]$$

Compared to equation 2.3, the dimension of input $\mathbf{v}_n$ is reduced from 5 to 3.

2.5 Decoder

The goal of the decoder is to construct efficient routes for a fleet of vehicles. In section 2.2 we presented an overview of the decoder, and in this section we presented a more detailed description.

We start by describing the construction of a single vehicle route. We introduce a new variable p, which denotes the step in the current route being constructed. It indicates how many nodes the vehicle visited already during the decoding phase. When the vehicle is at the starting point (the depot), $p = 0$. For every node visited, p is incremented.

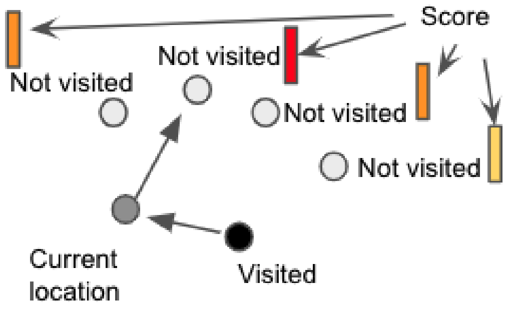


Figure 2.5: Decoder mechanism
The black node is already visited. The dark gray node is the last node the decoder chose to visit. The light gray nodes are not visited yet. From the current location (the dark gray node), the decoder chooses which node to visit next based on the score of each node. The bars shown near each node show the score which the decoder calculates. In this figure, red is the highest score, so the decoder chooses that node to visit next.

The architecture of the decoder is just one attention layer. Before data is passed to the attention layer, a linear projection is applied to the input vector. The embedded vector $\mathbf{Z}^e$ is passed to the decoder from the encoder. $\mathbf{Z}^e$ is the output of the encoder (and the input of the decoder), where e denotes the decoder. Initially, $\mathbf{Z}^e$ is projected to the same dimension by a linear projection. $\mathbf{W}_{\text{in}}$ are learnable weights for the initial projection of size 128×128.

$$\hat{\mathbf{Z}}^e = \mathbf{Z}^e \mathbf{W}_{\text{in}}$$

After the linear projection, the projected matrix $\hat{\mathbf{Z}}^e$ is also projected into two matrices $\mathbf{E}^e$ and $\mathbf{A}^e$ respectively by independent learnable weights $\mathbf{W}^e_{\text{key}}$, $\mathbf{W}^e_{\text{value}}$ as in equations 2.5, 2.6.

$$\mathbf{E}^e = \hat{\mathbf{Z}}^e \mathbf{W}^e_{\text{key}} \tag{2.13}$$

$$\mathbf{A}^e = \hat{\mathbf{Z}}^e \mathbf{W}^e_{\text{value}} \tag{2.14}$$

These matrices are used as fixed graph embedding, that is, these matrices are not changed during the decoding phase. It is necessary for the network to know the *current state*. The current state C_p contains three variables (the subscript p is the step as mentioned in the beginning of this section). The first variable is the current node embedded vector $\mathbf{P}_p$. This is a vector in the output of the encoder (from the equation 2.11) corresponding to the current visited node. The second one is the current available capacity R_p which is defined as follows (q_v is the demand of node v. r_i is the i-th node in the route currently being constructed).

$$R_p = Q - \sum_{i=1}^{p} q_{r_i} \tag{2.15}$$

The third one is the elapsed time t which is the same as arrival time defined in equation 2.1. The current state is defined as follows.

$$\mathbf{C}_p = [\mathbf{P}_p, R_p, t]$$

The input of the attention layer is prepared with another linear projection as follows ($\mathbf{W}^e_{\text{query}}$ is a learnable weight matrix of size 130 × 128 and [,] is a horizontal concatenation):

$$\mathbf{U}^e_p = [\hat{\mathbf{Z}}^e, \mathbf{C}_p]\mathbf{W}^e_{\text{query}}$$

The attention mechanism is the same as for the encoder (see equation 2.7). $\mathbf{U}^e_p$, $\mathbf{E}^e$ and $\mathbf{A}^e$ are used as input for the decoder's attention layer. The attention layer in the decoder uses multi-head attention, MHA, and the output of the layer is projected by a linear projection with learnable weights $\mathbf{W}_o$. $\mathbf{O}_p$ is a N+1 dimensional vector representing the node scores.

$$\mathbf{O}_p = \mathbf{W}_o \text{MHA}(\hat{\mathbf{Z}}^e)$$

The output score is clipped by tanh. After the clipping, the output is filtered by a mask to prevent visiting nodes unavailable due to constraints. The mask is described in the next section. C is a constant value. We set it to 20.

$$\hat{\mathbf{O}}_p = C \cdot \tanh(\mathbf{O}_p) \tag{2.16}$$

$$\hat{\mathbf{O}}^m_p = \text{mask}(\hat{\mathbf{O}}_p) \tag{2.17}$$

Finally, the next node to visit, g_p, is decided by picking the max value of $\hat{\mathbf{O}}_p$.(i means i-th node)

$$g_p = \arg\max_i(\hat{\mathbf{O}^m_p})$$

2.6 Masking

To prevent visiting a node which violates the constraints, a masking technique is used in our decoder. The current available capacity R_p (defined in equation 2.15) and all unvisited nodes' demand (q_i for i-th node) is checked to determine which node cannot be visited due to capacity constraints. If the node cannot be visited next, the corresponding score for the node in $\hat{\mathbf{O}}$ (equation 2.17) is replaced with $-\infty$. m_i is a Boolean value indicating whether the i-th node is not available (it is true if the node is not available).

$$\text{mask}(\hat{O}_i) = \begin{cases} \hat{O}_i, & \text{if } m_i \text{ is false.} \\ -\infty, & \text{otherwise} \end{cases}$$

After visiting a node, R_p is reduced by q_i given the next node is the i-th node. For instance, if the given vehicle capacity is $Q = R_0 = 10$ and the first customer's demand is $q_i = 3$, R_1 becomes 7 (Figure 2.6).

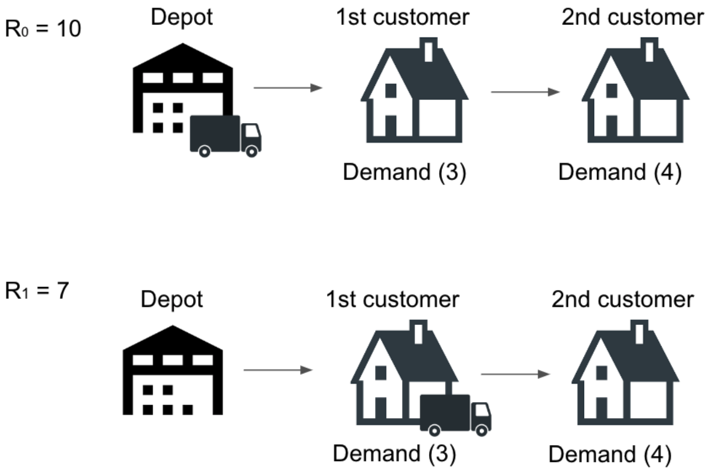


Figure 2.6: Capacity is used as the vehicle is traveling.

If there is no node that satisfies the capacity constraint, the vehicle must return to the depot. Figure 2.7 shows such a scenario. In the figure, the current available capacity is R_2=3 and the other nodes' demand is more than 3, which means that the vehicle is not able to visit any node. So it has to return to the depot.

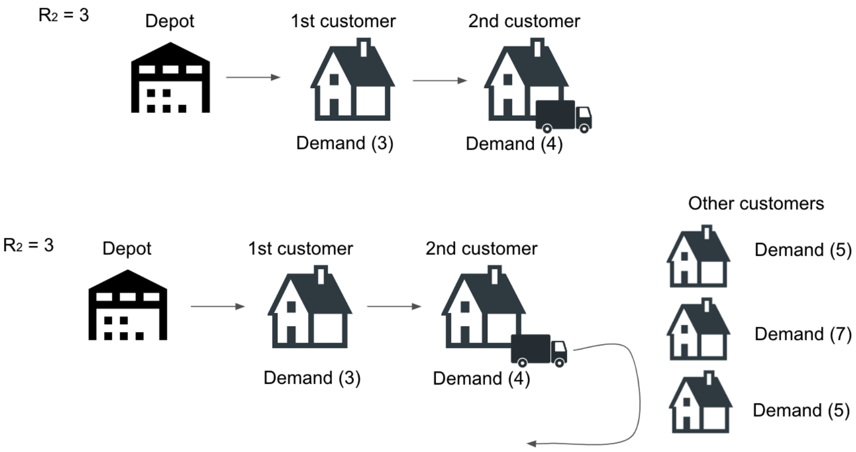


Figure 2.7: Return to the depot due to a capacity constraint.

As mentioned before, we enforce a hard time window constraint, thus, it is not allowed for a vehicle to arrive after the end time of the time window. However, it is allowed to arrive before the start of the time window. In this case, the waiting time at the node is included in the elapsed time t.

The time elapsing is shown in Figure 2.8. Suppose that the start time is 10:00 and the distance between the depot and the first customer is 30 minutes, then the elapsed time t is 30 minutes and the time is 10:30 when it reaches the first customer.

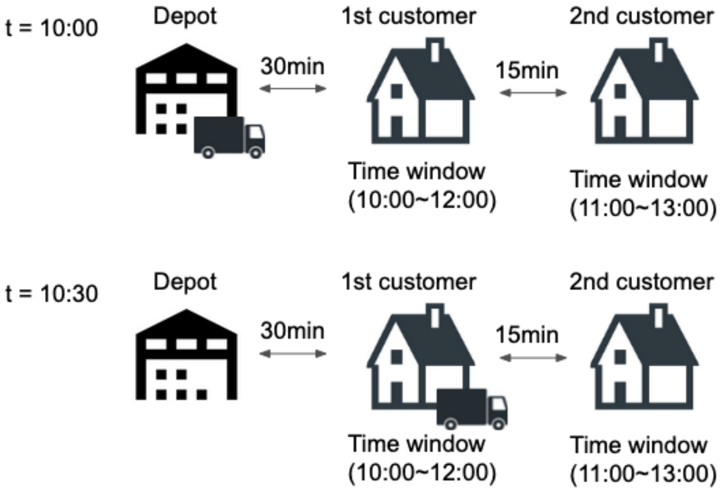


Figure 2.8: the time elapsing

If there is no node that satisfies the time window constraint, the vehicle must return to the depot. Figure 2.9 shows two scenarios.

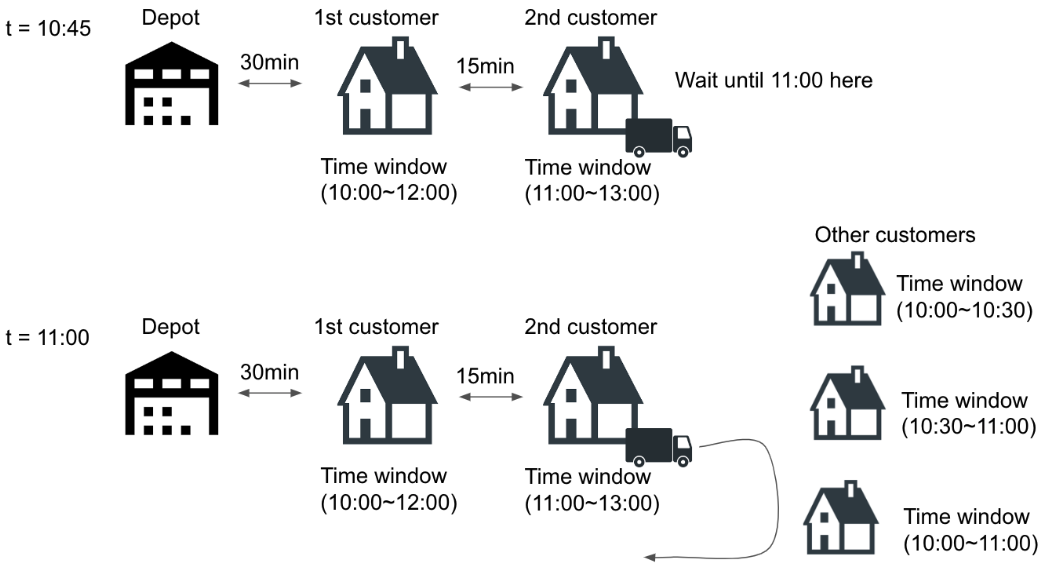


Figure 2.9: return to the depot (time window constraint)

In the first scenario the vehicle arrives earlier than the start time s_i and has to wait until s_i. Suppose the current time is t=10:30 at the first customer. Then, the vehicle visits the second customer which is 15 minutes away from the first one. It arrives at t=10:45 but $s_2 = 11 : 00$ so it has to wait until t=11:00. In the second scenario the vehicle must return to the depot because there is no customer that can be reached before the end time of the time window. The current time t=11:00 is later than other customers' end time. In this situation, the vehicle must return to the depot to finish working. As mentioned in the previous section, after one vehicle returns to the depot and there are still unvisited nodes, a new vehicle starts to travel from the depot to make deliveries for rest nodes.

So far we have described the capacity constraint and the time window constraint *individually*, however, CVRPTW must maintain these constraints *simultaneously*. Therefore, the mask is constructed with a logical OR. Let a_i and b_i denote the mask boolean of the capacity constraint and the time window constraint respectively. When either a_i or b_i is true, then the mask is true, which means that the node is not available.

Chapter 3

Experimental setup

We conduct experiments to show the effectiveness of our attention network on CVRPTW. We use Google-ORtools [16] as a benchmark to evaluate the performance (quality of the solution) of our models. The performance of Version 1 and Version 2 is compared with ORtools. Regarding the comparison of the performance of our models, our main goal is not to outperform ORtools. Even if the objective value is not much better than ORtools, it is desirable to shorten the computation time while producing similar quality solutions. The computing hardware environment is as follows. CPU - 64 core, 1GHz. Main memory - approximately 500GB. GPU - GeForce RTX 2080.

3.1 Dataset

We use the well-known Solomon dataset [17] for both training and testing. The Solomon dataset is a handcrafted CVRPTW problem set with various patterns. The Solomon dataset has 3 types of instances, type R, type C, and type RC. We do not use the bare Solomon data but create a new dataset based on the Solomon dateset's *distribution* (this is standard practice in the literature).

- Type R - An instance with randomly distributed customers.
- Type C - An instance with customers distributed in a way that they are packed into several clusters.
- Type RC - A mixture of the above two types, some of customers are clustered and others are randomly located.

Each type has 2 patterns. Pattern 1 is an instance with a relatively short time horizon. The time horizon is the time within delivery to all costumers must complete and the vehicle must return to the depot. A short time horizon allows a vehicle to deliver only to a few customers, while Pattern 2 is an instance with a relatively long time horizon. We use Type R with Pattern 2 (R201) for our experiments because this type suits our use case the best. The reason for this is that in real world scenarios customers are distributed randomly. Also, the time horizon is similar to that in a business setting, in which delivery may span several hours. The same dataset is used in previous works [14]. Below is a description of our dataset.

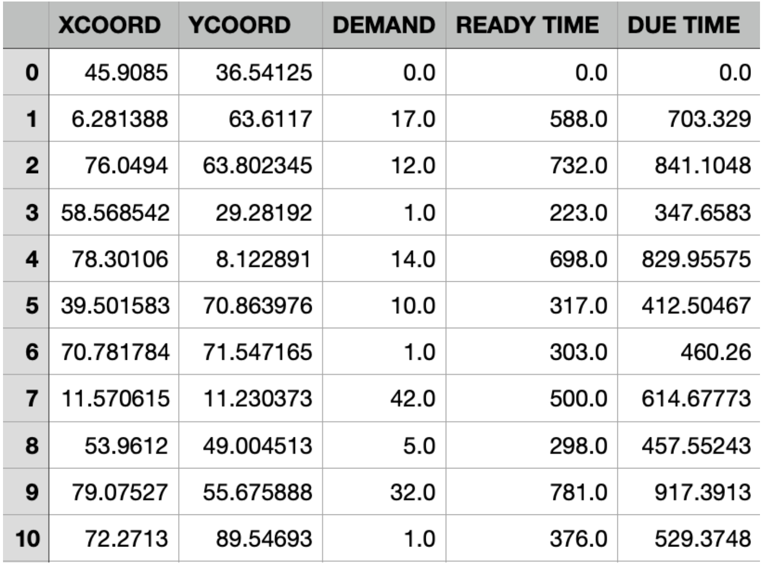
- Location
  - All customers and the depot are located in 2-dimensions. X-Y coordinates are in $[0, 100]$ for both x and y.
  - Both customers and the depot are distributed uniformly.
  - The number of customers can be arbitrary. We create the dataset with N=25,50,100,150,200.

- Demand
  - The capacity of each vehicle is 200.
  - Demand for each customer follows a normal distribution.
    $q \sim \max(\min(\mathcal{N}(15, 10), 42), 1)$

- Time window
  - Time horizon is $[0, 1000]$.
  - The start time (the beginning of a time window) is uniformly distributed.
    $s \sim U(\max(100, \text{distance from depot}), 849)$
  - The length of the time window is normal distribution. The due time (the end of a time window) is the following.
    $f \sim s + \min(\mathcal{N}(115.96, 35.78), 1000)$

The following is an example from the generated dataset. The left-most column is the index of each customer. The XCOORD and YCOORD columns represent X and Y coordinates, respectively. The DEMAND column is customer demand. The READY TIME column is the start time of the time window and DUE TIME is the end time of the time window. The first row is for the depot.

Table 3.1: An instance from the dataset.



	XCOORD	YCOORD	DEMAND	READY TIME	DUE TIME
0	45.9085	36.54125	0.0	0.0	0.0
1	6.281388	63.6117	17.0	588.0	703.329
2	76.0494	63.802345	12.0	732.0	841.1048
3	58.568542	29.28192	1.0	223.0	347.6583
4	78.30106	8.122891	14.0	698.0	829.95575
5	39.501583	70.863976	10.0	317.0	412.50467
6	70.781784	71.547165	1.0	303.0	460.26
7	11.570615	11.230373	42.0	500.0	614.67773
8	53.9612	49.004513	5.0	298.0	457.55243
9	79.07527	55.675888	32.0	781.0	917.3913
10	72.2713	89.54693	1.0	376.0	529.3748

We create the training dataset, test dataset and validation dataset independently. These three datasets are created based on the same distribution (described above).

3.2 Training

The training dataset contains 12800 instances. The number of epochs is 25 and each epoch has 512 batches with a batch size of 1024. After each epoch, the learned network parameters are saved, and the best network parameters are used at inference time. This is done by evaluating the network of each epoch on the validation dataset (1000 instances) and picking the best performing model.

During the training phase, the network learns using the REINFORCE algorithm [18], where the objective value is defined in equation 2.2. Although usually the objective value for VRP is the total number of vehicles, we set the objective value as the total working time because it better captures the cost of real-world businesses. The distance between nodes is given by the distance matrix, even for Version 1. Although Version 1 doesn't take the distance matrix as input, it uses the distance matrix just for the calculation of the distance between nodes during the decoding phase. It doesn't mean the network encodes the distance matrix to use in the decision making, but it just refers to the matrix to get distances for checking constraints and calculating the elapse time.

3.3 Hyperparameters

For our model (attention based encoder decoder model), the number of encoder layers is 2, and the number of decoder layer is 1 as mentioned in the previous chapter. Each node is encoded into a 128 dimensional vector before it is passed to the encoder layer, and the output of the encoder is embedded into a 128 dimensional vector before it is passed to the decoder layer. Adam [20] is used as an optimizer with learning rate $lr = 0.0001$. At the decoding phase, the score is clipped by tanh (Equation 2.16). The constant value C in equation 2.16 is set to 20, which means the output score is in the range [-20, 20].

3.4 Testing

The test dataset is independently generated based on the same data distribution. We conduct the testing with 1000 instances and compare the objective values. We report the average value of all instances' costs, the number of vehicles, and runtime. We conduct the experiment on three test patterns (recall that a service time is the time it takes to deliver the package to the customer upon arrival):

1. Instance without a service time.
2. Instance with a service time.
3. Instance without a service time. The distance matrix is replaced by a different distance matrix with the same distribution.

The first case is the simplest setting. The second case is an instance with a service time that is set to 10 for all customers. All tests ($N = 25, 50, 100, 150, 200$) are done with the network learned with $N = 50$. We choose $N = 50$ as the training dataset size because the training time is shorter and the model's performance is good on other size of test data. The goal of the third pattern is to evaluate the effectiveness of the direct distance matrix connection of Version 2.

Version 1 encodes the XY coordinates $\mathbf{x}_i$, therefore the network understands the XY coordinates of each node from the encoded vector. However, in the real-world the distance between nodes is not Euclidean. For instance, suppose the distance between a point A and a point B is 10 minutes in Euclidean distance. However the street between point A and point B is always busy and it takes 40 minutes to transit between these nodes. On the other hand, point C and point D are 60 minutes apart in Euclidean distance, but there is a high way between them and it takes only 15 minutes to get from one node to the other. The above scenarios often occurs in real world instances, making Euclidean distance highly unreliable.

Even though Version 1 has access XY coordinates, it is useless if the actual distance between nodes is different. On the other hand, Version 2 does not take XY coordinates but instead takes in the distance matrix. If the distance matrix has the *true* distance between nodes, Version 2 can be expected to be much more realiable in a real-world setting. In Pattern 3, we replace the distance matrix provided for evaluating the objective function. The new distance matrix is seen as a *true* distance, and the old distance matrix is the mere Euclidean distance. This simulates the above example. Since it is hard to completely characterize the relation between the Euclidean distance and the *true* distance, we replace the matrix with an independently generated distance matrix with the same distribution.

Chapter 4

Results

4.1 Performance

Table.1 shows our experimental results. Pt 1, Pt 2, and Pt 3 correspond to the test cases described in the previous section.

Table 4.1: Results of the experiments

		ORtools			Attention ver 1			Attention ver 2		
		Cost	k	Time	Cost	k	Time	Cost	k	Time
N=25	Pt 1	2692.7	3.5	119.1	2073.7	2.6	3.7	2034.7	2.5	3.3
	Pt 2	2895.3	3.7	118.6	2385.9	3.0	3.6	2377.2	3.0	3.2
	Pt 3	2785.9	3.6	122.3	2266.7	2.9	3.7	2039.7	2.5	3.2
N=50	Pt 1	4018.2	5.0	315.7	3155.7	3.8	4.4	3129.3	3.7	4.3
	Pt 2	4439.0	5.4	311.8	3783.0	4.4	4.5	3752.6	4.4	4.3
	Pt 3	4121.3	5.1	380.7	3686.0	4.4	4.3	3151.8	3.8	4.3
N=100	Pt 1	5961.1	7.3	1049.1	4923.5	5.7	8.0	5058.0	5.9	8.5
	Pt 2	6899.9	8.3	815.6	6120.5	6.8	8.0	6305.2	7.1	8.6
	Pt 3	5925.9	7.3	1082.5	6395.0	7.4	8.0	5069.2	5.9	8.5
N=150	Pt 1	7424.1	9.1	2152.3	6571.5	7.4	12.2	6787.0	7.7	13.3
	Pt 2	8992.2	10.7	2320.3	8456.4	9.3	12.4	8730.6	9.8	13.6
	Pt 3	7449.7	9.1	2757.8	8691.5	10.0	12.4	6802.8	7.7	13.5
N=200	Pt 1	8658.7	10.5	4149.0	8015.4	9.0	17.6	8324.5	9.4	19.5
	Pt 2	10798.1	12.7	3478.8	10656.5	11.7	17.4	10922.4	12.1	19.2
	Pt 3	8701.9	10.5	4224.5	10876.1	12.4	17.5	8346.2	9.4	18.9

Attention ver 1 is Version 1. Attention ver 2 is Version 2. The size of the instance for the test is varied from a small size, such as an instance with 25 nodes (N=25) to a relatively large size such as an instance with 200 nodes (N=200). Although the objective value of the test is the total working hours of all drivers (denoted as cost in the table), we report the number of vehicles (denoted as k) as well. Furthermore, as the objective of our work, shortening the running time is essential for business operations, so we also report the running time to compute a solution. The running time is the total computing time to solve 1000 instances (denoted as Time, measured in seconds).

Our results show that the running times for both Version 1 and Version 2 are much shorter than ORtools in all cases. The running time of ORtools increases exponentially with the size of an instance. On the other hand, our running time increases almost linearly with the size of the instance. Regarding the performance (quality of the solution), both Version 1 and Version 2 outperform ORtools on small instances on pattern 1 and 2. The performance is about the same level on large instance (N=200). Both Versions result in a smaller number of the vehicles than ORtools.

4.2 Learning process

As mentioned in the previous section, the training is done with a dataset where N=50. Figure 4.1 shows the learning curves for both Versions.

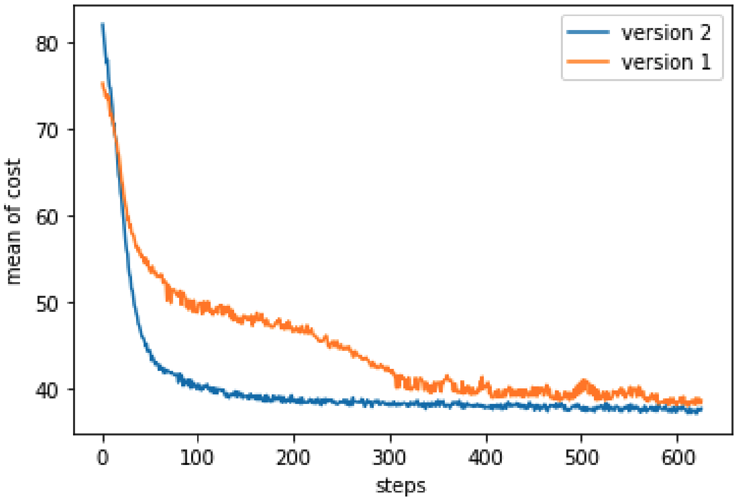


Figure 4.1: The learning curve for Version 1 and Version 2.

Both versions converge, however Version 2 converges faster than Version 1. As mentioned in the previous section, the training is done with learning rate $lr = 0.0001$. The approximate computing time for the training for 25 epochs is 123 minutes for Version 1 and 129 minutes for Version 2.

4.3 Solution examples

Next we plot several solution examples. We compare our solutions to OR-tools. The black dots are customers and the red dot is the depot. The XY axes of the figure correspond to the coordinates of customers (and the depot's) location. Arrows show the movement of vehicles between nodes, and arrows with the same color indicate that the delivery is done by the same vehicle.

A notable difference between ORtools and our results is the number of vehicles (k). In the figures below , ORtools uses $k = 5$ vehicles and both Version 1 and 2 use only $k = 3$ vehicles. This is in line with the general trend observed in Table 4.1.

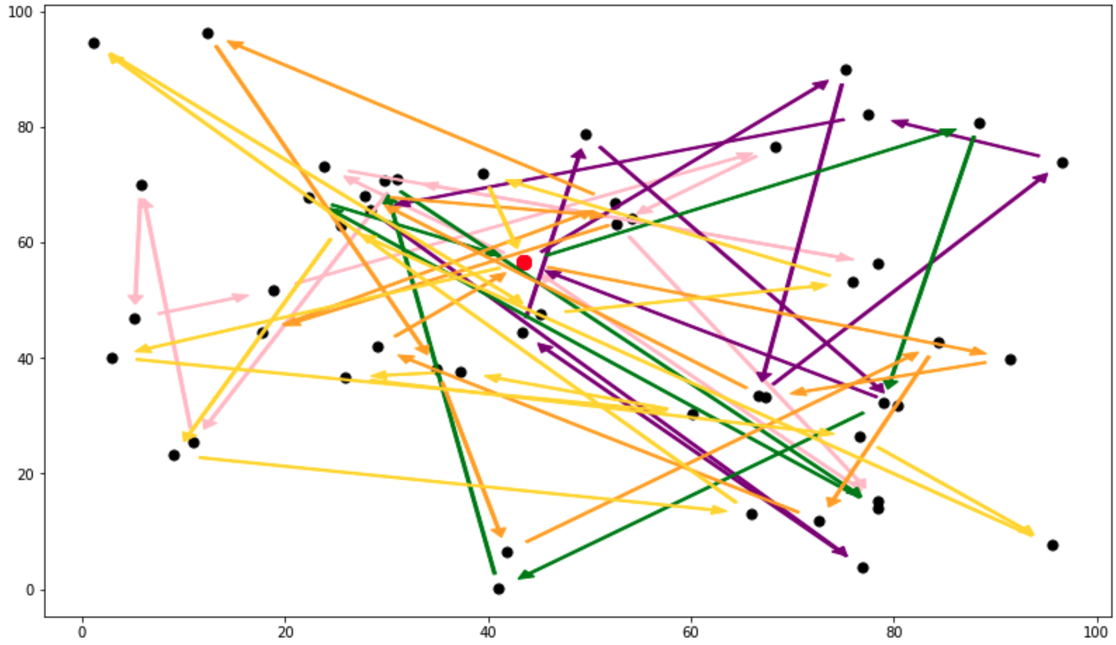


Figure 4.2: ORtools solution example with $N = 50$

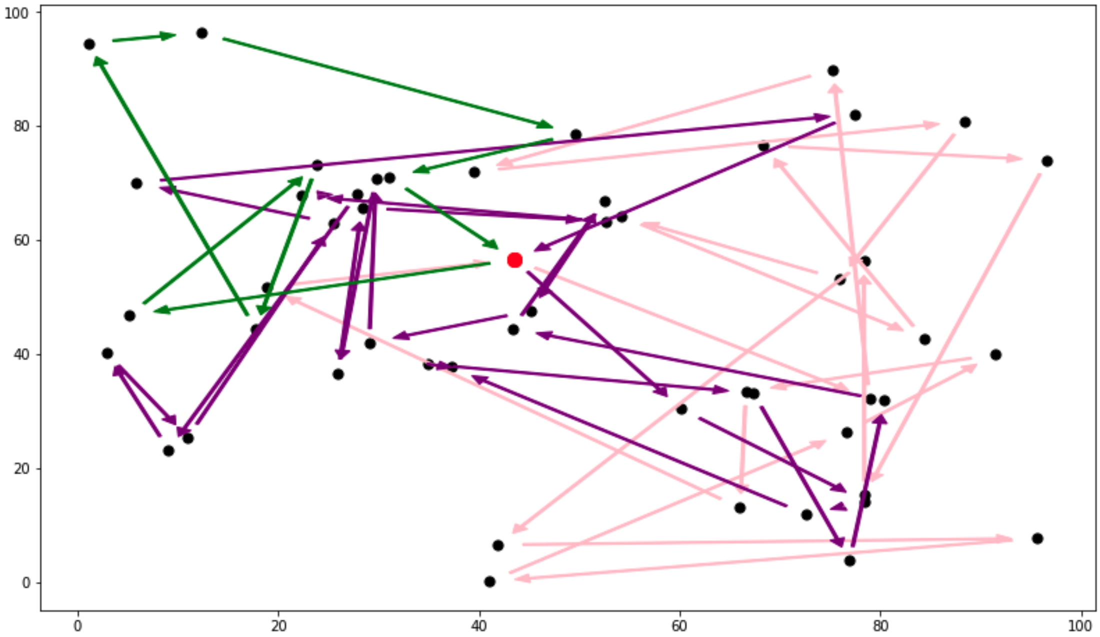


Figure 4.3: Version 1 solution example with $N = 50$

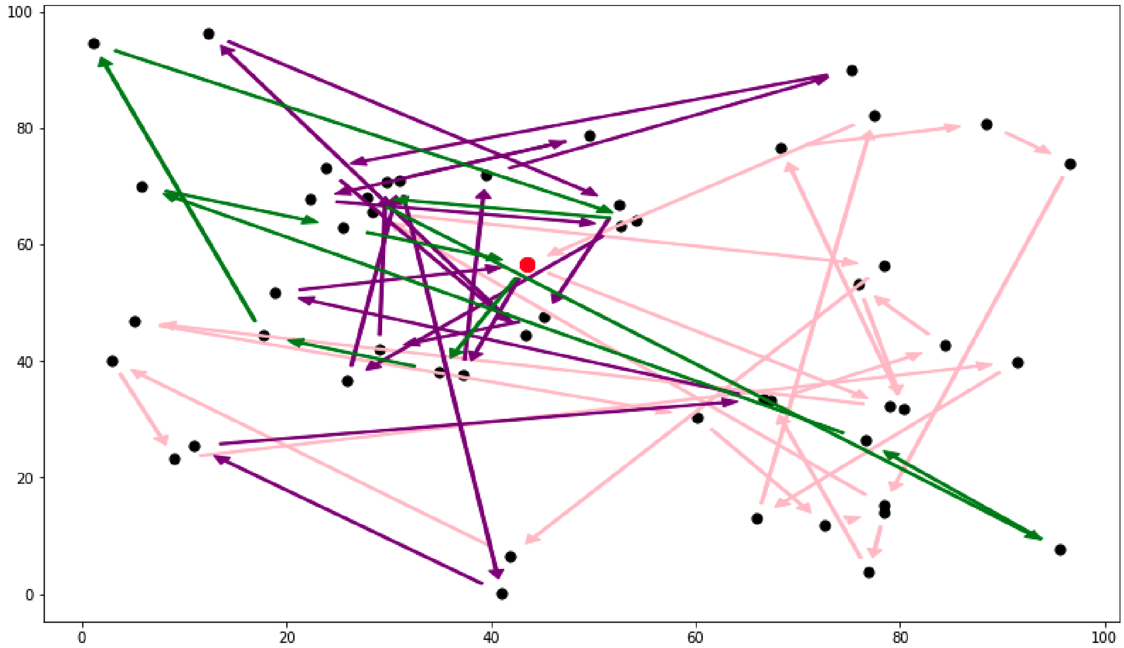


Figure 4.4: Version 2 solution example with $N = 50$

Compared to ORtools' solution, the solutions of both Versions tend to find a near node as the next node in the route. Due to the capacity constraint and the time window constraint, it is not always possible to visit the nearest customer. However, Version 1 takes the nodes' XY coordinates into consideration because it encodes the nodes' Euclidean coordinates; it tends to choose neighbor nodes to reduce the cost. Similarly, since Version 2 takes in the distance matrix, it tends to choose neighbor nodes too. Nevertheless, we can still observe that sometimes far away nodes are chosen for the route. This is because the customer distribution is not very dense in a small instance, and it is not possible to pick neighbors every time.

Contrasted with our approach, ORtools uses local search. In local search, a random initial solution is created, and then modified by swapping nodes or inserting nodes iteratively. Thus, the routes appear to be more evenly constructed.

The figures below show examples of solution plots for instances where $N = 25$. Since the density of customers is smaller than for instances where $N = 50$, routes are less densely packed than in the previous plots. In this example, the solution generate by ORtools uses a larger number of vehicles, $k = 3$, compared to our models.

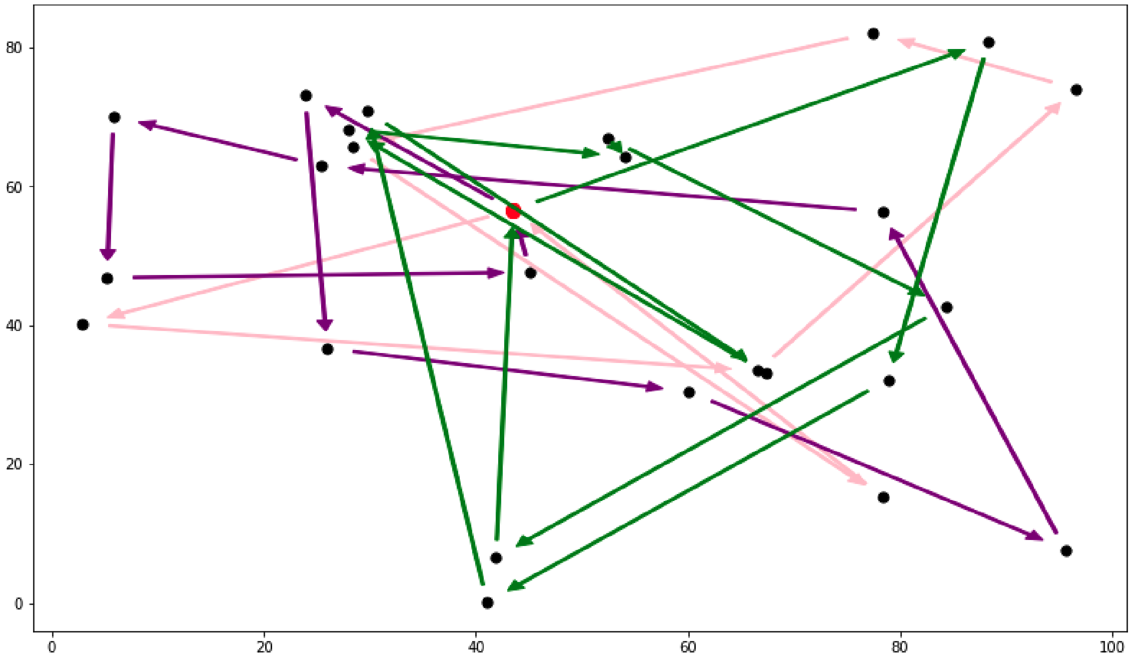


Figure 4.5: ORtools solution example with $N = 25$

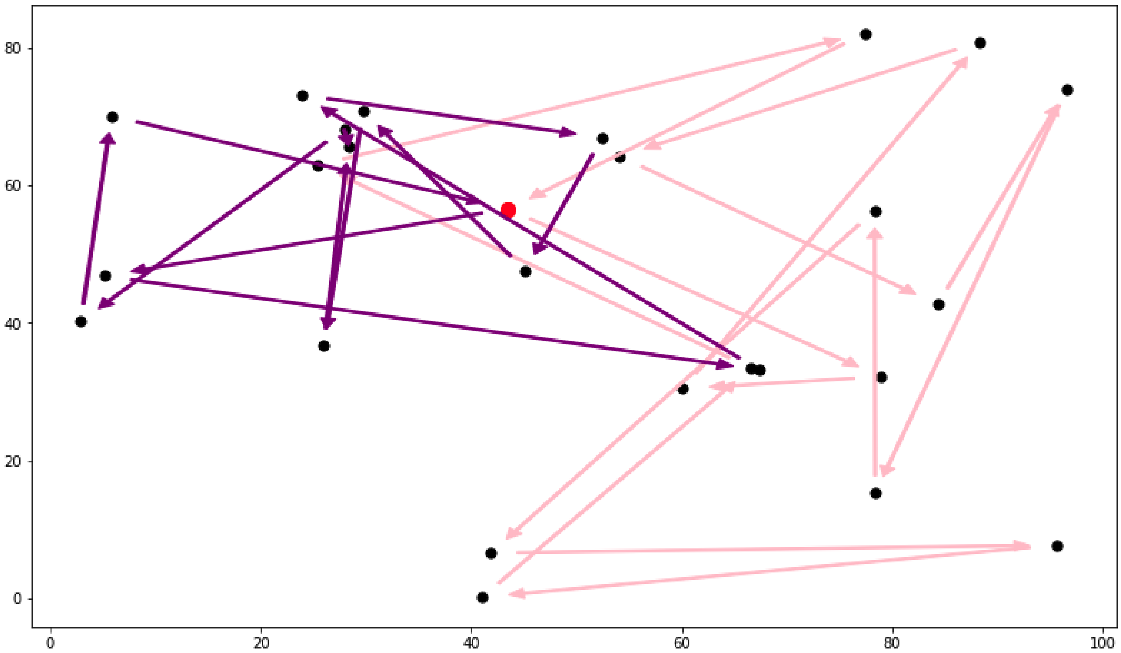


Figure 4.6: Version 1 solution example with $N = 25$

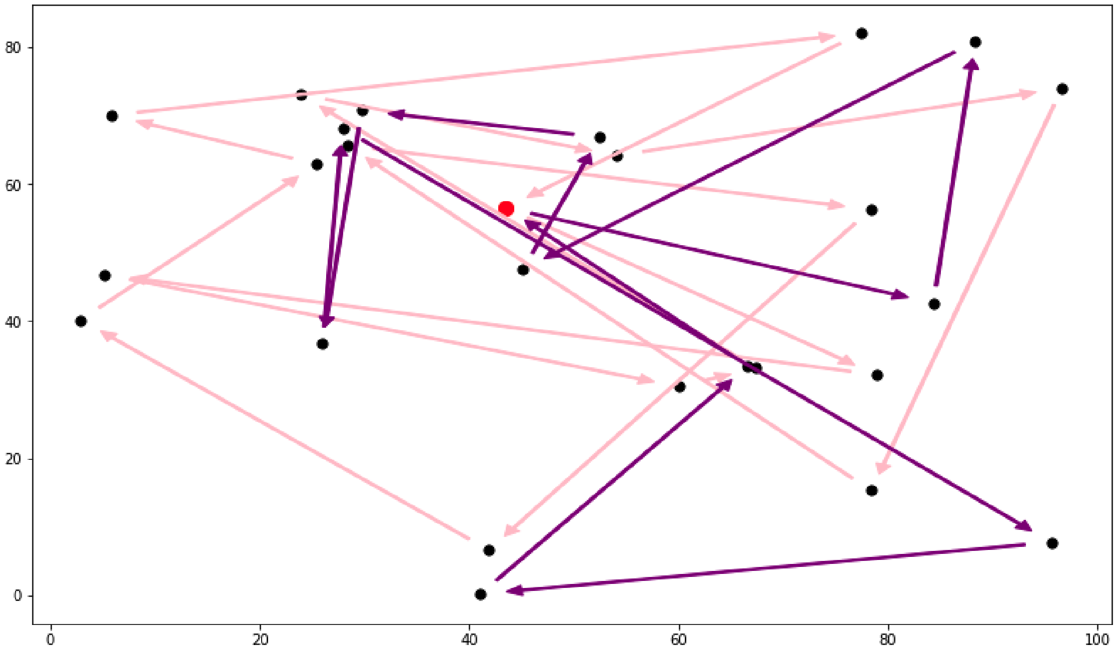


Figure 4.7: Version 2 solution example with $N = 25$

The figures below are for instance of size 150 ($N = 150$). The plot looks messy because the customers are densely packed and the number of vehicles is larger than the in previous plots. The key takeaway is that both Versions tend to pick neighboring nodes for some vehicles but it is not always the case. The models pick nodes greedily. This means that in the decoding phase the routes created earlier have a much larger selection of nodes to visit, compared to those created later. Toward the end of the decoding phase, the number of available nodes becomes very small because nodes are already assigned to existing routes. Meanwhile, ORtools exhibit a good balance, because the local search modifies all routes iteratively and there is less bias between routes.

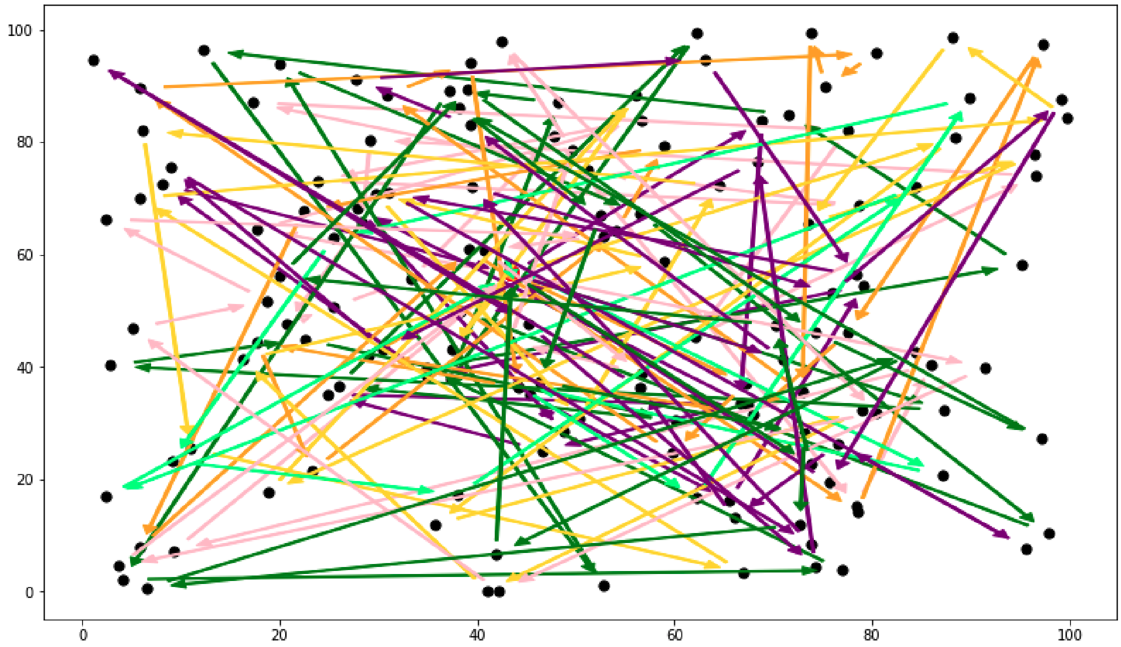


Figure 4.8: ORtools solution example with $N = 150$

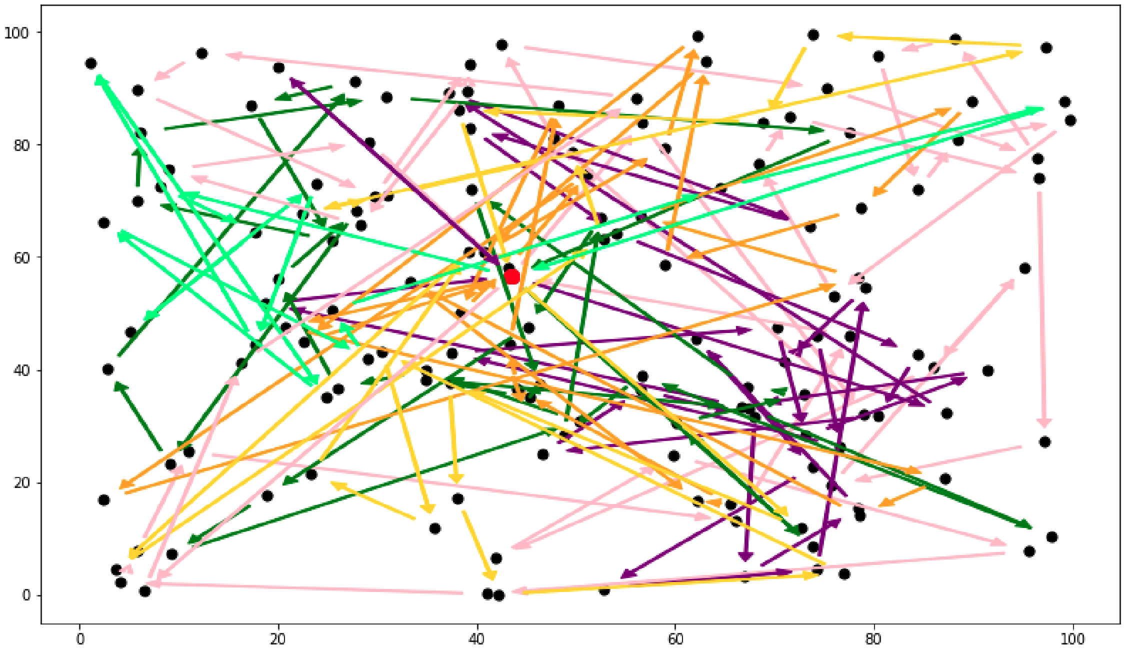


Figure 4.9: Version 1 solution example with $N = 150$

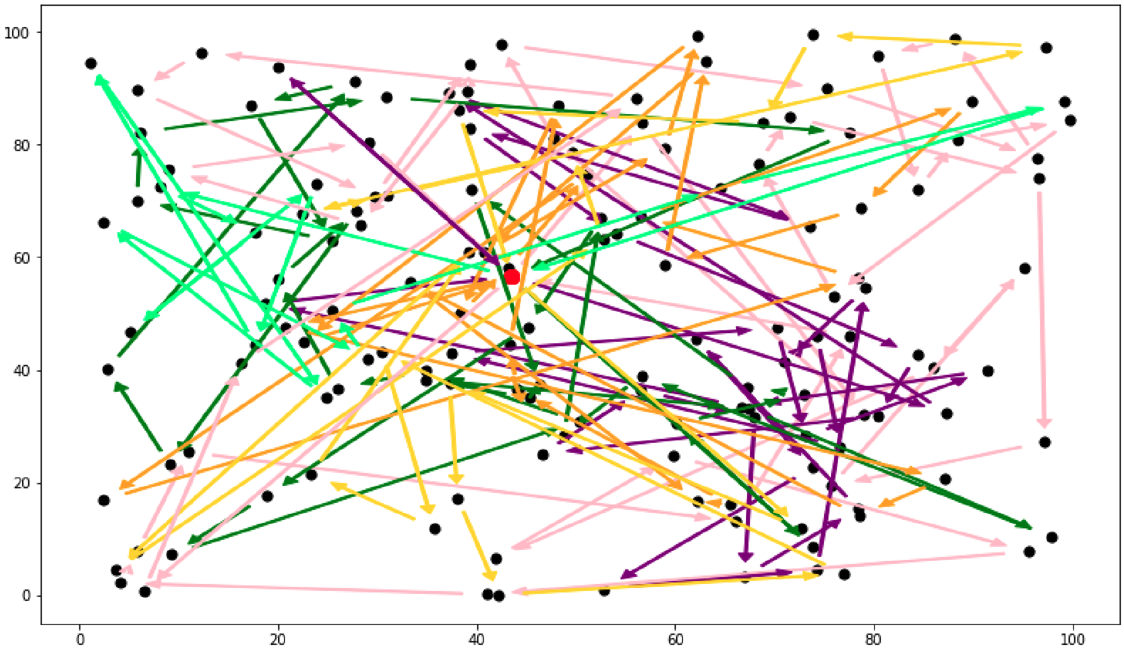


Figure 4.10: Version 1 solution example with $N = 150$

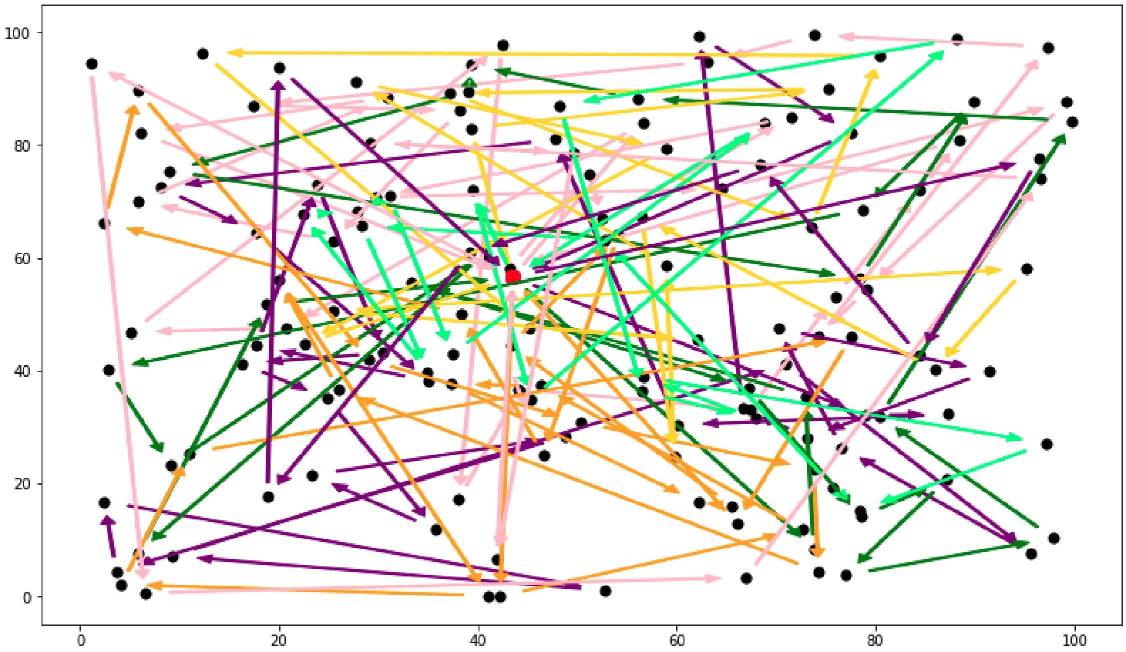


Figure 4.11: Version 2 solution example with $N = 150$

4.4 Discussion

Overall, the performance of both versions is competitive enough to use in practice on Pattern 1 and 2 instances. Even on relatively large instances ($N = 200$), both Versions achieve good solutions (compared to ORtools). The performance of Version 1 and Version 2 is similar on Pattern 1 and 2 instances.

The main difference is the performance on pattern 3 instances, where Version 2 achieves better performance. Recall that Pattern 3 are the instance where the distance matrix use to evaluate the objective function is replaced with a different matrix taken from the same distribution, and that Version 1 encodes Euclidean coordinates to determine where customers are located. However, after changing the matrices, the XY coordinates are no longer reliable.

That is, the network makes a decision based on unreliable distance information. Table 4.1 shows that the cost of Version 1 gets worse on Pattern 3 compared to Pattern 1 for all sizes. On the other hand, Version 2 uses the distance matrix to make decisions, instead of relying on nodes' XY coordinates.

The result shows that the ORtools gives better performance than Version

1 on a large instance ($N = 150, N = 200$) which is intuitive. However, Version 1 achieves better performance on a small instances ($N = 25, N = 50$).

Trying to understand why Version 1 outperforms ORtools for small instances, we must look deeper at the objective values achieved. Since Pattern 1 and 3 have no service time, the cost consists of the distance and the waiting time (when the vehicle arrived at a node before the start time of the time window, it has to wait). Table 4.2 shows the values of these factors of pattern 1 and pattern 3 on instance size $N = 50, N = 100$ and $N = 150$. Specifically, *distance* denotes the total transit time and *wait* denotes the total waiting time. The biggest difference between our models and ORtools is that the total transit time of ORtools' solution is smaller than that of our models. Although ORtools takes the waiting time into consideration as the objective value, the solution tends to have more waiting time than our models. That is, as the instance becomes bigger the transit time takes up a larger fraction of the total value of the objective function. Therefore the larger the instance gets, the less competitive version 1 becomes on Pattern 3 instances (as it does not have access to the actual distances between clients).

Table 4.2: Comparison of total transit time and waiting time

		ORtools		Attention ver 1		Attention ver 2	
		distance	wait	distance	wait	distance	wait
	Pt 1	1136.5	2881.7	2131.9	1023.8	2146.4	982.9
N=50	Pt 3	1134.5	2986.8	2417.9	1268.1	2141.7	1010.1
	Pt 1	1719.7	4241.4	3636.5	1287.0	3773.4	1306.6
N=100	Pt 3	1722.1	4203.8	4384.8	2010.2	3784.3	1284.9
	Pt 1	2186.8	5237.3	5009.1	1562.4	5504.9	1282.1
N=150	Pt 3	2193.1	5256.6	6355.7	2335.8	5529.5	1273.3

Next we compare the distribution of distances between consecutive nodes the constructed routes for Version 1. We compare the distribution between Pattern 1 and 3 for large ($N = 150$) and small ($N = 50$) instances.

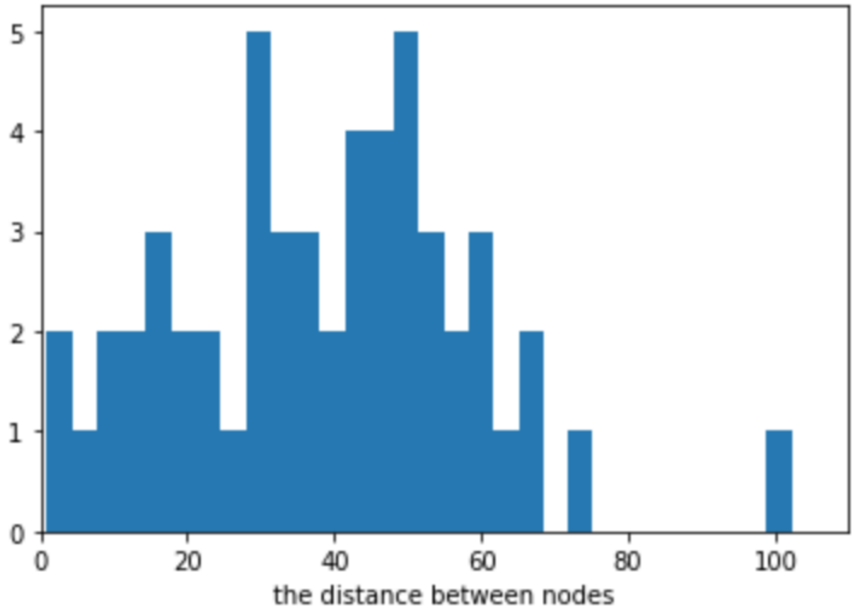


Figure 4.12: The example of the distance distribution of routes on Pattern 1, N=50

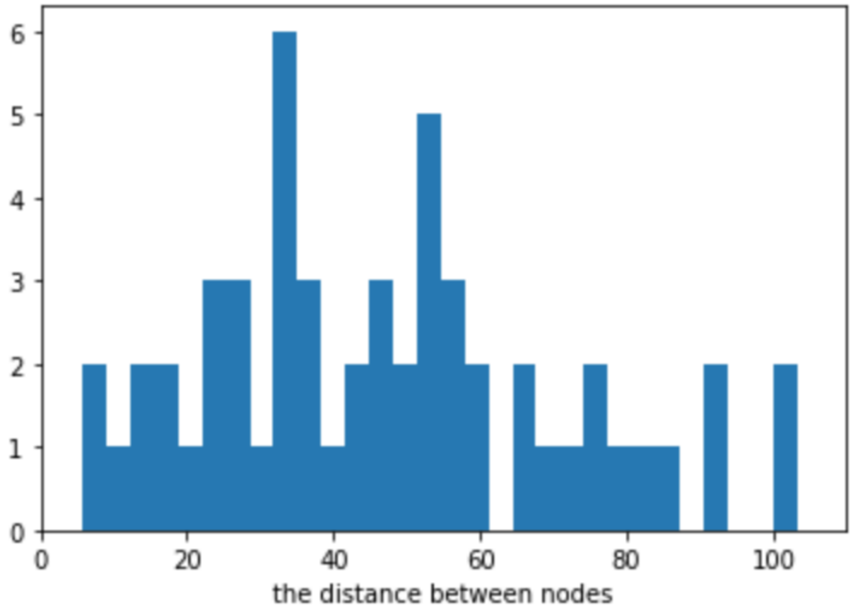


Figure 4.13: The example of the distance distribution of routes on Pattern 3, N=50

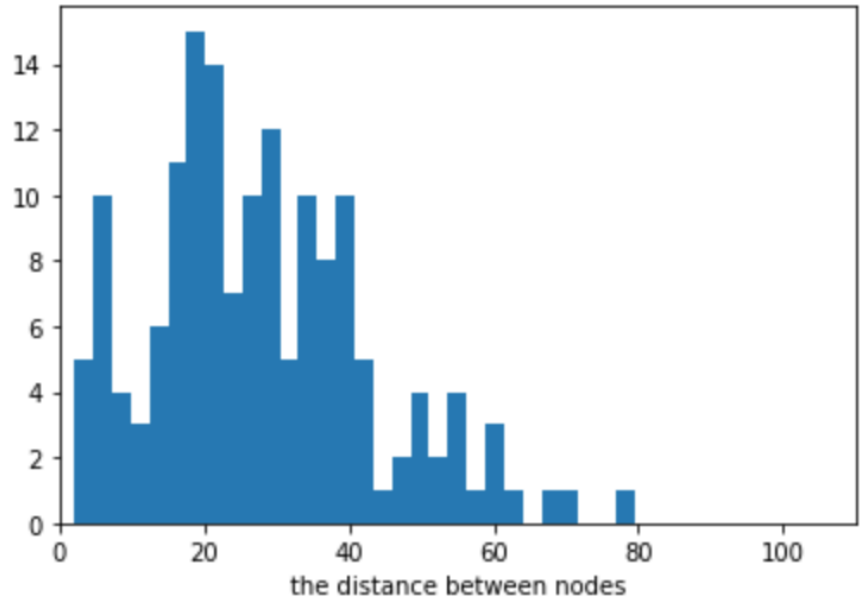


Figure 4.14: The example of the distance distribution of routes on Pattern 1, N=150

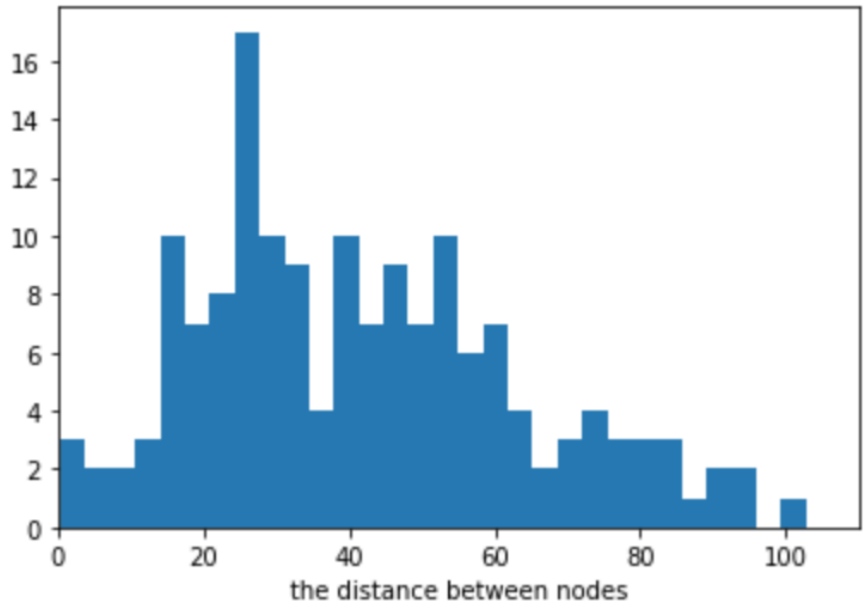


Figure 4.15: The example of the transit time distribution of routes on Pattern 3, N=150

Since the customer location is less dense in the small instances, the transit time between nodes tends to be larger even in Pattern 1 solutions. Comparing Figure 4.12 and Figure 4.13, the transit time distribution shifts to the right because the network does not have access to the distance matrix. The total transit time of the solution for Pattern 1 ($N = 50$) instance is 2131.9, and for Pattern 3 ($N = 50$) is 2417.9. Thus, after replacing the distance matrix, the total transit time increases by 13.4%. On the other hand, comparing Figure 4.14 and Figure 4.15, the transit time distribution also shifts to right, however, the total of the transit time of the solution increases by 26.9% (the total transit time on pattern 1 ($N = 150$) is 5009.1, and that of the transit time on pattern 3 ($N = 150$) is 6355.7).

Therefore the larger the instance becomes the transit times becomes a more important part of the total objective, which explains the phenomena we observed.

Chapter 5

Conclusions

5.1 Conclusion and future work

In this study, we proposed two attention based networks (Version 1 and 2) for CVRPTW. We achieve the following results:

1. We successfully extend previous work to support time window constraints. The experimental results shows that both versions achieve a solution quality similar to that of ORtools, with a much faster running time.

2. While Version 1 assumes Euclidean distance between nodes, Version 2 accepts a distance matrix directly as input. We believe it is more useful in practice, as a distance matrix is a much more robust way of encoding distance, compared to XY coordinates. Version 1's performance is slightly better, but Version 2 still outperforms ORtools on small instances.

5.2 Future work

As future work, we would like to use our attention based solver for real-world instances and get detailed feedback. There is an abundance of research regarding VRP (CVRPTW) in the academic setting, however, major solvers that are used in real world business setting are based on heuristic algorithms that are slow on large instances.

We believe that our work is important for the logistics field because there is no solver that is able to solve CVRPTW within a reasonably short time on large instances. In Japan, there are several delivery companies that deliver tens of thousands of parcels every day. In Japan, an increased demand for packages is predicted, while the number of delivery drivers is expected to remain the same. Thus, it is critical to increase the efficiency of delivery services. CVRPTW solvers are a key tool for these businesses. As mentioned above, heuristic based solvers take a prohibitively long time to execute on large instances, which makes it hard to implement customer friendly delivery services such as delivery within X hours and same-day-delivery. We believe that our attention based solver is able to solve the typical CVRPTW instance sufficiently fast while achieving good performance. Therefore, the next natural step is to apply our solver in real-world businesses settings.

Bibliography

[1] Chris Voudouris and Edward Tsang. Guided Local Search, *Technical Report CSM-247*, (1995)

[2] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks, *In Advances in Neural Information Processing Systems*, pp. 2692–2700, (2015)

[3] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. arXiv preprint arXiv:1611.09940, 2016.

[4] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems, *International Conference on Learning Representations*, (2019)

[5] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Li, and Yoshua Bengio. Graph attention networks. *In International Conference on Learning Representations*, 2018.

[6] Rongkai Zhang, Anatolii Prokhorchuk and Justin Dauwels, Deep Reinforcement Learning for Traveling Salesman Problem with Time Windows and Rejections, IEEE WCCI 2020

[7] Ke Zhanga, Fang Heb, Zhengchao Zhanga, Xi Lina, Meng Lia. Multi-Vehicle Routing Problems with Soft Time Windows: A Multi-Agent Reinforcement Learning Approach. *Transportation Research Part C: Emerging Technologies*, ISSN: 0968-090X, vol 121, pp 102861

[8] Lin, B., Ghaddar, B., Nathwani, J. Deep Reinforcement Learning for the Electric Vehicle Routing Problem With Time Windows. *IEEE Transactions on Intelligent Transportation Systems*

[9] Joe, W. and Lau, H.C., 2020. Deep reinforcement learning approach to solve dynamic vehicle routing problem with stochastic customers. *In*

Proceedings of the International Conference on Automated Planning and Scheduling (Vol. 30, pp. 394-402).

[10] Sultana, N. N., Baniwal, V., Basumatary, A., Mittal, P., Ghosh, S., Khadilkar, H. Fast Approximate Solutions using Reinforcement Learning for Dynamic Capacitated Vehicle Routing with Time Windows. arXiv preprint arXiv:2102.12088.

[11] Lei Gao, Mingxiang Chen, Qichang Chen, Ganzhong Luo, Nuoyi Zhu, and Zhixin Liu. Learn to design the heuristics for vehicle routing problem. arXiv preprint arXiv:2002.08539, 2020.

[12] Xin, L., Song, W., Cao, Z., Zhang, J.. NeuroLKH: Combining Deep Learning Model with Lin-Kernighan-Helsgaun Heuristic for Solving the Traveling Salesman Problem. arXiv preprint arXiv:2110.07983

[13] K. Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[14] Jonas K. Falkner. Lars Schmidt-Thieme Learning to Solve Vehicle Routing Problems with Time Windows through Joint Attention. arXiv preprint arXiv:2006.09100.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Infor- mation Processing Systems*, pp. 5998–6008 (2017)

[16] Laurent Perron and Vincent Furnon. Or-tools. URL https://developers.google.com/ optimization/

[17] LMarius M Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2):254–265, 1987.

[18] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[19] Steven J Rennie, Etienne Marcheret, Youssef Mroueh, Jerret Ross, and Vaibhava Goel. Self-critical sequence training for image captioning. *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 7008–7024, 2017.

[20] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.