Deep Learning Assisted Heuristics and Exact Methods for the Vehicle Routing Problem with Side Constraints

Vom Promotionsausschuss der
Technischen Universität Hamburg

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation (Monographie)

von
Jorin Dornemann

aus
Düsseldorf

2025

1. Gutachter: Prof. Dr. Anusch Taraz
2. Gutachter: Prof. Dr. Kathrin Fischer

Tag der mündlichen Prüfung: 26. März 2025

doi: https://doi.org/10.15480/882.15061
ORCiD: https://orcid.org/0000-0002-8053-0284

Summary

This thesis focuses on solving vehicle routing problems (VRP) with side constraints that complicate the process of efficiently finding optimal solutions. The goal is to develop approaches that require less extensive expert knowledge compared to existing highly specialized exact and heuristic methods, while still delivering close to optimal solutions.

To achieve this, we develop novel approaches for solving the capacitated vehicle routing problem with time windows (CVRPTW) by incorporating new deep learning models into heuristic combinatorial optimization methods. For this, we design graph convolutional neural networks that can efficiently extract relevant information from the graph input with respect to the optimization problem. These networks generate graph representations to predict the most promising edges to be included in the optimal solution. These predictions are then integrated into heuristic approaches to solve vehicle routing problems by building solutions sequentially. Firstly, we employ a limited-width-breadth-first tree search that iteratively constructs a solution by adding one vertex per iteration. This approach maintains only a limited number of search tree nodes at each layer, guided by a scoring policy informed by the neural network's predictions. Secondly, we develop a Monte Carlo Tree Search for the CVRPTW. This method aims to balance the exploration of the entire search space with the exploitation of the most promising areas, leveraging the predictions of the neural network within the search.

Additionally, we explore new methods for integrating novel hardware techniques specialized in solving quadratic unconstrained binary optimization problems by quantum(-inspired) computation. While the current state of quantum computing hardware cannot handle optimization problems of practical size, we explore methods to incorporate such hardware into frameworks that utilize its computational capabilities without exceeding its limitations to solve vehicle routing problems. For this, we employ a deep learning-based problem reduction to reduce the size of the instance representation formulated as a quadratic unconstrained binary optimization problem, which is subsequently solved using specialized quantum-inspired computing hardware. Furthermore, we design a hybrid heuristic that leverages the strengths of this specialized hardware in combination with deep learning-assisted heuristic methods to find optimal solutions while providing scalability for larger instances. The heuristic is structured into three phases, each addressing different aspects of the optimization process. This approach allows us to leverage quantum-inspired computing for solving binary optimization problems while addressing side constraints, which are challenging to incorporate within a binary framework, through heuristic methods in their respective phases. We then utilize quantum-inspired computing hardware to execute the developed heuristic.

Contents

Chapter 1

Introduction

1.1 Motivation

The Capacitated Vehicle Routing Problem (CVRP), a generalization of the Travelling Salesman Problem (TSP), is one of the most studied combinatorial optimization problems. In the CVRP, a fleet of vehicles must be routed to deliver goods from a depot to a number of customers. Each customer's good has a specific size, so the capacity of the vehicles must be taken into account when deciding on the routing. Finding a solution that minimizes the total distance traveled is NP-hard [TV14].

First introduced by George Dantzig and John Ramser in 1959 [DR59], who proposed a simple heuristic to solve this problem in the context of petrol deliveries, numerous approaches have been proposed over the years to solve vehicle routing problems. Exact algorithms such as branch-and-bound with their derivatives branch-and-cut and branch-and-price offer optimal solutions, but are computationally demanding and therefore often cannot be applied to larger-sized problems. Alternatively, heuristics such as Large Neighborhood Search or genetic algorithms offer approximate solutions with lower computational costs than exact methods, often providing scalability and adequate solution quality for large instances where exact techniques fail. What these methods have in common is that they are based on highly specialized, hand-crafted algorithms that require expert knowledge to efficiently extract the relevant information and guide the search for an optimal solution.

Over the years, a number of different variants of the CVRP have been proposed, often motivated by real-world applications with additional constraints. One of the most well-known variants of the CVRP is the Capacitated Vehicle Routing Problem with Time Windows (CVRPTW), where customers not only have a demand but also a time window that specifies the allowable time periods for deliveries for each customer. First introduced by Marius Solomon in 1987 [Sol87], the CVRPTW finds application in a variety of practical contexts, including delivery scheduling, emergency response planning, and supply chain management. Other variants of the CVRP include the consideration of heterogeneous vehicle fleets, customers who not only have a delivery demand but also a pickup requirement, or the consideration of multiple depots, all based on real-world applications. What they all have in common is that finding the optimal routes for the vehicles that minimize the total

driving distance while taking into account the constraints remains a challenging problem.

On the other hand, Machine Learning (ML) has celebrated many successes recently and is very present in the public perception. In the context of optimization, however, machine learning is not trivial to apply, as it is not possible to simply assign an artificial intelligence (AI) the task of finding the best solution as optimization problems often involve complex constraints, objective functions, and solution spaces that are not directly compatible with standard machine learning approaches, which are typically designed to recognize patterns in data rather than systematically search for the best solution. Additionally, machine learning models require extensive training on relevant data, and their predictions or decisions may not always satisfy the strict requirements of optimization problems. Integrating AI into optimization requires combining data-driven insights with domain knowledge and specialized algorithms. But since [VFJ15] has shown that combinatorial optimization problems can be tackled effectively using deep learning methods, these techniques are increasingly being used to solve combinatorial optimization problems. Since combinatorial optimization problems have a high-dimensional solution space, ML techniques can help classical approaches to guide them more efficiently through the large search spaces and thus accelerate the approaches and increase the quality of the solutions found. The combination of recent advances in the field of artificial intelligence with classical methods has the potential to revolutionize the field of optimization [BLP21].

In addition, interest in quantum computers and their potential for solving complex optimization problems that exceed the capabilities of classical computers has increased recently. Quantum computers are able to solve certain types of optimization tasks much faster and more efficiently than classical computers [vA20]. As quantum computers may become more advanced and accessible in the near future, the development of models for optimization problems that can take advantage of their unique capabilities will become more important. However since these hardware systems can only handle a limited number of variables, it is crucial to develop new scalable methods for integrating this hardware into optimization frameworks.

In this thesis, we will focus on the CVRPTW and explore new methods for approximately solving the CVRPTW by developing novel approaches to combine deep learning with combinatorial optimization techniques as well as quantum-inspired computing techniques. Our goal is to develop approaches that require less in-depth expert knowledge than the existing highly specialized exact and heuristic methods, while at the same time providing high-quality solutions.

1.2 Main Results

In this section, we provide an overview of the most important results of this work. For this purpose, the various approaches that are developed and presented in more detail in the following chapters are briefly outlined.

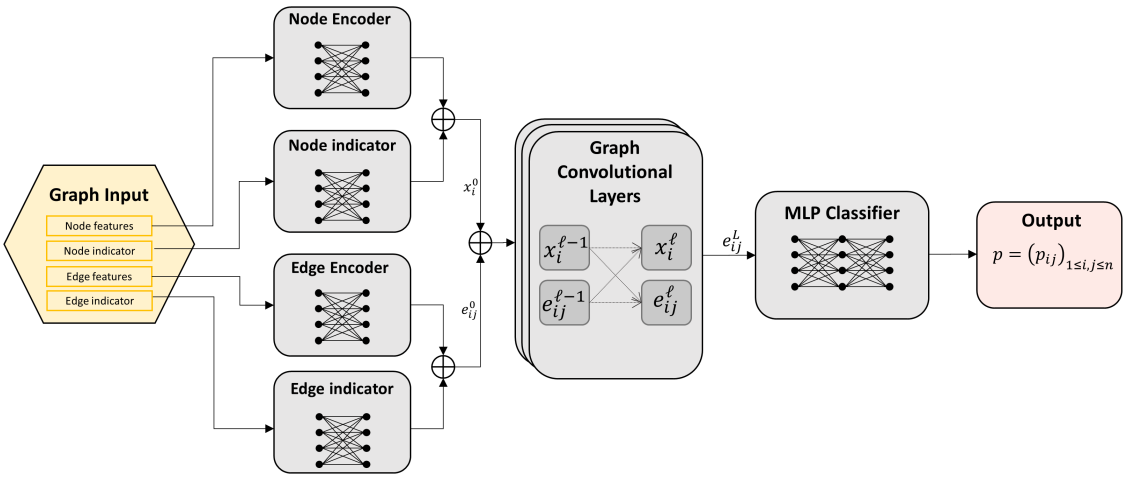


Figure 1.1: Graph Convolutional Network Diagram for the CVRPTW

1.2.1 Deep Neural Network Assisted Tree Search

We develop a new deep neural network for the CVRPTW to assist in finding solutions by providing what can be interpreted as a probability distribution over the set of edges of the graph representing a CVRPTW instance. For this, we build on the work of [BL17], who propose the Residual Gated Graph Convnet, a neural network structure that is especially suited to extract information of graph-structured data of arbitrary size and find efficient and meaningful representations of graphs.

We use this model to develop a graph convolutional neural network for the CVRPTW, which takes the instance as input and assigns a value to each edge in the graph as output, which indicates how likely it is that this edge is contained in the correct solution. The input consists of the node features and edge features as well as other indicator functions that declare the depot as a special node, for example. The node features consist of the coordinates, the demand and the time window at this node. The edge features consist of the travel times that are assigned to the edges.

All this information is then embedded in an $h-$dimensional space, where $h \in \mathbb{N}$ denotes the number of parameters each hidden state within the network stores, before they are passed on to the convolutional part of the neural network, which consists of a number of stacked layers through which all information is passed. Within each of these layers, the embeddings of the nodes x_i and edges e_{ij} are updated, using the representations of neighboring nodes and edges from the previous layer, combined with learnable parameters.

Finally, the edge representations e_{ij}^L of the last convolutional layer are passed to a multi-layer perceptron, which is a fully connected feedforward neural network, which then assigns a value p_{ij} to each edge (i, j). Figure 1.1 provides a simplified overview of the network's structure.

The network is trained in a supervised manner, i.e. we give the network a training dataset consisting of pairs of inputs and targets. The inputs are instances of the CVRPTW, the targets are $n \times n$ adjacency matrices with an entry equal to 1 if the corresponding edge

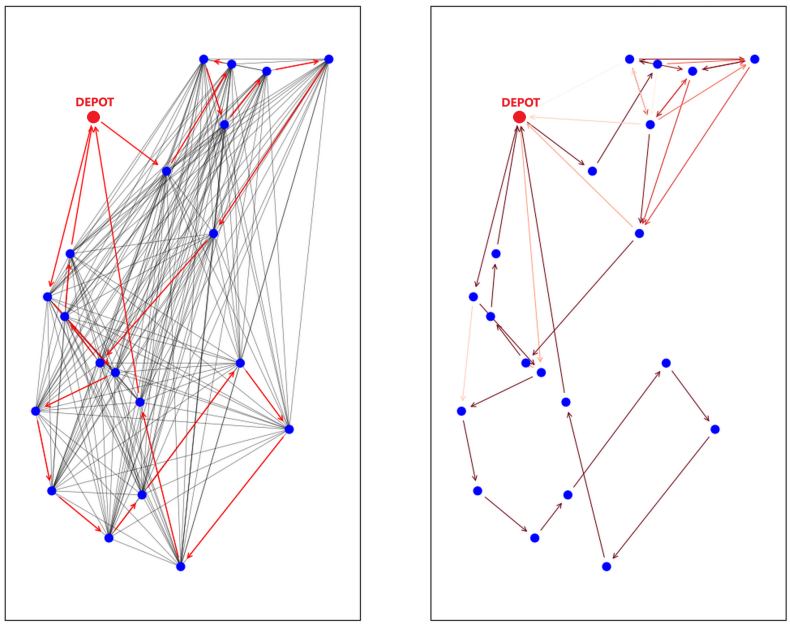


Figure 1.2: Example Output of the Graph Convolutional Network

is contained in the optimal solution. An extensive training procedure is used to train the model parameters by minimizing the cross-entropy loss using the gradient descent method.

The output of the network can be interpreted as a heatmap over the set of edges, indicating promising edges with higher values. Figure 1.2 shows an example of the output of the neural network. On the left side, we see a graph with 20 customers, the depot (marked in red), as well as the correct solution of the instance, shown as red edges, consisting of two routes starting and ending at the red depot node. The edges of the complete graph on 20 nodes can be seen in gray in the background. The right side shows the heatmap of the neural network, in which only edges with a probability of at least 25% are shown. The higher the assigned value to the edge, the darker the red color in which the edge is represented, where the most probable edges appear brown. As we can see, for this instance the neural network is able to identify most of the correct edges as being highly probable to be contained in a solution.

This offers great opportunities to build solutions by using the output of the neural network. However, the question also arises as to how this can be done efficiently, as simply selecting the edges with the highest probabilities is not guaranteed to generate valid solutions with respect to the side constraints.

Our first approach is to apply a limited-width-breadth-first tree search, also called a beam search, to iteratively construct solutions [Dor23]. By considering the network's values during this process, we follow the paths with the highest probabilities while considering the validity of the built solutions. We start with the depot node 0 and iteratively build a search tree. Each node in the search tree represents a partial solution, and in each layer of the search tree one customer node is added to the partial solution represented by the

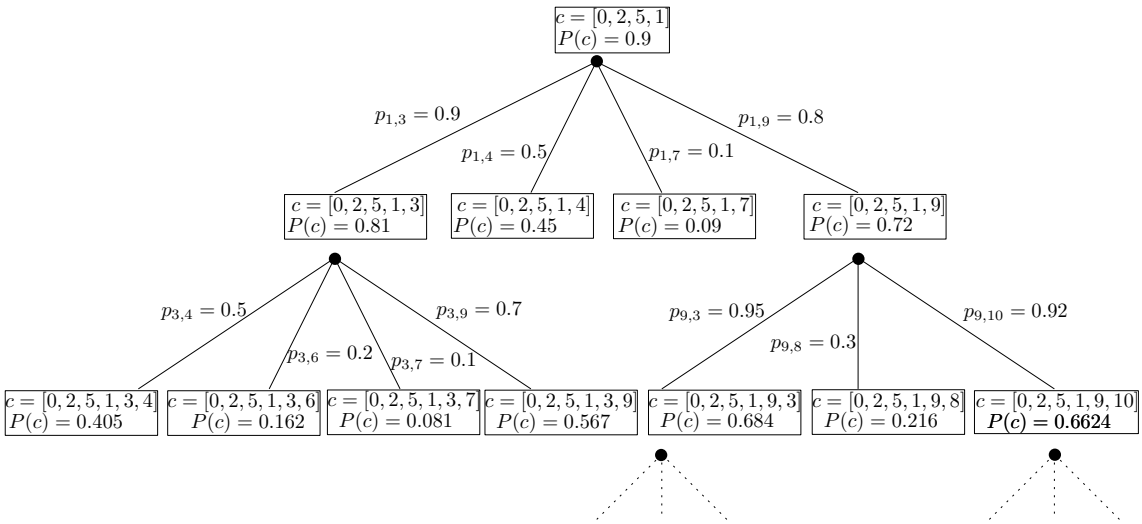


Figure 1.3: Beam Search Example

parent search tree node. In each iteration, we select the b most promising leaves in the search tree with respect to a scoring policy. The scoring policy P of a (partially) built tour $c = [v_1, \ldots, v_k]$ is given as

$$P(c) = \prod_{i \sim j} p_{ij},$$

where $i \sim j$ denotes that node v_j follows node v_i in c.

For each of the b selected leaves in the search tree, we add child vertices corresponding to adding a customer node to the partial solution that is represented by the search tree node. While adding the child vertices, we consider feasibility with respect to the capacity and time window constraints, assuring that we only build valid solutions. After adding all valid child vertices to the b selected leaf nodes, we proceed to the next iteration. In this iteration, we evaluate the scoring policy once again to select the b most promising leaves in the newly created layer of the search tree. Therefore, by the end of the search tree, we have built b valid solutions that are most promising with respect to the neural network's predicted edge probabilities. For each of the b solutions we calculate the objective value and return the solution that minimizes this.

Choosing $b = n2^n$ would result in an asymptotically optimal and therefore exact method. However, selecting smaller values for b allows us to reduce the search space, thereby optimizing computational performance, though this may result in pruning the best solutions during the process.

Example 1.1. *Figure 1.3 shows an example of the execution of a beam search. Given a graph G with $n = 10$ nodes, we want to apply a beam search with beam width $b = 2$. We start with a partial solution $c = [0, 2, 5, 1]$ that has a scoring policy value of $P(c) = 0.9$.*

From the current partial solution, we generate all possible child nodes by adding one more node to the current sequence that has not yet been included in c and that results in a feasible partial solution with respect to the side constraints. In this case, this holds true

for the vertices $\{3, 4, 7, 9\}$*. Therefore, four child nodes are added and the scoring policy is evaluated for each of the new partial solutions. As the scoring policy is the highest for the partial solutions* $c = [0, 2, 5, 1, 3]$ *and* $c = [0, 2, 5, 1, 9]$*, these two search tree nodes are selected and further explored, while the others are disregarded.*

In the next layer, the partial solution $c = [0, 2, 5, 1, 3]$ *has 4 valid expansions, i.e. 4 not yet visited nodes,* $\{4, 6, 7, 9\}$ *can be added to* c *without validating any constraints. For* $c = [0, 2, 5, 1, 9]$*, only 3 not yet visited nodes,* $\{3, 8, 10\}$ *comply with the constraints and are therefore allowed to be added to* c *as new child nodes. For all partial solutions in this layer, the scoring policy is calculated, and the 2 with the best score with respect to the scoring policy* P *are selected, namely* $c = [0, 2, 5, 1, 9, 3]$ *and* $c = [0, 2, 5, 1, 9, 10]$*. Only these two are explored further, and all other search tree nodes are disregarded. The beam search continues, expanding and selecting the best paths iteratively, ensuring that at each step, only the top* b *solutions are kept for further exploration, until all nodes have been visited.*

Computational experiments for this approach show that we can produce solutions efficiently with a high quality in terms of optimality gap with respect to the best known solution for smaller instances of 20 nodes with beam widths up to 50000 [Dor23]. For medium instances with 50 nodes and large instances with 100 nodes, the results show that this tree search has its weaknesses. Sometimes it does not succeed in finding the right paths in the search space to obtain the best solutions, as the size of the search space grows exponentially with the problem size.

1.2.2 Monte Carlo Tree Search

To overcome the limitations of the beam search, we develop a more sophisticated tree search that utilizes the edge probabilities of a neural network to guide the search through a search tree and uses the developed beam search as an indicator of how promising partial solutions are, which is subsequently also used to guide its search. This approach, the graph convolutional neural network assisted Monte Carlo tree search [DK24], also constructs a solution iteratively.

At each iteration, one node is added to the partially constructed solution. Within each iteration, a thorough exploration of the search space is done to investigate which node is the most promising to add. Each iteration consists of a predefined number of so-called rollouts, and each rollout consists of the following phases: *Selection, Expansion, Simulation* and *Backpropagation.*

Starting from the root node in iteration i, which holds the partial solution constructed in the first $i-1$ iterations, similar to the beam search each search tree node represents adding one node to the partial solution of its parent node. In the Selection phase, we iteratively select a child node from the currently selected node until we reach a leaf node. The selection criterium is a variant of the *Upper Confidence Bound applied to Trees* algorithm, which is suited to balance exploration and exploitation of the search tree. It takes into account the best (shortest) solution $q_{\min}$ and worst (longest) solution $q_{\max}$ found within the subtree

of this search tree node, the number of times n this part of the tree and of the parent node n_{parent} was already explored as well as the edge probability p_{ij} assigned by our graph convolutional neural network. We select the child node that minimizes:

$$\min \frac{q - q_{\min}}{q_{\max} - q_{\min}} - c_{\text{puct}} \cdot p_{ij} \cdot \sqrt{\frac{n_{\text{parent}}}{n}}, \tag{1.1}$$

with q being the simulated tour travel time at this node and c_{puct} a manually chosen parameter to weigh the tradeoff between the two terms. This approach ensures that while more promising areas of the search tree are investigated more deeply, other parts of the search tree are not entirely neglected, which helps mitigate the main drawback of the beam search - focusing too narrowly on one part of the search tree.

For better predictions, we develop a new graph convolutional neural network, which adapts the neural network depicted in Figure 1.1 and complements it with more inputs, namely the already partially constructed solution as well as the context of this partial solution, consisting of the geographical point where the vehicle is at the end of the partial solution together with its current time and capacity. This neural network is evaluated at the start of each iteration, i.e. after we choose the next node to add to the partial solution. The updated edge probabilities are then incorporated in the next iteration.

After a leaf node is selected, in the Expansion phase we add child nodes to this node for every node that is eligible to be added to the partial solution of the leaf node, similar to the expansion in the beam search. For each of the added child nodes, a simulation is done estimating the travel time of the complete solution if started with the given partial solution of this node. For this, we utilize the beam search from Chapter 2, which is well-suited for efficiently estimating the travel time of the entire tour and identifying promising partial solutions. The weakness of the beam search, not always finding the optimal solutions, does not impact our approach in this context, as these simulated values are solely used to navigate the search in the search tree.

After simulating the travel times of all newly added child nodes, the Backpropagation phase updates the values stored at each node, consisting of the best and worst simulated solutions in the respective subtree and a count of how often this node was explored. These values are then backpropagated up to the root node. Then, the next rollout is initiated. Once the specified number of rollouts is reached, the child node of the root node with the best solution found in its subtree is selected and initialized as the root node for the next iteration.

The following example shows the steps of one rollout within one iteration of the algorithm.

Example 1.2. *Assume the search tree of our algorithm is at the point as depicted in Figure 1.4.*

First, in the Selection phase in Figure 1.5, we select nodes starting from the root node by criterium (1.1) until we reach a leaf node.

We then proceed in Figure 1.6 to expand this search tree node by adding a child node for every not yet visited vertex that is feasible to add to the partial solution $[0, 2, 5, 7]$. *In*

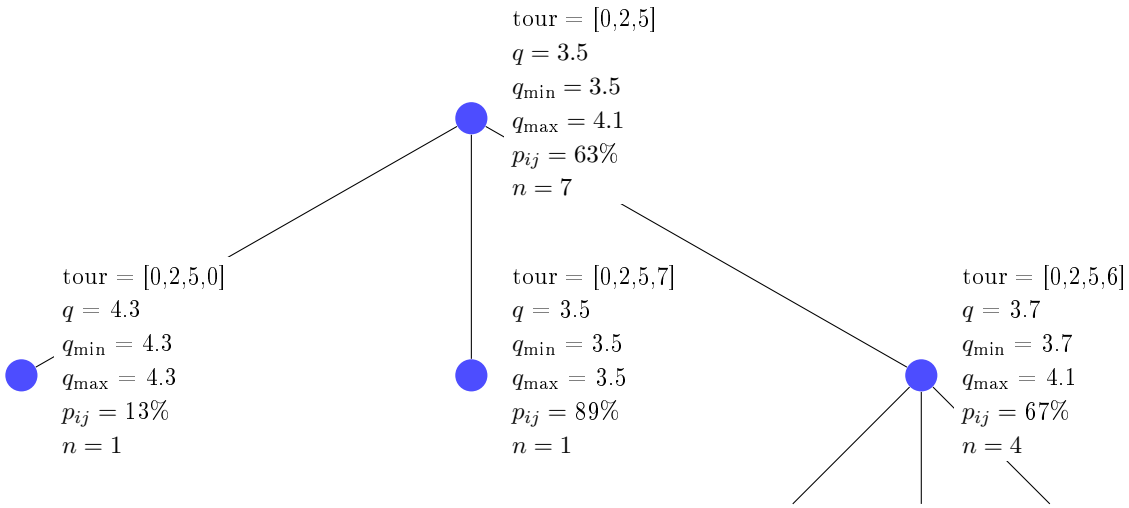


Figure 1.4: MCTS Example Current State

Figure 1.7, all child nodes are simulated.

Next, in Figure 1.8 we backpropagate the updated values up to the root node.

The Monte Carlo Tree Search algorithm allows us to efficiently search the search tree guided by our context-complemented graph convolutional neural network while balancing the exploration of the search space and the exploitation of the most promising areas of the search tree. Computational results show that we can outperform commercial solvers in terms of solution quality and runtime. Furthermore, the algorithm is suitable for being adapted easily. Instead of applying our beam search heuristic in the Simulation phase, we can also apply other heuristics. With the state-of-the-art heuristic LKH [Hel17] we are able to find even better solutions for large instances at the expense of runtime. However, there are also some disadvantages to our approach. The runtime scales significantly with the size of the problem instances, making the application of this method for large instances very time-consuming.

1.2.3 Quantum-inspired Computing

To develop an approach that is both computationally efficient, produces high-quality solutions and is scalable to large instances, in Chapter 4 we explore how we can approach the CVRPTW with new hardware technology. Quantum computers offer great potential to solve combinatorial optimization problems very quickly. The optimization problems are formulated as quadratic unconstrained binary optimization (QUBO) problems, which are currently the most widely used formulation in quantum computing among other things due to their equivalence to the Ising model [GKD22]. An introduction to quadratic unconstrained binary optimization can be found in Section 1.3.2.

For this purpose, in Chapter 4.1 we first develop a formulation of the CVRPTW as a QUBO. This poses particular challenges in the formulation of the constraints because

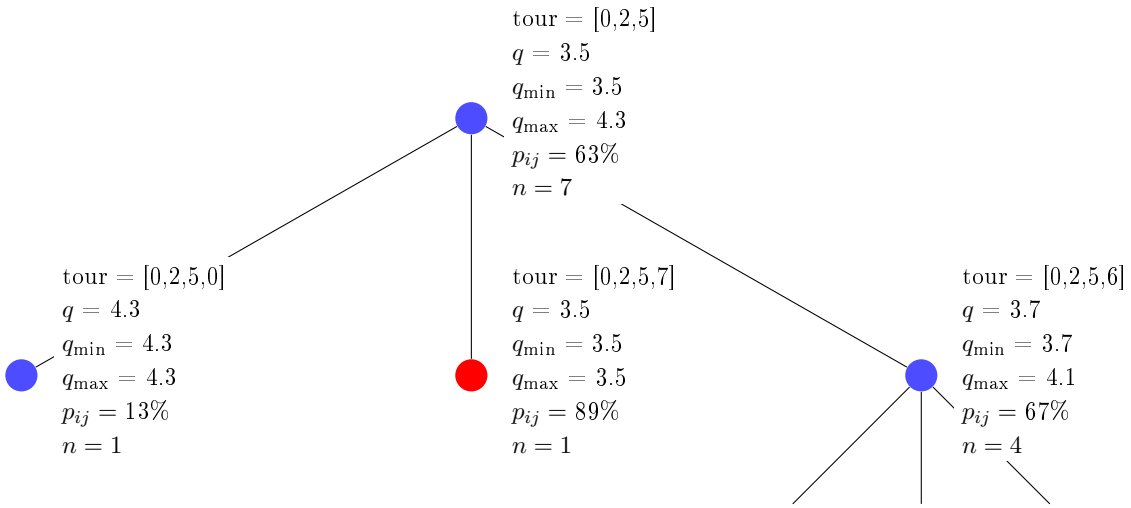


Figure 1.5: Selection Phase

both the capacity constraints and the time window constraints require integer variables and inequalities to be modeled. We discuss this in more detail in Section 1.3.1. Both are concepts that do not exist in QUBOs. This means that in order to solve the CVRPTW directly as a QUBO, we have to define binary representations for the integer variables and convert the inequalities into equations using slack variables, which also have to be converted from integer to binary. This leads to a large number of variables that are included in the QUBO representation of a CVRPTW instance.

We then proceed to solve this QUBO with the help of specialized hardware. We use the Digital Annealer (DA), which is a hardware by Fujitsu specialized to solve fully connected quadratic unconstrained binary optimization problems [MTM+20]. With the DA, Fujitsu is trying to close the gap between classical computers and quantum computers, the latter of which promise great increases in performance, but are still at an early stage in their development and realization and cannot be used for such complicated optimization problems as we are considering here. A detailed description of the DA can be found in Section 1.3.3.

Computational experiments show that an effective solution of the CVRPTW as a QUBO on the DA is difficult as the size of the QUBO representation reaches the limit of what is currently possible to solve with the DA. To overcome this difficulty, we use the graph convolutional network from Figure 1.1 as a so-called *learned problem reduction*. This means that we use the edge probabilities produced by the network to exclude potentially irrelevant edges from the graph and thus significantly reduce the problem size of the QUBO. We apply different thresholds for the edge probabilities to include edges in the problem, and compare the results when we then let the DA solve the reduced QUBO. It turns out that this leads to performance increases for smaller problem sizes, but for medium and large instances the DA still reaches its limits. This is because the CVRPTW with all its integer variables and inequalities is so complex that even a reduced representation as a

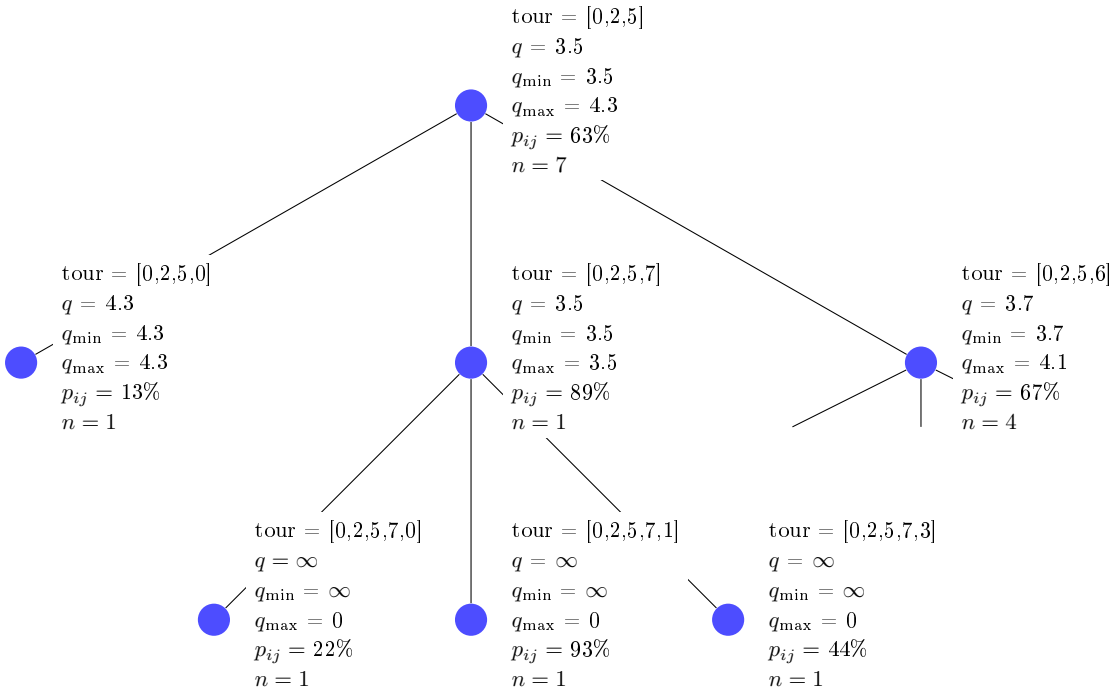


Figure 1.6: Expansion Phase

QUBO is quickly too large for the current hardware.

To address this, we focus on the strengths of the DA: solving optimization problems with only binary variables and without inequality constraints. In Section 4.2, we present a heuristic that integrates the DA's capabilities with our prior methods, achieving both scalability and high-quality solutions. This approach circumvents the problems of formulating all side constraints in a QUBO but rather outsources the handling of the side constraints to a heuristic.

For this, we develop a heuristic that we divide into three different phases. In the first phase, we use the DA to divide our problem instance into smaller subproblems. We do this by formulating a cluster problem that splits the node set into a given number of k subsets. This cluster problem can be easily formulated as a QUBO. Let the binary variable x_{im} be equal to 1 if and only if node i is assigned to cluster m. The QUBO of the cluster problem then has the following form:

$$\min \sum_{m=1}^{k} \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{im} x_{jm} + \left(\sum_{i=1}^{n} \left(\sum_{m=1}^{k} x_{im} - 1 \right)^2 \right). \tag{1.2}$$

The weighting c_{ij} for each edge is crucial for this problem to produce accurate clusters. We introduce the concept of path probabilities, which define the c_{ij} in this optimization problem. For this, we apply the graph convolutional network to output the edge probabilities p_{ij} and define

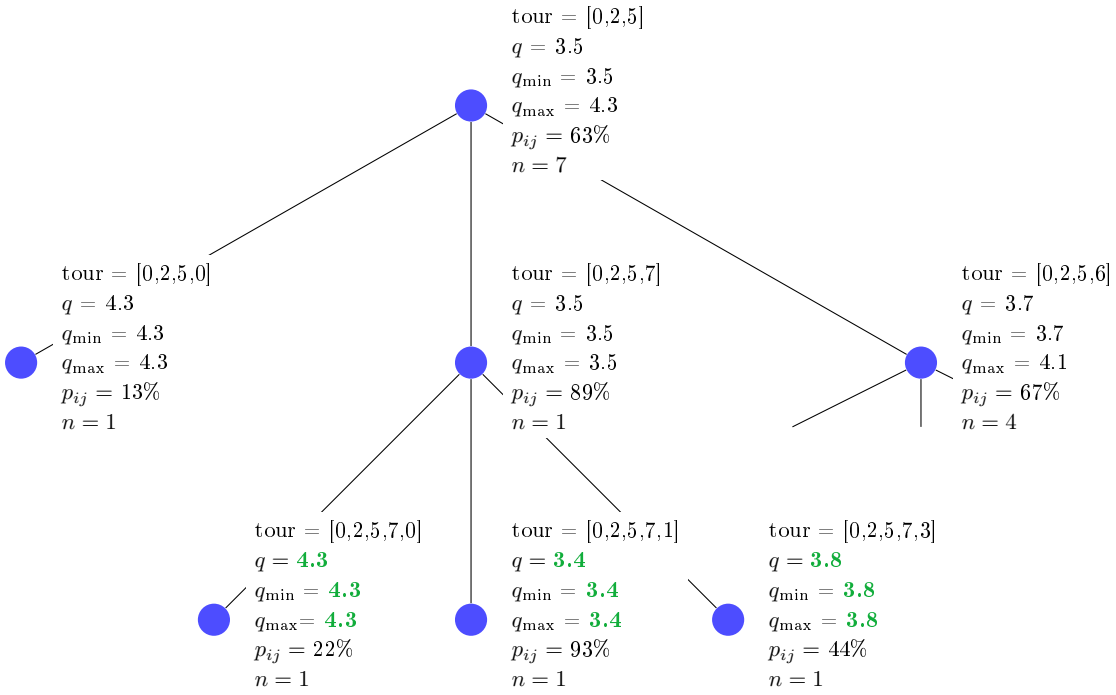


Figure 1.7: Simulation Phase

$$p_{ij}^{\text{path}} := \max \left\{ \prod_{(v,w)\in P} p_{vw} : P \text{ is } i,j-\text{path} \right\}.$$

Then using $(1 - p_{ij}^{\text{path}})$ instead of c_{ij} in the QUBO formulation (1.2), we maximize the probability that nodes within the same cluster are served by the same vehicle in the final solution.

Next, we apply the beam search to generate candidate routes for each of the clusters. The beam search is very useful for this task as we are only interested in generating a set of routes for each cluster that will be used to construct our solution in the next step. Therefore, it is advantageous to generate not only one possible solution per cluster but several different routes. The beam search can do this in parallel and has proven to find near-optimal solutions for smaller instances, making it well-suited for the small clusters we generated in the first phase.

For the last phase, we again utilize the DA to solve a QUBO that chooses a subset from the set of candidate routes R generated in the second phase, such that every node is included in exactly one route, therefore building a complete solution for our problem instance. For this, let y_r be the binary decision variable that is equal to one if and only if candidate route $r \in R$ is selected for the final solution, let δ_{vr} be a coefficient that is one if node v is included in route r and zero otherwise and let c_r be the travel time of route r. With this, we formulate a set partition problem as follows:

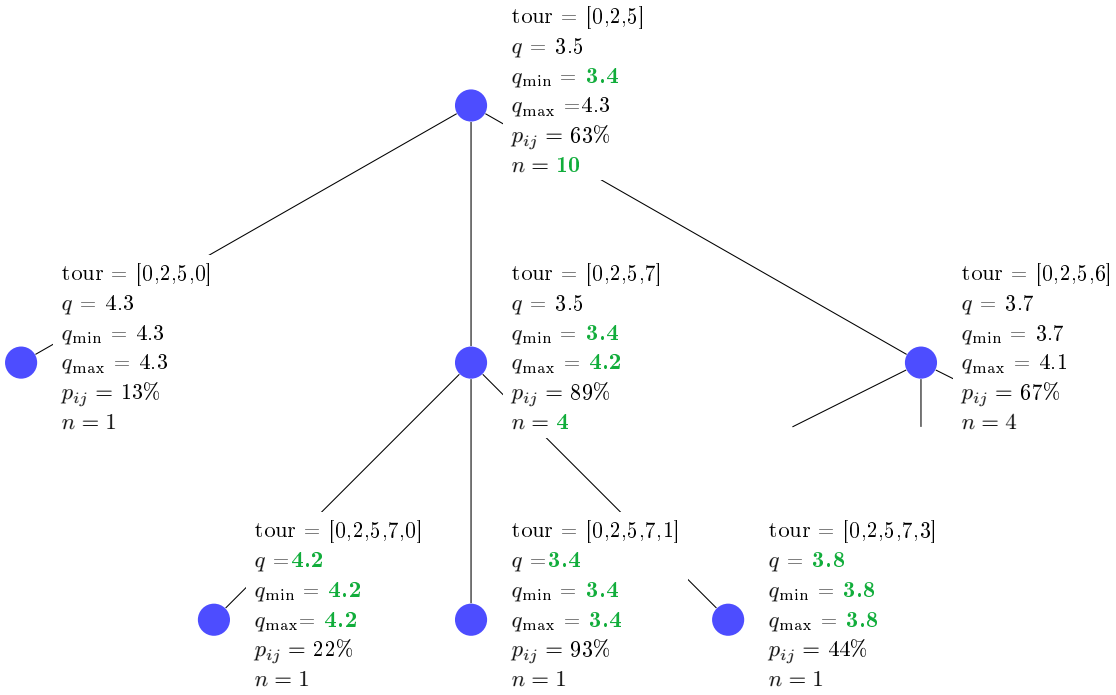


Figure 1.8: Backpropagation Phase

$$\min \quad \sum_{r \in R} c_r y_r + \left(\sum_{v \in V}^{n} \left(\sum_{r \in R} \delta_{vr} y_r - 1 \right)^2 \right). \tag{1.3}$$

A solution to this QUBO is equivalent to a feasible solution of our instance, as every node is visited and all routes fulfill the side constraints, as guaranteed by our beam search. The optimal solution minimizing this QUBO therefore corresponds to the best solution of the instance given the candidate route set R.

The big advantage of this 3-phase heuristic is that we use the strengths of the DA to solve QUBOs with a smaller number of variables without the need to handle integer and slack variables. In the cluster QUBO (1.2), the number of variables is equal to the number of nodes times the number of clusters, while the set partition QUBO (1.3) contains a number of variables equal to the number of candidate routes generated in the second phase, which we can control by the choice of the beam width. Computational experiments show that this 3-phase heuristic is both time efficient and provides optimal solutions to smaller instances while offering better scalability to larger instances than the previous approaches. This shows what potential new hardware technologies and the development of new deep learning models offer for efficiently solving complex combinatorial optimization problems in the future. Research into the combination of new technologies and new developments from different areas is an important future research path.

Before discussing all the approaches described above in detail in the following chapters,

we first define our tools and notation and provide a literature review in the following sections.

1.3 Tools and Notation

In the following section, we formally define the CVRPTW and introduce the notation used throughout this thesis. Additionally, we will present the tools employed in the subsequent chapters.

1.3.1 Problem Definition

A CVRPTW instance is given as a directed complete graph $G = (V, E)$ with $n + 1$ nodes, where node 0 is the special depot node. The cost for edge $e = (i, j)$ is given by c_{ij} and represents the transition costs from node i to j. Besides its coordinates, each node has additional attributes, namely a demand and a time window, imposing conditional constraints on the nodes. Therefore we can define a problem instance of the CVRPTW to consist of:

- $X = \{x_1, \ldots, x_n\}$, where $x_i \in [0, 1]^2$ are the coordinates of node i in the two-dimensional unit square.
- The location of the depot, given as $x_0 \in [0, 1]^2$.
- The demands at each node $i \in [n]$, given as $D = \{d_1, \ldots, d_n\}$.
- $T = \{[a_0, b_0], [a_1, b_1], \ldots, [a_n, b_n]\}$, where $[a_i, b_i]$ are the time windows for each node $i \in [n]$ and $[a_0, b_0]$ represents the planning horizon regarding earliest possible departure from and latest possible return to the depot.
- The capacity Z of the vehicles.

Moreover each node $i \in V$ requires a specific service duration h_i. Given a fleet of K vehicles, the goal is to find up to K routes r_k, $k \in [K]$, such that all nodes are visited and the capacity and time window constraints are met. A tour r_k is a sequence of nodes, starting and ending at the depot node 0, representing the order in which vehicle k visits the nodes. A set of tours $R = \{r_1, \ldots, r_K\}$ is considered a solution, if

$$\begin{aligned} &\cup_{k \in [K]} r_k = V, \\ &r_{k_1} \cap r_{k_2} = \{0\} \quad \forall k_1, k_2 \in [K], k_1 \neq k_2 \end{aligned}$$

and all tours satisfy the capacity and time window constraints. The capacity constraint is given as $\sum_{i \in r_k} d_i \leq Z$ and the time window constraint states, that the time service starts at node i, s_i, has to satisfy $a_i \leq s_i \leq b_i$. An arrival at node i before a_i is considered valid, the vehicle then has to wait until a_i to start the service.

The components of the instances contain a few assumptions. We assume a homogenous fleet of K vehicles so that the capacity and travel time are equal for all vehicles. Furthermore, the edge weights c_{ij} represent the transit costs from node i to node j and without loss of generality include the service durations h_i of node i. To simplify the notation, in the remainder of this thesis we assume that the transition cost c_{ij} is equal to the travel time it takes to get from node i to j and refer to our objective as minimizing travel times or equivalently the overall tour length or cost. To model this separately, we could introduce a second parameter representing travel times to model the time window constraints.

There are different approaches to formulating the objective function. The classical objective is to minimize the total time traveled over all vehicles. But there are more advanced formulations of the objective, for example, [FST20] take a more holistic perspective by also including the waiting times into the objective, searching for a good trade-off between total travel time, waiting times and number of vehicles. The main objective can also be defined to first minimize the number of vehicles, including the total travel time just as a secondary ([KLMS05], [FST20]). Other objectives include minimizing the total time traveled while using all K vehicles [ABH13]. In this work, we focus on the classical approach by minimizing the total travel time.

We can formulate the CVRPTW as a Mixed Integer Linear Program (MILP), which can be used to solve the problem by general-purpose MILP solvers such as Gurobi [GO22]. There are various ways to formulate the CVRPTW as a MILP, which differ in how the decision variables are defined. For the CVPRTW, there are 2-index and 3-index formulations that refer to the number of different indices for the decision variables [KLMS05]. In the following, we briefly present both formulations.

Following [KLMS05], for the 2-index formulation, let the decision variables x_{ij} for $i, j \in V$, with $i \neq j$, be defined as

$$x_{ij} = \begin{cases} 1, & \text{if edge } (i,j) \text{ is used by a vehicle} \\ 0, & \text{else.} \end{cases} \tag{1.4}$$

Note that in order to minimize the number of decision variables needed, the vehicle that uses this edge is not specified with the decision variable. To model the time window constraints, let the variable s_i represent the time at which the vehicle arrives at node i and let y_i be the available capacity of the vehicle after visiting node $i \in [n]$. The 2-index-MILP for the CVRPTW can be formulated as follows [KLMS05]:

$$\min \quad \sum_{i,j=0}^{n} c_{ij} x_{ij} \qquad \text{s.t.} \tag{1.5}$$

$$\sum_{i=0}^{n} x_{ij} = 1 \qquad \forall j \in [n] \tag{1.6}$$

$$\sum_{i=1}^{n} x_{0i} - \sum_{j=1}^{n} x_{j0} = 0 \tag{1.7}$$

$$\sum_{i=0}^{n} x_{ih} - \sum_{j=0}^{n} x_{hj} = 0 \qquad \forall h \in V \tag{1.8}$$

$$y_j \leq y_i - d_j x_{ij} + Z(1 - x_{ij}) \qquad \forall i, j \in V, i \neq j \tag{1.9}$$

$$0 \leq y_i \leq Z \qquad \forall i \in V \tag{1.10}$$

$$s_j \geq s_i + c_{ij} x_{ij} - M(1 - x_{ij}) \qquad \forall i, j \in V, i \neq j \tag{1.11}$$

$$a_i \leq s_i \leq b_i \qquad \forall i \in V \tag{1.12}$$

$$\sum_{j=1}^{n} x_{0j} \leq K \tag{1.13}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in V, i \neq j \tag{1.14}$$

$$s_i, y_i \in \mathbb{Z}_+ \qquad \forall i \in [n], \tag{1.15}$$

where (1.6) are the assignment constraints requiring each customer to be served by exactly one vehicle (note that the depot is excepted), (1.7) and (1.8) are the flow constraints that guarantee that the number of vehicles entering a node is equal to the number of vehicles leaving this node. Constraints (1.9) and (1.10) are the capacity constraints, where (1.9) guarantees the demands at each node are loaded and (1.10) restricts the maximal load to the capacity of the vehicle. Constraint (1.11) linearizes the conditional statement that, if edge (i, j) is used, then the arrival time at node j is at least the arrival time at node i plus the travel time to get from i to j. The constant M can be set to $\max_{i,j}\{b_i + c_{ij} - a_j\}$ [KLMS05]. Constraint (1.12) guarantees the arrival time to be within the time window, while Constraint (1.13) sets the maximum number of vehicles to be used. Finally, (1.14) and (1.15) are the binary and integer constraints.

A more natural way to formulate the CVRPTW as a MILP is to define the decision variables with three indices as follows:

$$x_{ijk} = \begin{cases} 1 & \text{if vehicle } k \text{ uses edge } (i, j) \text{ in the tour} \\ 0 & \text{else.} \end{cases}$$

With the decision variables being structured by also having index k for specific vehicles, one can keep track of the capacity constraints by simply adding the inequality

$$\sum_{i \in V} d_i \sum_{j \in V} x_{ijk} \leq Z \tag{1.16}$$

for each vehicle $k \in [K]$, which eliminates the need for the additional integer variables y_i in the 2-index-formulation. Let the variable s_{ik} be the time at which vehicle k arrives at node i. In the case vehicle k does not serve node i, the variable s_{ik} has no meaning. Following [KLMS05], the 3-index-MILP formulation for the CVRPTW is then given as:

$$\min \quad \sum_{k \in K} \sum_{i,j=0}^{n} c_{ij} x_{ijk} \qquad \text{s.t.} \tag{1.17}$$

$$\sum_{k\in K}\sum_{i=0}^{n} x_{ijk} = 1 \qquad \forall j \in [n] \tag{1.18}$$

$$\sum_{j=1}^{n} x_{0jk} \leq 1 \qquad \forall k \in K \tag{1.19}$$

$$\sum_{i=0}^{n} x_{ihk} - \sum_{j=0}^{n} x_{hjk} = 0 \qquad \forall h \in V, \forall k \in K \tag{1.20}$$

$$\sum_{i\in[n]} d_i \sum_{j\in V} x_{ijk} \leq Z \qquad \forall k \in K \tag{1.21}$$

$$a_i \leq s_{ik} \leq b_i \qquad \forall i \in [n], \forall k \in K \tag{1.22}$$

$$s_{ik} + c_{ij} - M(1 - x_{ijk}) \leq s_{jk} \qquad \forall i, j \in V, i \neq j, \forall k \in K \tag{1.23}$$

$$x_{ijk} \in \{0, 1\} \qquad \forall i, j \in V, i \neq j, \forall k \in K \tag{1.24}$$

$$s_{ik} \in \mathbb{Z}_+ \qquad \forall i \in V, \forall k \in K \tag{1.25}$$

where (1.18) are the assignment constraints requiring each customer to be served by exactly one vehicle (note that the depot is excepted), (1.19) ensures each vehicle leaves the depot at most once, (1.20) is the flow constraint and (1.21) is the capacity constraint. Inequalities (1.22) define the upper and lower bounds for the variable s_{ik}. Constraint (1.23) linearizes the condition that, if edge (i, j) is used, then the arrival time at node j is at least the arrival time at node i plus the travel time to get from i to j with the constant M chosen as before. Again, (1.24) and (1.25) are the binary and integer constraints.

By using the 3-index formulation, the number of inequalities needed to model the CVRPTW decreases significantly. Specifically, Constraint (1.21) only adds K inequalities, instead of the n^2 inequalities required for our formulation of the capacity Constraint (1.9). On the other hand, the number of decision variables x in the 3-index formulation increases to n^2K compared to the n^2 decision variables needed in the 2-index formulation. Depending on which method is chosen to solve the problem, this may or may not be advantageous.

In this thesis, we will focus on the classical 2-index formulation (1.5) as detailed above, but the CVRPTW is predestined to model other real-life optimization problems as well. Additional constraints or properties can be added to the problem so that even more complex problems can be modeled properly. This includes the consideration of heterogeneous fleets in which each vehicle has individual properties [GLMT99]. In the multi-depot variant, customers are served from several depots, each of which has its own fleet [LSL90]. In the Vehicle Routing Problem with Pick-up and Delivery, each vehicle has to bring one item to the customer and pick up another [BCGL07]. In the Vehicle Routing Problem with Soft Time Windows, arrivals outside the given time windows are also allowed, but these are then penalized, where a non-decreasing penalty function is given [KPS92]. For a detailed overview of all variations of the CVRPTW, we refer to [ES10].

1.3.2 Quadratic Unconstrained Binary Optimization

Oftentimes, optimization problems can be formulated as finding the minimum of a function that models problem $\mathcal{P}$. The global minimum of the function represents the optimal solution of $\mathcal{P}$. Quadratic unconstrained binary optimization (QUBO) aims at formulating optimization problems as quadratic polynomials, where the decision variables are binary. In detail, a QUBO problem is of the form

$$\min_{x \in \{0,1\}^n} x^T Q x, \tag{1.26}$$

where x is our decision vector containing the binary decision variables and Q is a square upper-triangular matrix taking values in the reals. The goal is to find the vector x^* that minimizes (1.26). This general form includes quadratic as well as linear objective functions if Q is a diagonal matrix and one notices that $x_i^2 = x_i$ for $x_i \in \{0, 1\}$. A more comprehensive introduction to QUBO can be found in [GKD22].

In general, we are given an MILP problem of the form

$$\begin{aligned} \min \quad & \sum_{i=1}^{k} c_i z_i \qquad \text{s.t.} \\ & \sum_{i=1}^{k} a_i z_i = b \\ & z_i \in \{0, 1\}, \end{aligned} \tag{1.27}$$

where $a_i, b \in \mathbb{R}$, or equivalently in matrix notation

$$\begin{aligned} \min\, & y = z^T C z \\ & Az = b \\ & z_i \in \{0, 1\} \end{aligned} \tag{1.28}$$

with C being a diagonal matrix and again $z_i^2 = z_i$ for $z_i \in \{0, 1\}$. We can then obtain a QUBO formulation by transforming the equality constraints into the objective function:

$$\min \quad \sum_{i=1}^{k} c_i z_i^2 + P(\sum_{i=1}^{k} a_i z_i - b)^2, \tag{1.29}$$

where the scalar $P \in \mathbb{R}_{\geq 0}$ is a factor for the penalty term that has to be set to weigh the constraints [GKD22]. This means that violating the constraint would lead to a greater increase in the objective value than violating it could reduce the value of the actual objective function. Setting an appropriate penalty value is crucial for the success of the solution process. A penalty value that is too large can hinder the solution process by causing the penalty terms to dominate the original objective function. This dominance disguises the differences in solution quality, making it challenging to distinguish between potential solutions. Conversely, a penalty value that is too small compromises the search for feasible

solutions, as the penalties may not adequately enforce the constraints [GKD22]. In practice, there tends to be a range of penalty values that strike the right balance. Within this region, the penalty is neither too large nor too small, allowing the algorithm to effectively explore the solution space while maintaining feasibility and optimizing the objective function. Identifying this region often involves domain knowledge, empirical testing or adaptive techniques that adjust the penalty dynamically during the optimization process [GKD22].

This way, we obtain a new function that is equal to the original MILP if and only if the binary decision variables z_i fulfill the equality constraint. To obtain the matrix Q from (1.26), we just have to transform the matrix form as follows:

$$\begin{aligned} y &= z^T C z + P(Az - b)^T (Az - b) \\ &= z^T C z + z^T D z + k \\ &= z^T Q z + k, \end{aligned} \tag{1.30}$$

where the matrix D and the constant k are obtained by mulitplicating. Since the constant is to be neglected during optimization, this is equivalent to (1.26). In the remainder of this thesis, for readability, we will typically refer to formulation (1.29) when discussing QUBO formulations.

Now, assume we are given a MILP problem, where the variables z_i are not binary, but rather integer variables with bounds $z_i^\ell \leq z_i \leq z_i^u$. Since a QUBO problem is only able to handle binary variables, we have to convert the integer variables to binary by replacing each variable z with its binary expansion [SGM22]:

$$B(z, z^\ell, z^u) := z^\ell + \sum_{j=0}^{k_z-2} 2^j x_{z,j} + \left(z^u - z^\ell - \sum_{j=0}^{k_z-2} 2^j \right) x_{z,k_z-1}, \tag{1.31}$$

where $k_z := \lceil \log_2(z^u - z_l + 1) \rceil$ and $x_{z,j}$ are new binary variables.

If linear inequality constraints are given in the form

$$\sum_{i=1}^{k} a_i z_i \leq b,$$

they must be converted into equality constraints by adding slack variables λ to derive

$$\sum_{i=1}^{k} a_i z_i + \lambda = b.$$

These additional integer slack variables also have to be optimized, resulting in a larger representation of the original problem after applying the binary expansion (1.31). To bound the number of additional variables needed, we can define sharp upper and lower limits for the value the slack variables can take. Generally speaking, it holds that

$$0 \leq \lambda \leq -\left(\sum_{i=1}^{k} \min(a_i z_i^\ell, a_i z_i^u) - b \right), \tag{1.32}$$

but problem-specific knowledge oftentimes allows us to find sharper bounds.

1.3.3 Fujitsu's Digital Annealer

We will utilize Fujitsu's Digital Annealer (DA) [MTM+20] to solve different QUBO formulations that we will develop in the following chapters. The DA is a product developed by Fujitsu to fill in the performance gap between classical computers, which hit their limit rather quickly when solving larger QUBO formulations, and quantum computers, which are still in their experimental stage. The DA is a hardware accelerator specifically designed to solve fully connected QUBO problems, which are problems where all pairs (x_i, x_j) have a non-zero coefficient Q_{ij} in the QUBO formulation (1.26). This is equivalent to saying the problem's graph is a complete graph, where every node (variable) is directly connected to every other node. The DA internally uses a modified version of the Markov Chain Monte Carlo (MCMC) method. A major advantage of the DA hardware is its massively parallel implementation and an innovative sampling technique. The hardware can take into account all bits simultaneously in each step before deciding which bit to flip. Each MCMC step takes the same amount of time, regardless of whether a bit flip is accepted or not. In addition, the computational effort for updating the effective fields, which determine the energy difference caused by a bit flip and thus the probability of this bit being flipped, remains constant. In addition, the DA supports parallel tempering. Parallel tempering is a MCMC technique designed to enhance the exploration of a system's state space, particularly for systems with complex energy landscapes. It helps overcome challenges such as getting trapped in local minima, which are common in high-dimensional or rugged energy landscapes. The method involves running multiple copies of the system simultaneously [KAHL22].

In its 3rd generation, the DA can handle optimization problems with up to 100000 decision variables, a major improvement from the 8192 supported decision variables in its 2nd generation. This is done by joining multiple Digital Annealer Units (DAU) together. These DAUs are dedicated processors executing the minimization algorithm parallel tempering. Using additional software, the large QUBO formulation is decomposed into smaller QUBO formulations, which are then minimized on one or multiple DAUs, depending on the problem size.

Another major improvement from its 2nd to 3rd generation is the ability to handle inequalities. As previously explained, formulating inequality constraints as binary quadratic polynomials is connected to introducing new variables for each inequality, as one needs to convert those inequalities to equalities by adding slack variables. To limit the number of variables needed for the formulation, finding sharp upper and lower bounds for those slack variables is important. Adding those slack variables and subsequently converting them to binary variables adds a large number of decision variables to the formulation, which makes it harder to solve. When using the DA in its 3rd generation, one can declare the inequalities separately from the QUBO formulation and is, therefore, able to solve optimization problems in the form of binary quadratic programming problems (BQP). This means that inequality constraints are not converted to a penalty term and thus do not use any additional decision variables. A detailed experimental analysis of the size of the problem formulation with and without inequalities is done in Section 4.1. There is no sharp upper

bound for the number of inequalities the DA can handle, but it is limited to a number in the lower 6-digit range. It depends on the complexity of the BQP formulation to solve, which has to be examined experimentally.

The DA has additional features such as auto-scaling, which automatically scales the penalty factors in the energy function. For that, the objective term and penalty terms are initialized separately to allow the adjustment of the penalty coefficient P. Starting from a preset initial value, P is multiplied with a factor $\theta \in [1, 2]$ every time the objective value is not improved for a set number of iterations. With this approach, the weighting of the penalty term is automatically adjusted to fall within the appropriate range, as outlined in Section 1.3.2. This eliminates the need for domain-specific knowledge or empirical testing to determine the correct value. For a more detailed explanation including examples concerning the DA for combinatorial optimization, we refer to [SWMU19].

1.4 Related Literature

Over the years, a wide range of approaches have been proposed for solving the CVRPTW, including both exact algorithms and heuristics. These methods differ in their complexity and the quality of the solutions they produce. In recent years, there has been a growing interest in developing approximate algorithms for the CVRPTW, as these methods can scale to large instances and produce high-quality solutions in a reasonable time. In the following, we present the existing approaches to tackling the CVRPTW.

1.4.1 Exact Approaches

Exact solution methods for the CVRPTW can be divided into three categories according to [TV14]: Branch-Cut-and-Price, Branch-and-Cut, and reduced set partitioning. Most successful algorithms have been based on column generation, where the problem is decomposed into a restricted master problem that selects new routes from a subset of candidate routes and a pricing subproblem that generates new routes to be considered in the restricted master problem. In general, the pricing subproblem obtained is a shortest path problem with resource constraint (SPPRC) [ID05], which is NP-hard [Dro94].

To obtain better lower bounds in the Branch and Bound search tree different methods have been proposed. [KDM+99] introduce the Branch-Cut-and-Price (BCP) approach by adding valid inequalities dynamically to strengthen linear relaxations. Other families of inequalities were subsequently proposed over the years [JPSP08], [PPPU17], [PCDU17]. Other approaches aim at strengthening the pricing subproblem by using algorithms to generate new routes which include labeling algorithms [FDGG04], [ID05], [BDD06] and heuristics [FGR07], [DLH08], but solving the SPPRC exactly remains an obstacle. [BMR11] therefore develop a different relaxation approach for the pricing subproblem, the ng-path relaxation, where a set partitioning formulation is used in which routes are dynamically generated via column generation. Their algorithm can be seen as a three-step method, where calculated lower and upper bounds are used to enumerate a subset of all feasible

routes whose reduced cost with respect to the dual solution is less than or equal to the gap between the lower and upper bound. Afterward, the CVRPTW is formulated as a MILP, where the previously determined subset of feasible solutions is incorporated and solved using a commercial MILP solver.

Currently, the best exact solution method for vehicle routing problems in terms of execution time and the quality of the solutions found is a complex Branch-Cut-and Price (BCP) incorporating most of the key elements found in the best BCP approaches of recent years [CCD19]. That includes sophisticated labeling algorithms for pricing [SUP21], ng-path relaxation [BMR11], rounded capacity cuts [LLE04], non-robustness control [PPP$^+$17], rank-1 cuts with limited memory [JPSP08], [PPPU17], [PCDU17] [PPP$^+$17], [BPPU18], path enumeration [BCM07], [CM14] and hierarchical strong branching [Rø12], [PPP$^+$17], as detailed in [EQSU23]. The main methodological and modeling approaches for the BCP are surveyed in [CCD19]. We refer the reader to [CCD19] for extensive details on the BCP as well as to [BMR12], [DMR14] for an extensive overview of exact solution methods for the CVRPTW.

1.4.2 Heuristic Approaches

Given the complexity and computational costs of exactly solving the CVRPTW, heuristics aim at quickly finding relatively high-quality feasible solutions. Because of their speed and their ability to handle larger instances, heuristics are widely employed in industry, even though there is no guarantee regarding solution quality.

Heuristics for routing problems can be divided into two categories: construction heuristics and improvement heuristics [TV14]. Constructive methods can generate solutions within a set number of steps that is linearly dependent on the size of the problem, usually, customers are selected sequentially to create feasible solutions. However, limited information available about the other tours while constructing new tours node by node can lead to inefficient constructions, and obtaining additional information in the space of solutions that are constructed sequentially becomes expensive quickly. The first constructive heuristic for the CVRPTW is proposed by [BS86]. The heuristic starts with all customers served individually and in each iteration, it is calculated which two routes can be combined to maximize the savings. [Sol87] proposes a similar savings-based constructive heuristic. Time-based heuristics, where first the customers are sorted depending on their time windows and then inserted in a route, were proposed by [PR93], [Sol87], [Rus95].

Improvement approaches typically rely on an initial solution that they can then refine over time. However, it may be challenging to find a suitable starting solution for more complex problems like the CVRPTW. In fact, finding the first feasible solution for the CVRPTW with a fixed number of vehicles is an NP-hard problem on its own [Sav85]. Improvement heuristics are based on local search operators. Given a feasible solution, the neighborhood of this solution is defined as the set of feasible solutions that are reachable by small changes to the solution using the local search operator. For the CVRPTW, these operators usually are defined as interchanging customers, i.e. exchanging two customers within a route, customer removal of one route and insertion in another route as well as

combining partial routes. [BG05] provide an extensive overview of the different neighborhood structures that are used to tackle the CVRPTW. One potential danger with these approaches is being trapped in local minima if the local search criteria are too narrow to escape from this minimum. To circumvent this problem, the concept of Large Neighborhood Search is introduced [Sha98]. Destroy and repair operators are applied repeatedly to explore more complex neighborhoods, making it possible to find better candidate solutions in each iteration and therefore traversing a more promising search path. This has been the base of the most successful heuristics for the CVRPTW ([PR07], [PGDR09]). The Lin-Kernighan heuristic [LK73] is such a local search improvement heuristic for the symmetric TSP, which swaps segments from one route with segments from other routes. The applied measure of distance between two routes is the number of edges that are in one but not the other. New routes are created by reordering old routes, sometimes by reversing the direction in which they are traversed. It has been refined and improved over the years and is one of the best known heuristics for the symmetric TSP [Hel17]. Building on this, [Hel17] introduces the LKH heuristic, an extension of the Lin-Kernighan heuristic for constrained vehicle routing problems by transforming the problems into standard symmetric TSPs, and is one of the best available heuristics for solving routing problems with side constraints [KvHGW22].

Clustering-based heuristics aim at dividing the problem instance into smaller sub-problems such that a good solution can be obtained by combining the results of the individual sub-problems. According to [LS02], clustering-based approaches for vehicle routing problems can be categorized into two groups, namely *Cluster First, Route Second* and *Route First, Cluster Second*. *Cluster First, Route Second* translates to decomposing the original problem into smaller problems, where each smaller problem (cluster) is solved individually. Depending on the criteria of how the clusters are built, there are different approaches to achieve this. If the number of vehicles to be used is known, this is done by building a number of clusters equaling the number of vehicles and then solving a TSP instance within each cluster, as done in [FJ81]. Other approaches build so-called macro nodes from each cluster, find routes to visit all macro nodes, then deconstruct the macro nodes and modify the routes found to accommodate for the constraints [DC07]. *Route First, Cluster Second* builds solutions by starting with one route visiting all vertices while neglecting the side constraints. Afterward, this route is iteratively divided into smaller routes until all constraints are fulfilled.

For the Capacitated Vehicle Routing Problem with Time Windows, there exist only a limited number of approaches that cluster the set of customers while taking into account the time window constraints. Most methods focus on the spatial characteristics of the customer's features. The sweep-based heuristic ([GM74], [Sol87]) clusters the customers based on their polar coordinates based on a center of gravity, which in the context of vehicle routing is the depot. An imaginary ray originating from the depot is swept counterclockwise over all vertices. The demand of each swept vertex is accumulated. Once the accumulated demands reach the capacity for one vehicle, or if including the next vertex exceeds the capacity, the current nodes are included in a cluster. Building on the work of [ND86],

who developed a spatial partition technique assuming underlying density distributions, [Ouy07] uses a combination of a disk model and weighted centroidal Voronoi tessellation to obtain spatial cluster zones. [GN07] apply a spatial clustering heuristic for the VRPDP by dividing the vertex set into three categories and then solve a general assignment problem to allocate vehicles to the clusters taking into account capacity constraints.

[TV02] propose the so-called *popmusic* framework, where a precalculated solution is decomposed into p parts, then some of these parts are aggregated into a subproblem, which is subsequently tried to be improved by some optimizer. As an improvement in the subproblem corresponds to an improvement of the whole solution, if the optimizer finds an improvement the solution is refined. This process is repeated for different subproblems until no improvement is found. [BVH07] propose a randomized adaptive spatial decoupling scheme that iteratively selects random vehicle-based subproblems, which are solved independently by an optimization algorithm, and then reinserted into an existing solution, where the decouplings depend not only on the instance data but also on the current solution. Similar to this, [ODH08] and [ODH$^+$09] present a spatial decomposition approach based on the popmusic framework in [TV02], where iteratively routes are chosen and subproblems are created based on nearness to that route. The iteration stops if for all routes subproblems are created that do not lead to improvement. Building on [BVH07], in [BVH10] the authors propose a refined adaptive clustering heuristic that also incorporates the temporal features of the customers. [DC07] and [Pug14] propose heuristic approaches, where the time window constraints are incorporated in the heuristic clustering phase, such that the waiting time due to early arrivals is kept as small as possible, while also minimizing the average distance between vertices within the clusters. The resulting clusters then represent TSPTW instances, which are solved and sequenced to build a full solution. The three-phase structure of these approaches, i.e., clustering, solving each cluster independently, and then combining the cluster results, is similar to our proposed approach presented in Section 4.2, with significant differences in the way each phase is performed. [QLLM12] use time geography theory to represent time and space in the same coordination system defining a spatiotemporal distance, which is used to build clusters minimizing this distance. The temporal part of the distance between two vertices is based on the time at which the destination is reached, which is penalized when the arrival time falls outside the time window. Based on this time window violation, an additional cost is added to the objective function and a VRP with soft time windows is solved for each cluster. For the VRP with hard time windows, this approach is not suitable as the routes are not necessarily feasible.

We refer to [DMR14] for a detailed and comprehensive description of all heuristic approaches for the CVRPTW.

1.4.3 Machine Learning in Routing Problems

The idea of solving routing problems with artificial neural networks dates back approximately 40 years, but only recently has it been applied in ways that outperform traditional heuristic methods. One of the earliest neural network models designed to address rout-

ing problems, namely the TSP, was introduced by Hopfield and Tank [HT85]. The proposed Hopfield neural network, a fully connected model, minimizes an energy function that represents the TSP's objective function. Unlike modern neural networks, the connection weights in the Hopfield model are not learned but are predefined based on the problem data [Pot93]. The network evolves according to its operational equations, ultimately converging to a locally optimal solution. Although the Hopfield neural network theoretically offers near-optimal solutions for the TSP, it has significant limitations, including poor scalability and a tendency to become trapped in local minima that often fail to represent feasible solutions. Over the years, many variants have been proposed to improve the model, a comprehensive overview can be found in [Pot93].

Around the same time, two alternative neural network approaches for solving the TSP were introduced: the elastic net approach [DW87] and the self-organizing maps approach [Koh90]. Both methods differ significantly from the Hopfield neural network and are conceptually closely related. In both approaches, a ring is initially defined and then progressively adjusted until it passes close enough to each city to form a tour. However, the two methods differ in how they update the coordinates of the points on the ring [Pot93]. By the time, both approaches outperformed the Hopfield neural network, but the performance gap to the best heuristics at that time were still quite large. Research on artificial neural networks stalled due to limitations in computational power, insufficient data, and a preference for simpler models like support vector machines, decision trees, and Bayesian methods, which were easier to train with limited data. The field experienced a resurgence with the availability of powerful hardware, particularly GPUs and large datasets, and the development of improved techniques for training deep networks.

With the introduction of the Pointer Network architecture by [VFJ15] to solve the TSP, the combination of neural networks with heuristics to solve routing problems has seen an uprise in recent years. Similar to the heuristic approaches, some machine learning approaches for solving routing problems that utilize neural networks, such as those in [CT19] and [LZY20], focus on iteratively improving an existing solution, while others, such as those developed by [NOST18], [KvW19] and [FST20] generate a solution for routing problems by adding one node at a time.

The Pointer Network approach by [VFJ15] is based on a supervised learning approach, which is then adapted to reinforcement learning by [BPL$^+$16]. The first deep learning model for sequential solutions to the CVRP is presented by [NOST18], who adapt the Pointer Network, but make substantial modifications to the encoder model by replacing the recurrent neural network part of the encoder with a linear embedding layer with shared parameters. Reinforcement learning has been explored as a training strategy for constructive methods for routing problems, including TSP variants [KW18], [JLB19b], [KCK$^+$20], as well as the CVRP [KvW19], [PWZ20], [DAT20], [KCK$^+$20], achieving promising results. This approach has gained traction due to the inherent limitations of supervised learning methods, which heavily rely on access to extensive datasets containing high-quality solutions [BLP21]. These constructive approaches employ autoregressive methods, wherein the model is evaluated for each new node that is added to the tour.

Works that apply non-autoregressive methods consider producing a heatmap of promising edges at the beginning and building solutions from this without further evaluating the neural network. The first approach by [NVBB17] trains a graph neural network [SGT+09] in a supervised way to directly output an adjacency matrix and construct the solution by applying a tree search. Their approach performs poorly even on smaller instances. [JLB19a] build on the work of [NVBB17], but instead of using a graph neural network, they apply a deep graph convolutional neural network [BL17], to enable learning from larger training sets. With their approach, they outperform other learning-based techniques for the TSP. This result is not surprising, since supervised learning techniques generally perform better when sufficient training data are available. [KvHGW22] build on the idea of predicting promising edges in a non-autoregressive form to solve the TSP and CVRP, but use these predictions more efficiently by applying a potential function in a dynamic programming framework to exploit known problem structures in a controlled manner.

However, incorporating time window constraints is a difficult task, and there have been only a few recent proposals for constructive approaches to the CVRPTW that utilize deep learning. The first constructive method using deep learning for the CVRPTW is proposed by [FST20]. They use the attention model from [KvW19] for the CVRP, which constructs one route at a time by treating all not yet visited nodes as actions and learning a policy model to choose the next node via reinforcement learning. [FST20] extend this approach for the CVRPTW by constructing multiple routes simultaneously and using the information of all partially constructed routes to choose the next node. Although the computational results of this model are promising, it is computationally intensive to use. Furthermore, their main objective is not to minimize the total distance traveled, but a combination of total distance traveled, waiting times for the vehicles, and the number of vehicles used as well as also considering soft time windows, where violating time window constraints is penalized rather than forbidden, but through appropriate weighting the objective function could be adjusted to focus on one goal. [FTST22] extend the work of [FST20] by replacing the self-attention layers of their model with graph neural networks to encode the problem and propose a Large Neighborhood Search using a learned construction heuristic via reinforcement learning to re-construct partially destructed solutions in an autoregressive manner. A hierarchical reinforcement learning model based on pointer networks is proposed by [WSL21]. This model involves learning to obtain feasible solutions at a lower level and using these solutions as input for a second decoder to minimize the total distance. However, this approach is only effective for relatively small numbers of customers. Two improvement-based methods for the CVRPTW have been proposed recently. The first one by [GCC+20] uses an enhanced version of the graph attention network [VCC+18] to learn a heuristic for Very Large-scale Neighborhood Search that includes both improvement and destruction operators. They are able to approximately solve instances with up to 400 nodes, with respect to standard heuristics the improvements in terms of solution quality are at around 4-5 %. A Variable Neighbourhood Descent heuristic with tabular Q-learning that uses eight different neighborhood functions is proposed by [SdSSB19]. In their approach, the local search functions are learned in a reinforcement learning setting,

which makes use of the graph attention network in the encoder part of the network, while the decoder structure is based on the architecture of the pointer network [VFJ15] to render pairs of different destroy and repair operators.

The only literature to our knowledge that uses machine learning assistance within a clustering algorithm for the CVRPTW is [Pou20]. They develop an Optimal Classification Tree (OCT) Method [BD17] to predict the number of vehicles. A classification tree takes a set of features as input and outputs a label, which in this case corresponds to the number of vehicles needed to serve all customers. Each node in the tree can be viewed as a partition of the feature space according to a simple equation, where the fulfillment of this equation indicates which branch to follow until a leaf is reached. The leaf then corresponds to the output label. The trees are trained on generated training datasets, from which several features are extracted, such as the spatial stop distribution, the demand served in combination with the number of stops needed and the approximate routing distance, calculated by a simple heuristic. The OCT outperforms two other deep learning approaches, namely a convolutional neural network for image classification and a graph neural network for graph classification, in terms of accuracy, but exhibits similar accuracy as the naive $k-$means clustering algorithm [SRV18].

1.4.4 Quantum(-inspired) Computing

The utilization of specialized QUBO hardware for solving vehicle routing and similar problems is still in its early stages of exploration. Most of these approaches have a hybrid structure, as the physical limitations of specialized hardware require dividing the problem into smaller subproblems that can be handled. [TDR$^+$21] propose a hybrid quantum-classical tree search, where a classical processer maintains a global search tree and enforces constraints on relaxed sub-problems, and a quantum annealer is applied to obtain strong candidate solutions by sampling from the configuration space of the relaxed problem. [RVO$^+$14] investigate the effectiveness of a quantum annealer in solving small instances within families of hard operational planning problems. They explore various mappings to QUBO problems and embeddings. [SC17] present a systematic, simplified approach to tune a Quantum Annealing algorithm for vehicle schedule problems implemented on a classical computer. [HNN$^+$20] model the dynamic multi-depot CVRP as a QUBO problem and propose a solution approach to solving it on D-Waves quantum annealer. [FRG$^+$19] follow the *Cluster First, Route Second* approach to solve the CVRP using D-Waves quantum annealer. In their 2-phase heuristic, they divide the set of customers into clusters by formulating this as a QUBO Knapsack problem with additional distance minimization. Each cluster is then mapped to a TSP QUBO formulation and subsequently solved using the quantum annealer. In their work, [BGK$^+$20] develop a hybrid algorithm for solving the CVRP using D-Wave's Leap framework by dividing the problem into smaller subproblems using classical components which are then solved as QUBOs. [IWT$^+$19] develop a QUBO formulation for the CVRP that incorporates constraints with time, state and capacity and conduct experiments on D-Waves quantum annealer, but their formulation quickly exceeds the hardware limits regarding the number of quantum bits even for smaller instances. [SPL22] propose a

multi-start approach driven by a prediction model to use QUBO solvers more efficiently by improving the initial states adjustment method and testing their approach with Fujitsu's DA.

1.5 Organization

The remainder of the thesis is organized as follows.

In Chapter 2, we present a neural network-assisted tree search for the CVRPTW, in which we develop and train a Graph Convolutional Network for the CVRPTW, which is subsequently used to guide a limited-width-breadth-first tree search to iteratively construct feasible solutions. This is based on the published work [Dor23].

Chapter 3 is based on a work published jointly with Tobias Klein [DK24]. We present a Monte Carlo Tree Search approach for the CVRPTW, incorporating a Context Complemented Graph Convolutional Network as well as the heuristic approach developed in Chapter 2.

In Chapter 4 we then develop different methods to use Quadratic Unconstrained Binary Optimization to solve the CVRPTW. First, we apply a learned problem reduction to reduce the instance size and thus the QUBO representation matrix. This is based on the published work [Dor23]. Second, we develop a 3-phase heuristic that exploits the strengths of QUBOs, the binary optimization of fully connected problems, in combination with a heuristic to which the handling of constraints that are difficult to fit into the framework of binary unconstrained optimization are outsourced. This is based on the joint work with Salwa Shaglel, Martin Kliesch and Anusch Taraz [DSKT24]. For both approaches, we conduct experiments using Fujitsu's DA.

Chapter 2

Solving the CVRPTW via Graph Convolutional Neural Network Assisted Tree Search

In this chapter, which is based on the paper [Dor23], we propose a novel method for approximately solving the CVRPTW via a supervised deep learning-based tree search. The model uses a deep neural network to assist in finding solutions in a non-autoregressive manner by providing a probability distribution to guide a tree search, resulting in a deep learning-assisted heuristic. The model is built upon a new neural network architecture, called graph convolutional network, which is particularly suited for deep learning tasks.

For this, we develop the graph convolutional neural network for the CVRPTW in Section 2.1 and the neural network guided tree search in Section 2.2. In Section 2.3, we conduct computational experiments and discuss the results.

2.1 Graph Convolutional Network

For our model, we use a graph neural network called Residual Gated Graph ConvNet (GCN) [BL17], which is adapted for the TSP in [JLB19a]. They provide a framework to solve routing problems using the GCN architecture, however, they only adapt it to solve the TSP. In this section, we present our extension to address the CVRPTW, which relies on modifying the layers to adapt additional constraints within the framework and modifying the search method for constructing full solutions.

The neural network assigns a value to each edge in the graph to predict which edges are most promising to include in a solution. These associated edge values can be interpreted as probabilities, although they do not sum up to one and therefore do not possess all the characteristics of a proper probability measure. In the following, we describe each of the neural network's layers in detail. The high-level structure of the neural network is shown in Figure 1.1 in Chapter 1.

Input Layer

The input for the node features is five-dimensional. For node i we have the two-dimensional coordinates $x_i \in [0,1]^2$, the time window given as $[a_i, b_i]$ and the normalized demand d_i/Z, where we set $d_0 = 0$ for the depot. These features are concatenated to the five-dimensional input feature vector y_i and are then embedded to a $\frac{h}{2}$-dimensional representation, where h denotes the hidden dimension of our network that represents the number of parameters each hidden state within the network stores. Similar to [KvHGW22], the special depot node gets a separate learned initial embedding parameter in order to mark the depot node as special for the network. For that, define $\hat{y}_0 \in \{0,1\}^{n+1}$ to be the unit vector with entry one at the first position and zeros otherwise. This is put together as the node input feature as follows:

$$\alpha_i = A_1 y_i \oplus A_2 \hat{y}_0, \tag{2.1}$$

where $A_1 \in \mathbb{R}^{\frac{h}{2} \times 5}$, $A_2 \in \mathbb{R}^{\frac{h}{2} \times (n+1)}$ are the weight parameters that are learned during the training procedure and $\cdot \oplus \cdot$ is the concatenation operator.

For the input edge feature, the edge values c_{ij} are embedded as a $\frac{h}{2}$-dimensional feature vector. We do not integrate the K-nearest neighbor feature used in [JLB19a], since, in contrast to the TSP without time windows, the assumption that a node in the solution is usually connected to nodes in its close proximity does not necessarily hold with time window constraints. Instead, we use an indicator function δ_{ij} of an edge which has the value one for edges connecting nodes i and j, with $i \neq j$ and i, j not the depot, and value two for edges connecting nodes with itself. To tag the depot as a special node, the indicator function δ_{ij} furthermore has a value of 3 for edges to and from the depot and a value of 4 for the depot self-loop. Together, the input edge feature is given as:

$$\beta_{ij} = A_3 c_{ij} \oplus A_4 \delta_{ij}, \tag{2.2}$$

where $A_3 \in \mathbb{R}^{\frac{h}{2} \times 1}$ and $A_4 \in \mathbb{R}^{\frac{h}{2} \times 4}$ as above are weight parameters to be learned. As for the parameters A_2, we apply a separate embedding layer to learn the embedding parameters A_4 for our indicator function δ_{ij}. These $h/2-$dimensional edge and node feature representations are then concatenated to form the $h-$dimensional input for the first graph convolution layer and are subsequently passed through all graph convolution layers before producing our desired output within our classifier layers, which is described in the following.

Graph Convolution Layer

In each of the graph convolution layers the model updates the edge and node embeddings. We leverage the design of the Residual Gated Graph ConvNet developed in [BL17] by adding an edge feature representation. Let ℓ be the current layer and for node i and edge (i,j), let x_i^ℓ be the node features vector and e_{ij}^ℓ the edge features vector. We define the

features for layer $\ell + 1$ in the following way:

$$x_i^{\ell+1} = x_i^\ell + \text{ReLU}\left(\text{BN}\left(W_1^\ell x_i^\ell + \sum_{j \in N(i)} \eta_{ij}^\ell \odot W_2^\ell x_j^\ell\right)\right), \tag{2.3}$$

$$e_{ij}^{\ell+1} = e_{ij}^\ell + \text{ReLU}\left(\text{BN}\left(W_3^\ell e_{ij}^\ell + W_4^\ell x_i^\ell + W_5^\ell x_j^\ell\right)\right), \tag{2.4}$$

where $W_k^\ell \in \mathbb{R}^{h \times h}$ for $k \in [5]$ again are the weight parameters to learn, with the notation $[n] = \{1, \ldots, n\}$ for any positive integer n, σ is the sigmoid function, ε is a small value, ReLU being the rectified linear unit, BN stands for batch normalization, $N(i)$ denotes the neighborhood of node i, moreover $\cdot \odot \cdot$ denotes the Hadamard product operator and η_{ij}^ℓ being defined as

$$\eta_{ij}^\ell = \frac{\sigma(e_{ij}^\ell)}{\sum_{j' \in N(i)} \sigma(e_{ij'}^\ell) + \varepsilon}.$$

For the input layer, we set $x_i^0 = \alpha_i$ and $e_{ij}^0 = \beta_{ij}$. We implement W_5^ℓ as a separate parameter to allow the model to distinguish different directions of edges since in the context of CVRPTW we have directed edges in our solutions. The training labels for the edges are also set accordingly, meaning if edge (i, j) is contained in the solution, then edge (j, i) will have label zero, although edge (i, j) is labeled with a one. Batch normalization is a mechanism that normalizes layer inputs to reduce internal covariate shifts which allows the usage of higher learning rates and hence accelerates the learning of deep architectures [IS15].

MLP Classifier

A Multi-layer Perceptron (MLP), which is a fully connected feedforward neural network with a number ℓ_C of hidden layers, is used for generating the desired output, a finite measure that represents probabilities over the edges of our fully connected graph. For each edge embedding e_{ij}^L of the last graph convolution layer L, the MLP outputs the probability p_{ij} that this edge is included in the tours of the CVRPTW solution:

$$p_{ij} = \text{MLP}(e_{ij}^L). \tag{2.5}$$

The edge representations are linked to the ground-truth tour, which is the best known solution for this instance, through a softmax output layer that converts the raw outputs of the network into probabilities that sum to 1, allowing them to be interpreted as the likelihood of each class. With this, we train the model parameters end-to-end by minimizing the cross-entropy loss via gradient descent.

2.2 Beam Search

To create valid and complete solutions from our network model's output, we cannot simply select the edges with the highest probability until all nodes are visited, as this often results

in invalid tours. Instead, we use a beam search [MCF+77], which is a limited-width-breadth-first tree search, to efficiently construct valid solutions.

Starting from the root node, which in our case contains only the depot but can also contain an initial partial solution as we will see in Chapter 3, in each layer of the search tree, only a subset of the nodes with regard to a scoring policy P is further explored, where P is given as

$$P(c) = \prod_{i \sim j} p_{ij},$$

where $i \sim j$ denotes that node v_j follows node v_i in c.

The descendants of a node in layer ℓ of the search tree are those nodes that are eligible as the next stop for the partially constructed tour. In the context of CVRPTW, where we have dynamic parts of partial solutions, such as the current point of time and the already occupied capacity of the vehicle, we apply a masking strategy to efficiently build valid solutions. This is done by masking out invalid descendants in layer $\ell + 1$ with respect to the time window and demand constraints as well as the already visited nodes in this partial solution.

For this, let $S^\ell \in \mathbb{R}^{b \times (n+1)}$ be the scoring matrix in layer ℓ with respect to P. Given the partially constructed tour $c_{b'} = [v_1, \ldots, v_\ell]$, the $b'-$th row of the scoring matrix $S^{\ell+1}$ is given as the policy score $P(c_{b'})$, multiplied with the decision weight given by the GCN to transit at the next step from v_ℓ to w for all nodes $w \in [n+1]$:

$$S^{\ell+1}_{b'w} = P(c'_b) \cdot p_{v_\ell w}.$$

Then, the invalid expansions are excluded. For this, define the binary matrix $M^{\ell+1} \in \{0,1\}^{b \times n+1}$ to be the mask for step $\ell + 1$, which will take the role of filtering the already visited nodes as well as those nodes that violate the capacity and time windows constraint, given the partial tour $c_{b'}$ for every row b' of $S^{\ell+1}$. The boolean entries in $M^{\ell+1}$ indicate infeasible expansions. Define $M^{\ell+1}_{bj} = 0$, if node v_j is already visited in the partial tour c_b or if adding v_j to c_b as the next node would violate the capacity or time windows constraints, and 1 otherwise. From the $b \cdot (n+1)$ values in the masked scoring matrix $S^{\ell+1} \cdot M^{\ell+1}$, the b highest scores are chosen, and the associated b search tree nodes are expanded in the next step.

The parameter b is called the beam width. In our case, the scoring function is the probabilities gained by the GCN and we choose the b nodes whose connecting edges hold the highest probability. This is done iteratively until all nodes in the graph are visited. If a node contains a full solution, the solution is evaluated and stored. The beam search stops when no more branches are possible on the current level, i.e. when b complete solutions have been found. The final solution then is the one that yields the highest score with respect to the scoring function, which translates to having the highest probability out of the b found solutions regarding the output of our neural network. For an example of several beam search steps and the underlying decisions, we refer to Example 1.1.

Beam search is asymptotically optimal for $b = n \cdot 2^n$, but choosing a smaller value for b allows us to trade solution quality for computational performance and memory needed

since it decreases the search space but possibly the best solution is pruned. The pseudocode for the beam search is presented in Algorithm 1.

Algorithm 1: Beam Search

Input: graph G, beam width b, policy model P
Output: tour length
Initialize $OpenList = \{[0]\}$
while $OpenList \neq \emptyset$ **do**
  $candidates = \emptyset$
  for each s in $OpenList$ **do**
    for each node v with $v \notin s$ **do**
      if $s \cup \{v\}$ is feasible w.r.t. the constraints **then**
        $candidates = candidates \cup \{s \cup \{v\}\}$
      end
    end
  end
  for each $c \in candidates$ **do**
    Compute policy score $P(c) = \prod_{i \sim j} p_{ij}$
  end
  $OpenList = \emptyset$
  while $|OpenList| < b$ and $candidates \neq \emptyset$ **do**
    $c = \underset{c' \in candidates}{\arg\max} \ (P(c'))$
    $OpenList = OpenList \cup \{c\}$
    $candidates = candidates \setminus \{c\}$
  end
end
Compute tour length of $c = \underset{c' \in OpenList}{\arg\max} \ (P(c'))$
return tour length

The approach of choosing the solution with the highest probability produced by the beam search is called GCNBS in the following. However, out of the b solutions found, we can also select the one with the shortest overall tour. This follows the approach in [BPL+16], where they sample a set of solutions and select the shortest one as the final solution out of this set, and can be interpreted as a shortest tour heuristic [JLB19a], which is therefore called GCNBSSTH in the remainder of this chapter.

2.3 Computational Experiments

We evaluate our approaches on three different problem sizes, ranging from 20 to 100 nodes. For each problem size, we train our neural network on different datasets. We report

the results of our approaches and compare them to highly developed heuristics like LKH [Hel17] and Google's OR-Tools [PF22], which frequently serve as heuristic baselines in related work, as well as the commercial state-of-the-art exact solver Gurobi [GO22]. The heuristic LKH plays a crucial role in this, as it serves as both the source of training data for our neural network and the benchmark against which the quality of solutions generated by other methods is evaluated. We refer to this as the solution gap, which denotes the disparity between the solutions produced by various methods and the solution obtained by LKH. The found solutions are compared in terms of objective value (cost), solution gap and computation times. We show that our approaches achieve results close to the LKH solutions for smaller problem instances, but are outperformed by Gurobi for large instances, which achieves better results but requires more computation time.

2.3.1 Implementation and Hyperparameters

All models are implemented in Python 3.9 and run under Linux. The neural network architecture is implemented using PyTorch version 1.12.1 [PGM+19] to use GPU computation with Cuda version 11.3.

Our neural network model does not contain a large number of hyperparameters. The graph convolutional neural network consists of $\ell_{\text{GCN}} = 30$ hidden layers and $\ell_C = 3$ layers in the MLP. We use a hidden dimension $h = 300$ in each of the layers. For the beam width b we use different values from $b = 1000$ to $b = 50000$.

For the implementation of the beam search, we define similar to [KvHGW22] an auxiliary graph $G' = (V', E')$, which is obtained from the input graph $G = (V, E)$ for the beam search in the following way: For $V = \{0, \dots, n\}$ with 0 being the depot node, let $V' = (0, \dots, n, n+1, \dots, 2n)$ be the set of nodes, where we add for each node i in the original graph (except the depot) a new node i'. Connections to these new nodes denote connections via the depot to node i. This allows us to implement the beam search such that at step k in the beam search exactly k of the n nodes are visited, so that comparisons of partial solutions and building the full solution incrementally are more straightforward. The transition probabilities p_{ij} for edges (i, j) for $i, j \in \{0, \dots, n\}$ are given by the neural network. For edges (i, j') with $j' \in \{n+1, \dots, 2n\}$ set $p_{ij'} = p_{i1} \cdot p_{1j} \cdot 0.1$, where multiplication is used so that $c_{ij'} \in (0, 1)$ can still be interpreted as a probability. The factor 0.1 is multiplied to incentivize building as few routes as possible and therefore implicitly minimize the number of vehicles used.

Remember that infeasible connections concerning the demand and time window constraints are masked out in each beam search step so that a connection from i to a node $j' \in \{n+1, \dots, 2n\}$ is only included if the travel times from i via the depot to $j = (j' \bmod n)$ are feasible with regard to the time window constraints. On the other hand, if a connection from i to a node $j \in \{1, \dots, n\}$ is masked out for a partial solution because of the demand constraint, the connection from i to $j' = j + n$ is included, as the route via the depot resets the occupied capacity.

2.3.2 Experiment Setup

Data Generation

We sample problem instances based on the distribution given in the R201 instance of the benchmark set of [Sol87], which consists of randomly generated geographical data, a long scheduling horizon and short to medium-sized time windows allowing only a few customers per route. We follow the approach used in [FST20] for the data generation.

The node locations are uniformly and randomly sampled within the square region $[0, 100]^2$ for all instances. Capacities are assigned based on problem sizes, with values of $Z^{20} = 500$, $Z^{50} = 750$, and $Z^{100} = 1000$. To determine the node demands, we adopt the R201 distribution, drawing values from a normal distribution $\hat{q} \sim \mathcal{N}(15, 10)$. This choice is motivated by the fact that the R201 instance exhibits demands with a mean of 17.24 and a standard deviation of 9.4175. The obtained demand values are then rounded down to integers. For the time windows, we ensure feasibility by considering the travel time required from the depot to each node. We define a suitable time window horizon, denoted as $T_i = [T_i^0, T_i^1]$, for each node v_i, which depends on the travel time between the depot and node v_i. To construct the time windows, we uniformly sample the start time t_i^0 from the interval T_i and then sample the end time t_i^1 uniformly from the range $[\hat{t_i^0}, T_i^1]$. Here, $\hat{t_i^0} = t_i^0 + 300\varepsilon$ is determined by adding a small perturbation, where $\varepsilon = \max(|\hat{\varepsilon}|, 1/100)$ and $\hat{\varepsilon} \sim \mathcal{N}(0, 1)$.

To generate solutions for the randomly sampled instances for training purposes, we use machines with two CPUs of type Intel Xeon E5-2680v3 @ 2,50GHz with 12 cores from the High-Performance Computing Cluster at Hamburg University of Technology. The instances with 20 customers are solved to optimality by Gurobi [GO22], whereas the instances with 50 and 100 customers, respectively, are solved using one run of LKH [Hel17].

Training and Evaluation

For each problem size, an individual neural network is trained on 1 million instances with the respective number of nodes, which is split into training, test, and validation sets with an 80/10/10 ratio. We use a supervised learning approach to train the model. Given a graph with additional node features such as time windows and demands, the model learns to output a probability matrix. This is achieved by minimizing the cross-entropy loss via gradient descent, using the adjacency matrix of the target solution as the reference. We utilize the Adam optimizer [KB14] along with a gradual decrease in the learning rate for smoother convergence, starting at a rate of 10^{-3}. The target solutions are generated using the LKH heuristic and are therefore not necessarily optimal. We train using a batch size of 24, which was chosen as the result of manual tests on our GPUs. We train the nets for 1500 epochs with 500 randomly chosen batches and select the point of training with the lowest validation loss. The training procedure is executed on machines with two CPUs of type Intel Xeon E5-2680v3 @ 2,50GHz with 12 Cores and four NVidia Tesla K80 GPUs with 12GB RAM each. We note however that it is not necessary to use a multi-GPU setup

to train or evaluate our models, identical results can be attained by training longer on a single GPU.

For the evaluation of our model GCNBS, we use the same machines as for the training procedure. The baseline models are run on a machine with two CPUs of type Intel Xeon E5-2680v3 @ 2,50GHz with 12 Cores.

Baseline Models

We compare our models to the commercial exact solver Gurobi version 10.0.0 [GO22], as well as the highly-optimized heuristic solvers LKH [Hel17] and the Google OR-Tools (GORT) version 9.5.2237 library [PF22], both of which frequently are used as heuristic baselines in related work. Although Gurobi may not be the best exact solution method, it serves as a representative example of commercial solvers. Gurobi is applied to the two-index-MILP presented in Section 1.3.1 and solves the model with a single run using all available threads. Since Gurobi strives to find the best solution, we have to set a time limit after which Gurobi stops searching for better solutions. We set this time limit to 1800, 3600 and 5400 seconds for the different problem sizes, respectively, as a lower time limit does not yield any high-quality solutions for large instances. LKH is executed using one run per instance. For GORT, one can choose different configurations regarding the underlying local search heuristic. Experiments have shown the guided local search (GLS) to be best suited for our needs. As a time limit has to be set for the GLS, we have chosen time limits of 10, 100, and 300 seconds for different problem sizes based on manual experiments.

2.3.3 Results

We compare our results in terms of the quality of the solutions, which is measured as the percentage gap between the found solution and the solution found by LKH, as this was the baseline solver for generating the training data for our neural network, and computation time with results from commercial solvers (Gurobi) and heuristics (LKH, GORT). However, the calculation times are generally difficult to compare, as they depend heavily on different factors such as the implementation (Python vs C++) and which hardware (GPUs vs CPUs) was used. Therefore the comparison of run times can only be done conditionally.

Experimental Results

Table 2.1 shows the results of our models for the three different problem sizes with beam width $b = 1000$ and $b = 50000$, denoted as 1K and 50K in the model name, and compares our results to results obtained by the previously described exact and heuristic baseline solvers. The dataset consists of 1000 randomly sampled instances from the distribution described in Chapter 2.3.2. We report the average gap to the solution that was obtained by LKH, as well as the average calculation time per instance in seconds. According to the data presented in Table 2.1, for instances with $n = 50$ nodes, Gurobi has a negative gap with respect to the ground-truth solutions.

Model	Problem size								
	$n = 20$			$n = 50$			$n = 100$		
	cost	gap	time	cost	gap	time	cost	gap	time
GCNBS 1K	6.40	7.56	6.82	12.14	19.84	32.49	19.91	32.03	97.28
GCNBSSTH 1K	6.28	5.54	7.24	11.73	15.79	33.51	18.63	23.54	98.79
GCNBS 50K	6.31	6.05	237.51	11.87	17.18	1148.73	18.86	25.06	4306.82
GCNBSSTH 50K	6.12	2.86	240.01	11.31	11.64	1153.01	17.86	18.43	4313.61
Gurobi	5,95	0.00	0.60	10.10	−0.29	2092.37	16.98	12.63	5337.80
GORT-GLS	5.95	0.00	10.00	10.13	0.03	100.00	15.12	0.28	300.00
LKH	5.95	0.00	6.20	10.13	0.00	11.74	15.08	0.00	19.69

Table 2.1: Mean Cost, Optimality Gap and Computation Time (sec) Per Instance

In general, our learning-based approach can improve its performance over the initial setting by additionally applying the shortest tour heuristic without noticeable implications regarding the computation time. For small instance sizes, with a beam width of 50000 GCNBSSTH finds solutions close to the LKH results with a mean gap of approximately 2.86% on the dataset used to also train the neural networks. Our heuristic outperforms the commercial solver Gurobi [GO22] in terms of speed for larger instance sizes. In fact for instances with 100 nodes, Gurobi could not find valid solutions within the time limit most of the time. The state-of-the-art heuristics LKH [Hel17] and Google's OR-Tools [PF22] outperformed our heuristic in terms of speed and solution quality on all instance sizes. Especially for large instances with 100 nodes the shortcoming of our model becomes apparent, as the beam search becomes computationally expensive for larger instances and beam widths. The computation time mainly depends on the chosen beam width and it seems to be close to a linear correlation between beam width and computation time. Tests have shown that most of the computation time in the current implementation is used in the masking of invalid next nodes with respect to the time window constraints within the beam search and a more efficient implementation of the masking scheme could lead to significant improvements regarding the computation time. The results of our heuristic for small instances are promising, as it is shown that the relevant graph and instance properties can be extracted by our graph convolutional network, which allows us to efficiently build high-quality solutions for small instances. However, it is evident that for larger instance sizes, the beam search has problems finding the best solutions. The search tree becomes very large very quickly and the greedy decisions of the heuristic lead to the wrong part of the search tree more often for larger instances.

Figures 2.1 and 2.2 show solution plots for two different instances with 20 nodes as well as the groundtruth tour in the upper left corner and the probabilistic heatmap output of our GCN in the upper right corner of Figure 2.1 (left and middle plot in Figure 2.2, respectively). A darker red color in the heatmap implies a higher probability and only edges with a probability of at least 0.25 are plotted. The grey edges show all edges of the fully connected graph. In Figure 2.1 we can see the improvement of the solution by

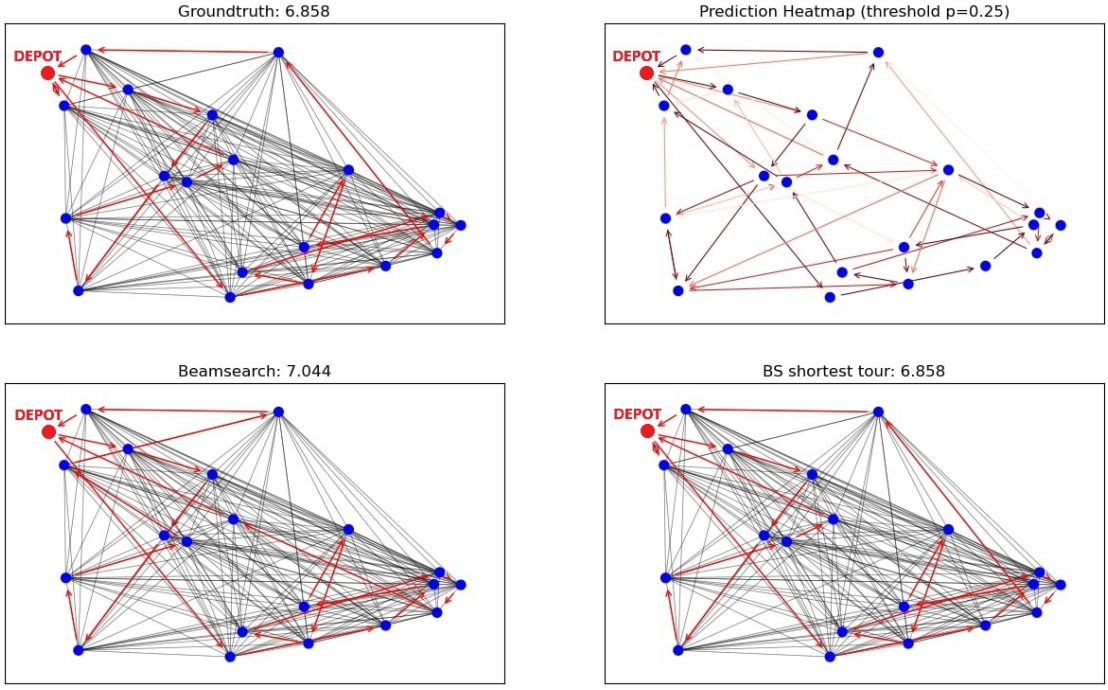


Figure 2.1: Heatmap and Solution Plot GCNBS for $n = 20$

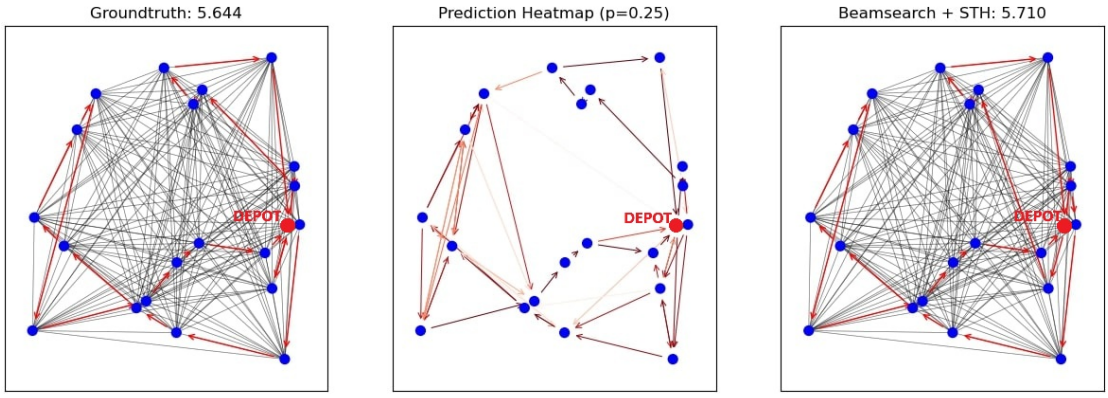


Figure 2.2: Heatmap and Solution Plot GCNBSSTH for $n = 20$

applying the shortest tour heuristic. The solution of the beam search uses one vehicle less than the groundtruth tour, but by using one more vehicle, GCNBSSTH can lower the total time traveled and find the best solution. Figure 2.2 showcases the ability of GCNBSSTH to

correctly identify most of the important edges as important, while also excluding most of the irrelevant edges. But in contrast to the solution found in Figure 2.1, here GCNBSSTH uses one vehicle less than the best solution.

2.4 Discussion

We proposed a deep learning-assisted heuristic to solve the capacitated vehicle routing problem with time windows. For this, we combined a Graph Convolutional Network with a classical tree search heuristic. Although our approach is not as advanced and specialized as other existing heuristics and the computational results for the approach using a beam search were not quite as good as those of state-of-the-art heuristic approaches such as LKH [Hel17] and Google's OR-Tools [PF22], it is still competitive on smaller instances, showing promising results by achieving close to optimal results on datasets containing instances with 20 nodes. Even though our model is not as efficient as these other heuristics, it is remarkable that it is possible to produce near-optimal results for smaller instances without requiring deep expert knowledge regarding the combinatorial optimization problem, which displays the potential of applying deep learning techniques to accelerate combinatorial optimization techniques.

On the other hand, it became clear that the beam search has its weaknesses for larger instances. While the results for smaller instances show the ability of the GCN to capture the relevant information of the graph and the instance constraints, the beam search was not always able to find high-quality solutions for larger instances. This is due to the fact that the decisions of the heuristic are greedy and choosing a wrong path in the search tree is irreversible. For this reason, in the following chapter, we develop a more sophisticated tree search heuristic to solve the CVRPTW. In doing so, we address the shortcoming of the beam search by not greedily deciding which path to follow, but iteratively simulating the effects of different decisions on the search process before deciding which node to add to the current tour. For this purpose, we adapt the GCN developed in Section 2.1 and use the beam search to support the decision process.

Chapter 3

Graph Convolutional Neural Network Assisted Monte Carlo Tree Search for the CVRPTW

In Chapter 2, we developed a deep neural network to assist a tree search in finding the best solution. While the computational results for smaller instances demonstrated the potential of using the deep neural network's predictions, the tree search for larger instances revealed its weakness: getting lost within the search tree. Therefore, in this chapter, we develop a more sophisticated tree search that utilizes the network's predictions to guide the search and applies mechanisms to balance the exploration of the most promising paths while ensuring that other parts of the search tree are not neglected. This chapter is based on the paper [DK24] that is joint work with Tobias Klein.

For this, we propose a novel context-complemented graph convolutional network which is integrated into a Monte Carlo Tree Search (MCTS) to solve the CVRPTW. The deep convolutional part of the proposed neural network builds efficient CVRPTW graph representations, whereas the context part processes information of partial solutions to output probabilities on which vertex to add next to the tour, which is used during the expansion of the search tree. In each iteration of the MCTS procedure, one vertex is added to the solution, building the solution sequentially. At each iteration, the context-complemented graph convolutional network (CCGCN) is evaluated given the new context of the partial solution. For simulating final solution values within the Monte Carlo search tree, we conducted experiments using two different heuristics: LKH and a beam search. By leveraging the probabilities provided by our CCGCN and employing a variation of the *Upper Confidence Bound Applied to Trees* method, we manage the tradeoff between exploration and exploitation.

In the subsequent sections, we provide a detailed description of the MCTS procedure, explain the neural network's architecture and in which ways it is used within the MCTS framework and conduct computational experiments.

3.1 Monte Carlo Tree Search

The MCTS is a powerful combination of the Monte Carlo Sampling and a Tree Search, first proposed by [Cou07]. By iteratively simulating decisions and assessing their outcomes, a search tree is gradually constructed while emphasizing promising paths. To balance the tradeoff between exploration of the search tree and exploitation of the most promising areas of the search tree, the Upper Confidence Bounds applied to Trees (UCT) [KS06] is applied (cf. Section 3.1.1).

In recent years, due to Google's famous algorithm AlphaGo [SHM+16], the MCTS gained popularity as a heuristic in combination with neural approaches for solving combinatorial optimization problems. [SHM+16] demonstrate with their program AlphaGo the potential of combining MCTS with neural networks in the selection and simulation phases by defeating professional players in the board game Go. In the context of two-player games such as Go or Chess, the value of a node in the search tree corresponds to the estimated probability of winning based on the aggregated outcomes of its subnodes. Simulation involves playing out the remainder of a game from a particular node, resulting in a win or loss outcome that is then propagated back through the tree, updating the values of all preceding nodes. In the context of CVRPTW, where we do not have an outcome of either a win or loss, but rather are interested in minimizing some objective value, modifications need to be made to all phases of the traditional MCTS algorithm [Cou07]. An overview of recent modifications and applications of the MCTS applied to a wide range of problems is given in [ŚGSM22].

The MCTS consists of a four-step process: selection, expansion, simulation and backpropagation. During the selection phase, starting at the root node, the tree is traversed to find the most promising leaf for further exploration. This leaf node is then expanded by generating a set of new child nodes, each representing a feasible continuation. Then, every new node undergoes simulation to generate an associated value estimating the costs of a complete solution starting from this node, which is then backpropagated through the tree up to the root node.

Algorithm 2 provides an overview of the MCTS process at a higher level, omitting the specific details for each individual phase of the process.

In each iteration $i \in [n]$ of the algorithm, one node of the graph is added to the solution. For this, an MCTS search tree T_i is iteratively constructed to guide the decision process for each iteration step $i \in [n]$. Starting from the root node of T_i, which represents the solution built in the first $i-1$ iteration steps, each node in the search tree represents a continuation of the solution from its parent node.

Each rollout of the iteration consists of selecting a leaf node L in the search tree T_i representing a partial solution $L_s = [v_0 \dots, v_k]$, which is then subsequently expanded by adding child nodes to L. Let $U_L = V \setminus L_s = \{v_{u_1} \dots, v_{u_r}\}$ be the set of vertices of G which are not in L_s. For each u in U_L, a child node L^u in the search tree is added to L, with $L^u_s = [v_0 \dots, v_k, u]$.

Each edge to a child node L^u of L is assigned a decision weight $p_{v_k u}$, which is used to

Algorithm 2: CCGCN-MCTS

Input: Graph $G = (V, E)$ with $|V| = n$, neural network N, number of rollouts M
Output: solution S
Initialise solution $S = \emptyset$.
for $i = 1 \ldots, n$ **do**
    Set root node $R = S$ of the MCTS search tree T_i.
    Evaluate neural network $N(G, S)$.
    for $r = 1, \ldots, M$ **do**
        `Select` leaf node L in T_i with associated partial solution L_s w.r.t. decision weights of edges in T_i
        if L_s is complete solution **then**
            `Backpropagate` tour length of L
        end
        else
            `Expand` L with decision weights for edges to child nodes of L obtained by $N(G, L)$
            `Simulate` tour length L_q of complete solution starting with partial solution L_s
            `Backpropagate` simulated tour length L_q of L_s
        end
    end
    Select $B = \underset{B \text{ child of } R}{\arg\min} (B_{q_{\min}})$
    Set $S = B$
end

guide the selection phase and can be interpreted as the probability of visiting u after v_k. These decision weights are obtained by the neural network presented in Section 3.2, which is evaluated at the start of every iteration i by propagating the graph G as well as the partial solution S from iterations 1 to $i - 1$ to incorporate the context of S with respect to time window and capacity constraints.

It is then simulated which travel time of a complete solution is to be expected, if we start with the partial solution L_s, by applying a value function. This simulated value is subsequently backpropagated in the search tree up to the root node of T_i and the values stored at each node in the search tree, which are used during the selection of a leaf node, are updated accordingly. These values consist of the best ($q_{\min}$) and worst ($q_{\max}$) tour lengths found in their respective subtrees and the number of times the search tree node was visited, which are used to guide the selection of leaf nodes during the selection phase (cf. Section 3.1.1). Additionally, parameters regarding the partial tour are stored at this node to accelerate the computation of the expansion process (cf. Section 3.1.4).

When the number of rollouts for this iteration is reached, which is also an upper bound

for the depth of T_i, the child node B of the root node R with the best (minimal) tour length found in its subtree is chosen to be the root node for the next iteration $i+1$ and all other child nodes of R are disregarded, while all information in the subtree of B obtained in iteration i regarding values and parameters at the search tree nodes is kept.

The flowchart presented in Figure 3.1 from [DK24] outlines the step-by-step progression within one rollout of the MCTS. Example 1.2 in Chapter 1 shows each phase of one rollout within one iteration of the MCTS.

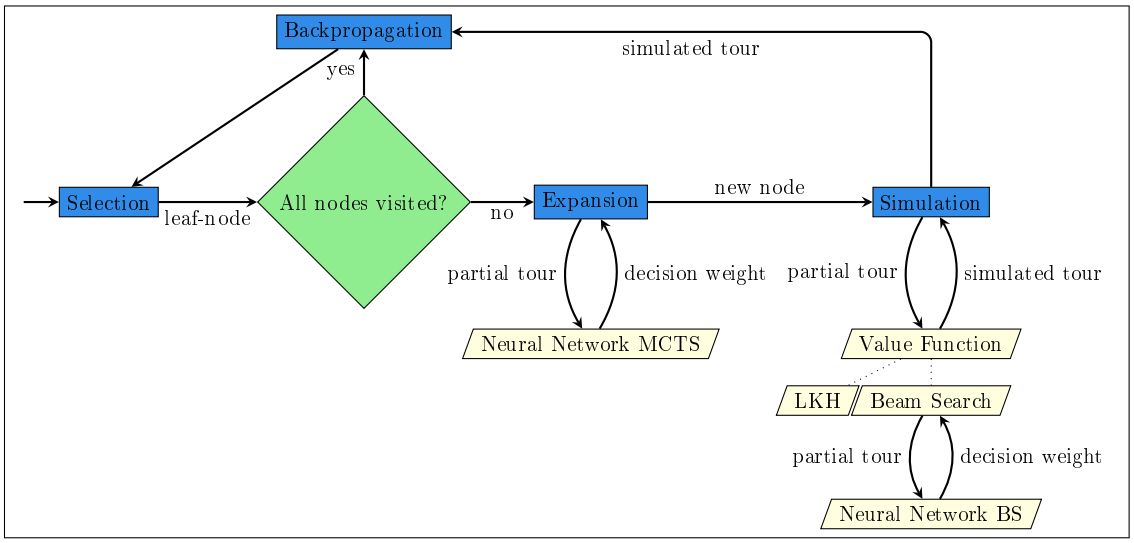


Figure 3.1: Decision Flowchart of One Rollout of the MCTS

A detailed explanation of the four phases is given in the following sections.

3.1.1 Selection Phase

The selection phase in our approach involves iteratively choosing the most promising child node until a leaf node is reached, starting from the root node. In each iteration, we calculate a value based on the simulated tour length q for every child node, which is combined with an adapted variant of the Upper Confidence Bound applied to Trees (PUCT) algorithm [KS06] to handle probabilities effectively while balancing exploitation and exploration. The child node is selected that minimizes the following expression:

$$\min \frac{q - q_{\min}}{q_{\max} - q_{\min}} - c_{\text{puct}} \cdot p_{ij} \cdot \sqrt{\frac{n_{\text{parent}}}{n}}. \tag{3.1}$$

In the first term, $q_{\min}$ and $q_{\max}$ represent the lowest and highest node values found in the subtree starting at the current node. The simulated tour length q is normalized to accommodate for variations in travel times across different problem instances, as we are concerned with tour lengths rather than win probabilities encountered in two-player games like Go. In the second term of (3.1), which represents the PUCT value, c_{puct} denotes a manually chosen constant value that regulates the trade-off between exploration and

exploitation, p_{ij} represents a decision weight that the current node v_j is visited by a vehicle after node v_i (cf. Section 3.1.2), n_{parents} indicates the count of how many times the backpropagation has processed the parent node, and n represents the count for the current node. As our objective is to find the tour of minimum length, we select the child node that minimizes (3.1). The inclusion of the second term ensures a balance of selecting the node with the best simulated tour length value, the first term of (3.1), and the one suggested by the decision policy p_{ij}. Not normalizing the simulated tour length would lead to an imbalance in the impact of the second term on the overall selection process across various problem sizes as the tour lengths increase. This imbalance could alternatively be mitigated by dynamically adjusting the value of c_{puct} based on the problem size. However, implementing such an adjustment would necessitate a significant amount of trial and error.

3.1.2 Expansion Phase

We use a deep neural network (cf. Section 3.2) to obtain a decision policy, denoted as p_{ij}, for each vertex $v_j \neq v_i$, indicating the likelihood of visiting vertex v_j next in the optimal tour given the partial tour represented by the selected node. A higher decision weight p_{ij} implies a higher probability of vertex v_j being visited after vertex v_i. However, it is important to note that although the decision weights are in the range of 0 to 1, they do not sum up to 1. The n-dimensional decision weight vector obtained from the CCGCN is subsequently filtered to exclude decisions that are not permissible. These exclusions occur when a vertex has already been visited or cannot be visited due to demand or time window constraints. For each remaining vertex v, a child node is added to the tree with the decision weight p_{ij} obtained from the CCGCN incorporating the context of the partial solution of v. The context consists of the current position, time and load of the partial tour. Subsequently, each expanded node is simulated to calculate the simulated tour length q, as well as the maximum $q_{\max}$ and minimum $q_{\min}$ tour lengths associated with the node. These values are essential for the selection phase of the MCTS algorithm.

3.1.3 Simulation Phase

In contrast to the AlphaGo approach [SHM+16], where a learned value function is applied to estimate the probability of the player winning from the current position, we apply a heuristic value function to calculate the expected tour length after completing the partial solution from the current position in the search tree.

For this, we apply two different value functions. First, we use the LKH heuristic [Hel17] by creating a new instance based on the partially given solution, which only consists of the unvisited nodes together with the last node v of the partial solution. By adding an auxiliary node with special edge weights, demand and time window we ensure that the LKH correctly completes any partial tour included in the partial solution of the initial instance with respect to the capacity and time window constraints. The weights of the edges connecting v_a with all unvisited nodes besides v and the depot are set to infinity, whereas the other two weights are set to zero, which forces the LKH to use these edges.

By setting the time window and demand of v_a to the time and load of the partially built route in the partial solution, we can mirror the context and obtain a valid estimation of the solution value.

For the second value function, we adapt the beam search developed in Chapter 2 and combine it with our trained CCGCN in a non-autoregressive form to get a good estimation of the possible tour length from the current partial solution. The value function is presented in Algorithm 3 and described in the following.

Starting from the root node, which usually contains a partial solution, in each layer of the beam search tree, only a subset of the nodes with regard to a scoring policy is further explored. For this, let b be the beam width, $S^k \in \mathbb{R}^{b\times(n+1)}$ be the scoring matrix at the $k-$th step of the beam search, which is subsequently calculated in each step of the beam search, and $c_{b'}$ be one of the b partial solutions currently ending in node v'. The scoring policy P of a (partially) built tour c is given as

$$P(c) = \prod_{i\sim j} p_{ij},$$

where $i \sim j$ denotes that node v_j follows node v_i in c and, for a graph G, partial solution c and $v \notin c$ a not yet visited node, we denote by $f(G, c \mid v)$ the probability of v being the next node to add to c given by our model. The $b'-$th row of the scoring matrix S^k is then given as the cumulated policy score $P(c_{b'})$, multiplied with the decision weight given by the CCGCN to transit at the next step from v' to w for all nodes $w \in [n+1]$:

$$S^{k+1}_{b'w} = P(c'_b) \cdot f(G, c_{b'} \mid w).$$

Then, the invalid expansions are excluded by applying a binary masking scheme for filtering the already visited nodes as well as those nodes that violate the capacity and time windows constraint, given the partial tour $c_{b'}$ for every row b' of S^{k+1}, as described in Section 2.2. From the $b\cdot(n+1)$ values in the masked scoring matrix, the b highest scores are chosen, and the associated b expansions are expanded in the next step. To encourage the simulation to minimize the number of routes, for tours via the depot node 0, we multiply the score by a factor of 0.1, similar to [KvHGW22].

Determining the invalid expansions during the beam search is computationally demanding since calculating them with regard to the constraints is unique for every beam in the search, which leads to a linear dependency between the run time of the simulation and the chosen beam width. To accelerate the simulation process, we alternatively apply a relaxed value function, where the task of filtering invalid expansions with regards to the constraints is delegated to the neural network's decision weights p_{ij}, as in theory, the network should naturally assign low weights to invalid next nodes. This translates to relaxing the CVRPTW instances to VRP instances to be solved with the deep neural network-assisted beam search. After the beam search is finished finding b solutions, a heuristic is applied to check their feasibility and returns the lowest value from the subset of feasible solutions. If all b solutions are deemed infeasible, the lowest value of those is returned. Given that the value function's determined value serves solely as a guide for the MCTS

Algorithm 3: Value Function

Input: graph G, beam width b, partially constructed solution T, policy function P
Output: tour length
Initialize $OpenList = \{T\}$
while $OpenList \neq \emptyset$ **do**
  $candidates = \emptyset$
  for each s in $OpenList$ **do**
    for each node v with $v \notin s$ **do**
      if $s \cup \{v\}$ is feasible w.r.t. the constraints **then**
        $candidates = candidates \cup \{s \cup \{v\}\}$
      end
    end
  end
  for each $c \in candidates$ **do**
    Compute policy score $P(c) = \prod_{i=1}^{|c|} f(G, T \mid c(i))$
  end
  $OpenList = \emptyset$
  while $|OpenList| < b$ and $candidates \neq \emptyset$ **do**
    $c = \underset{c' \in candidates}{\arg\max} \ (P(c'))$
    $OpenList = OpenList \cup \{c\}$
    $candidates = candidates \setminus \{c\}$
  end
  Choose arbitrary $\bar{c}$ from $OpenList$
  if $\bar{c}$ is a complete solution **then**
    Compute tour length of $c = \underset{c' \in OpenList}{\arg\max} \ (P(c'))$
    return tour length
  end
end

exploration, any errors resulting from invalid simulations during the simulation phase do not lead to overall invalid solutions, as the feasibility is monitored within the MCTS.

3.1.4 Backpropagation

The purpose of the backpropagation phase is to propagate the simulated tour values of a node back through the tree to use these values during the selection phase. In contrast to backpropagation methods employed in two-player games, where values of child nodes are typically added to those of the parent node, our objective is to find the shortest tour. Therefore, the simulated tour length q is propagated back alongside the minimum ($q_{\min}$)

and maximum (q_{max}) tour lengths using the following formula:

$$\begin{aligned} q_{\text{parent}} &= \min(q_{\text{parent}}, q_{\text{child}}) \\ q_{\text{min\_parent}} &= \min(q_{\text{min\_parent}}, q_{\text{min\_child}}) \\ q_{\text{max\_parent}} &= \max(q_{\text{max\_parent}}, q_{\text{max\_child}}). \end{aligned}$$

By applying these backpropagation rules, the q value of a node is only updated if the propagated value from its child node is smaller than its current q value. The same principle applies for updating q_{min}, ensuring that these values are only updated with smaller values during the backpropagation and never increase. Conversely, q_{max} is updated when the value of the child node is greater, thus q_{max} only increases during the backpropagation. Our approach ensures that during the selection phase, the q value of a node always represents the smallest q value among all its descendants, aligning with our goal of searching for the smallest tour. In addition, we update the step count n at each node that is visited during the backpropagation.

3.2 Neural Network Architecture

We propose an extension to the deep Graph Convolutional Network framework developed in Chapter 2 that incorporates two supplementary encoders. These encoders capture contextual information about previously visited nodes and the current partial tour state representation, respectively. By leveraging this additional information, our model can be used autoregressively to make more informed decisions regarding the selection of the next vertex in the graph traversal process. The complete architecture of our neural network model is shown in Figure 3.2 [DK24].

The GCN presented in Chapter 2 allows us to efficiently learn to find representations of a graph by computing h_1-dimensional representations of all nodes and edges integrating the node and edge features, respectively, which can be used to produce a 'heatmap' over the edges of a graph, indicating promising edges to be used in a solution. For a detailed description, we refer to Section 2.1.

For the context embedding, define the partial tour state to consist of the current position of the vehicle, given by the coordinates of the last visited node, the current time and the remaining available capacity of the vehicle. The partial tour, consisting of the already visited nodes, gets a separate learned initial embedding parameter. For that, define $y_{\text{tour}} \in \{0,1\}^{n+1}$ to be the indicator vector with entry one for every already visited node. Let h_s and h_v be the dimensions of the state and tour embeddings, respectively. The partial tour state features are concatenated to the four-dimensional partial tour state feature vector and are then embedded by a feed-forward neural network N_s to an h_s-dimensional representation consisting of the $\frac{h_s}{3}$-dimensional representations $\gamma^{\text{loc}}, \gamma^{\text{load}}, \gamma^{\text{time}}$ for the three partial tour state features, respectively. The tour vector y_{tour} is embedded into a h_v-dimensional representation by a second feed forward neural network N_v given as $\gamma^{\text{tour}} = N_v(y_{\text{tour}})$.

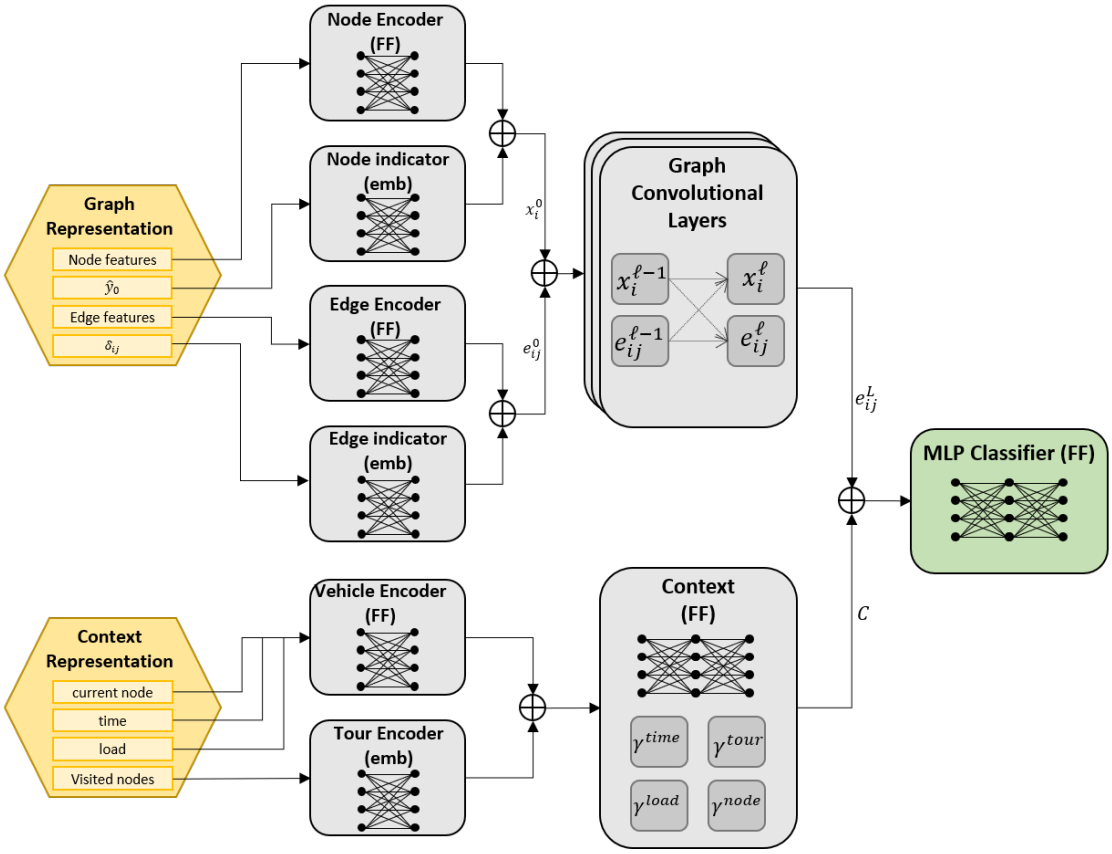


Figure 3.2: Context-Complemented Graph Convolutional Network Diagram

The context embedding C_{emb} is then given as

$$C_{\text{emb}} = [\gamma^{\text{loc}}, \gamma^{\text{load}}, \gamma^{\text{time}}, \gamma^{\text{tour}}],$$

the concatenated state and tour representations with a dimension of $h_2 = h_s + h_v$, which is then subsequently passed through the context feed-forward neural network N_c as follows to derive the h_2-dimensional context representation $C = N_c(C_{\text{emb}})$:

$$C^{\ell+1} = C^{\ell} + \text{ReLU}\left(W^{\ell} C^{\ell}\right),$$

where $W^{\ell} \in \mathbb{R}^{h_2 \times h_2}$ are the learned parameters and ReLU is the rectified linear unit activation function.

The h_2-dimensional context embedding C together with the embedded h_1-dimensional graph representation, which is passed through several graph convolutional layers, are used to compute the probability of each node to be selected as the next node of the partial solution by being passed through an MLP Classifier, which is a fully connected feedforward neural network, to generate the desired output.

3.3 Computational Experiments

We evaluate our approach on randomly generated instances for three different sizes, ranging from 20 to 100 nodes, and compare the results to baseline models, which include exact commercial solvers as well as state-of-the-art heuristics for routing problems and the neural network-assisted beam search heuristic from Chapter 2. The results are compared in terms of solution quality and computation times. For each instance size we train different neural networks based on datasets containing one million pairs of instances and solutions of the same instance size. Furthermore, we analyze how the GCN's predictions enhance the effectiveness of the MCTS by conducting experiments where the prediction values are uniformly distributed.

3.3.1 Implementation and Hyperparameters

For the neural network, we set the hidden dimension for the graph convolutional part to $h_1 = 300$ with 30 hidden layers. The context hidden dimension is set to $h_2 = 128$ with 6 layers and the MLP classifier contains 3 layers. In the MCTS we vary the values for the number of playouts per round between 200 and 1000, the beam size for the simulation is set to values between 1 and 10. To find the best configuration of hyperparameters for the MCTS, each combination of (number of playouts, beam size) was tested on datasets containing 50 instances with different c_{puct} values. We report the results for the parameter combinations (200,10), (500,5) and (1000,10) to illustrate the trade-off between computational efficiency and accuracy. We choose a c_{puct} value of 1.4 based on manual experiments.

All models are implemented in Python 3.10 and run under Linux, but it is to be mentioned that computational runtimes could be reduced by coding it in C++. The neural network architecture is implemented using PyTorch version 2.0.0 [PGM+19] to use GPU computation with Cuda version 11.8.

3.3.2 Experiment Setup

We use the same set of instances sampled as described in Section 2.3.2. For each problem size, we generate a dataset containing one million pairs of instances and solutions, which is split into training, test, and validation sets with an 80/10/10 ratio. For each instance, we divide the solution $s = [v_1, \ldots, v_{\hat{n}}]$ at a randomly chosen position r to obtain the partial solutions $s_1 = [v_1, \ldots, v_r]$ and $s_2 = [v_r, \ldots, v_{\hat{n}}]$. The partial solution s_1 is then added to the instance which is passed to the neural network as input. The target for the training procedure is computed as the $(n \times n)-$dimensional adjacency matrix M with $M_{ij} = 1$ if node v_j follows node v_i in s_2, and zero otherwise.

In our approach, we employ a supervised learning methodology to train the model. The input consists of a graph with the additional node features time windows and demands along with a partial solution. The objective is to train the model to generate a probability matrix by minimizing the cross-entropy loss via gradient descent with respect to the adjacency matrix given as the target. To optimize the learning process, we utilize

the Adam optimizer [KB14] and gradually reduce the learning rate to facilitate smoother convergence, starting at a value of 10^{-3}. Each neural network is trained for a maximum of 2000 epochs, each epoch consisting of 500 batches containing 48 randomly chosen pairs of instances and targets. We then select the training level with the lowest validation loss. The training procedure is conducted on machines equipped with the same CPUs as for the data generation. Additionally, the machines are equipped with four NVidia Tesla K80 GPUs, each with a RAM capacity of 12GB. It is worth noting that although a multi-GPU setup is available, achieving identical results can be accomplished by training for a longer duration on a single GPU.

3.3.3 Computational Results

We compare our approach to the same baseline solvers that include highly specialized and optimized methods tailored explicitly for this problem, such as the LKH heuristic [Hel17] and Google's OR-Tools library [PF22], as well as commercial general-purpose solver Gurobi [GO22]. The details of the baseline solvers are outlined in Section 2.3.2. The solution quality is measured by calculating the gap between the solution obtained by our approach and the solution obtained by one run of the LKH heuristic, as these solutions were used to train the neural networks. Furthermore, we report the results of the LKH heuristic, where the number of runs is adjusted to achieve a similar runtime as our approach, to make the results more comparable, which is abbreviated by LKH_{adj}.

The evaluation of our approach as well as the baseline models is done on datasets containing 1000 randomly generated instances following the same data distribution as discussed in Section 2.3.2. Table 3.1 shows the results of our approach, abbreviated as CGM^{bs} for the beam search with exact masking and CGM^{rs} for the relaxed simulation approach, as well as CGM^{LKH} for the version with LKH as the value function, compared to the baseline models.

For $n = 20$, our heuristic CGM^{bs} achieves an optimality gap of 0.43% in the best case, while CGM^{LKH} as well as the baseline models can find the optimal solution. While for medium and large-sized instances the optimality gap widens for CGM^{bs}, it is able to outperform the commercial solver Gurobi in both computational time and solution quality for instances with 100 nodes and is also able to outperform the beam search with a beam width of 50000. With an optimality gap of 8 to 12% for large instances, and runtimes between 1300 and 5500 seconds, we can see the trade-off between running time and solution quality by choosing different hyperparameters.

For medium instances with 50 nodes, CGM^{LKH} can outperform LKH with one run and GORT and achieve optimal solutions, albeit with a significantly longer runtime, which also holds for Gurobi and LKH with an adjusted number of runs. The CGM^{LKH} heuristic achieves optimality for small instances while outperforming GORT-GLS, Gurobi as well as LKH with one run on medium and large instances, albeit with a significantly longer runtime. It also outperforms the beam search heuristic for all instance sizes.

The negative optimality gap with respect to LKH indicates that the solutions calculated by one run of LKH are 0.31% worse on average compared to an optimal solution. As our

Model	$n = 20$			$n = 50$			$n = 100$		
	cost	gap	time	cost	gap	time	cost	gap	time
$CGM^{bs}_{(200,10)}$	6.03	1.28	29.48	11.18	10.39	421.05	16.90	12.05	1355.04
$CGM^{bs}_{(500,5)}$	6.01	0.98	50.47	10.93	7.89	766.18	16.53	9.60	2458.87
$CGM^{bs}_{(1000,10)}$	5.97	0.43	112.99	10.79	6.50	1877.02	16.36	8.52	5464.28
CGM^{LKH}_{200}	5.95	0.00	91.47	10.12	-0.14	1128.64	15.04	-0.24	2641.07
CGM^{LKH}_{500}	5.95	0.00	187.02	10.11	-0.19	2687.38	15.01	-0.45	5486.11
CGM^{LKH}_{1000}	5.95	0.00	410.29	10.10	-0.31	4392.90	14.92	-1.04	9542.22
$CGM^{rs}_{(200,10)}$	6.15	3.37	19.62	12.07	19.16	245.62	19.99	32.57	825.47
$CGM^{rs}_{(500,5)}$	6.13	2.96	36.36	11.57	14.22	546.02	19.21	27.35	1639.50
$CGM^{rs}_{(1000,10)}$	6.09	2.39	73.89	11.29	11.48	1107.66	18.92	25.48	3243.79
BS 50k	6.12	2.97	240.01	11.31	11.73	1153.01	17.86	18.43	4313.61
Gurobi	5,95	0.00	0.60	10.10	-0.29	2092.37	16.98	12.63	5337.80
GORT-GLS	5.95	0.00	10.00	10.13	0.03	100.00	15.12	0.28	300.00
LKH 1x	5.95	0.00	6.20	10.13	0.00	11.74	15.08	0.00	19.69
LKH_{adj}	5.95	0.00	89.40	10.10	-0.31	453.61	14.88	-1.30	2126.33

Table 3.1: Mean Cost, Optimality Gap and Computation Time Per Instance

neural network was trained with datasets calculated by one run of LKH, this negative optimality gap could influence the accuracy. The fact that a better optimality gap can be achieved shows the strength of this approach, that even with imperfect data the neural network manages to learn the important characteristics of the instances and can abstract them and thus outperform the heuristic used to create the data.

This holds for large instances with 100 nodes, where an optimality gap of -1.04% is achieved by CGM^{LKH}_{1000}, outperforming the LKH with one run and coming close to the optimality gap of LKH with an adjusted number of runs. This comes at the cost of long runtimes but displays the potential of neural network-guided tree searches.

3.3.4 Value Function

The value function plays a pivotal role in the MCTS, as it is utilized in every playout within each iteration of the MCTS. Hence, we undertake a comprehensive analysis of the performance of the value function concerning both accuracy and computation, while considering different beam widths as well as varying lengths of partial solutions provided as initial inputs to the value function.

Figures 3.3 and 3.4 demonstrate the relationship between the beam width and the performance of the value function, where larger beam widths yield superior results, but it is worth noting that slight fluctuations can be observed. This can be attributed to the evaluation of the scoring policy at each step of the beam search. This continuous evaluation process can potentially yield different solutions when for example using a beam width of 2 compared to the solution obtained with a beam width of 1. Although utilizing

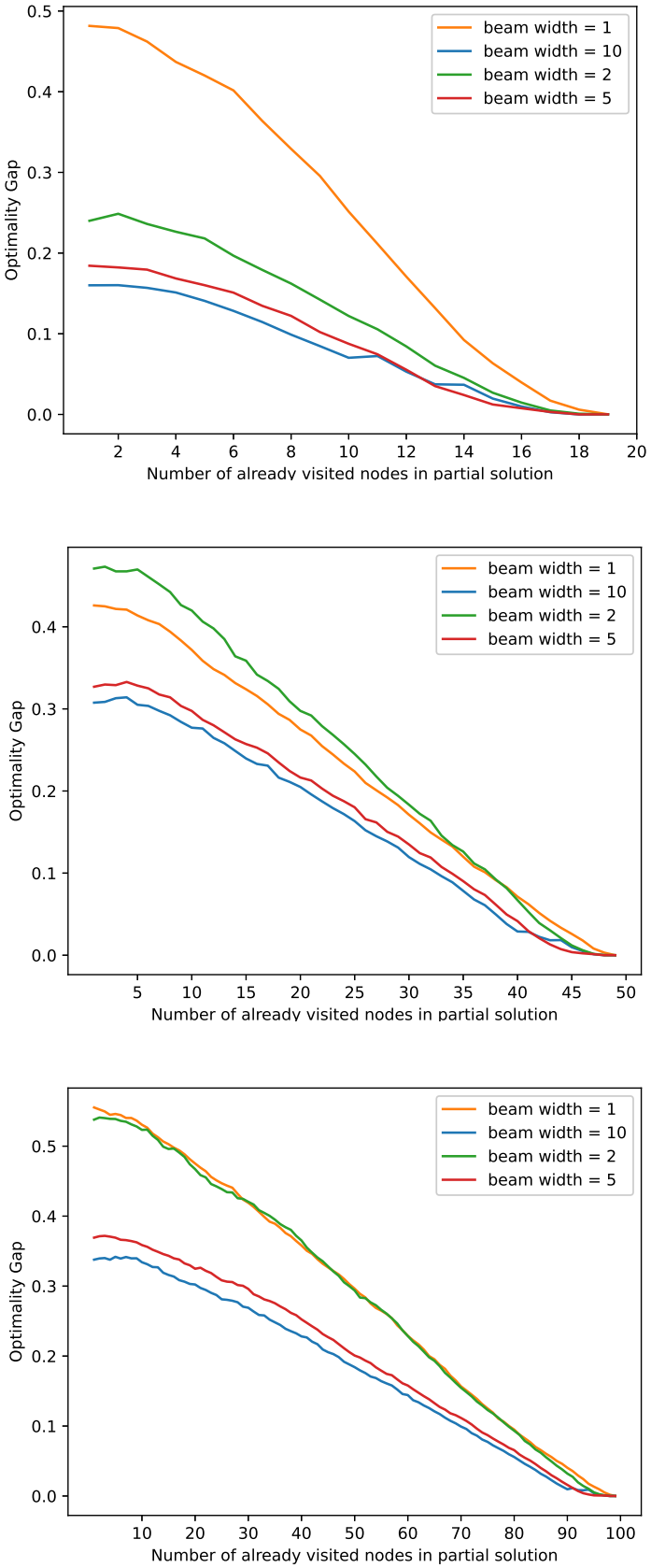


Figure 3.3: Performance of Value Function for Different Problem Sizes

		selection	expansion	simulation	backpropagation
$\text{CGM}^{\text{bs}}_{500}$	$n = 20$	0.0910	0.4371	99.4440	0.0279
	$n = 50$	0.0076	0.2669	99.7093	0.0163
	$n = 100$	0.0025	0.1375	99.8413	0.0187
$\text{CGM}^{\text{rs}}_{500}$	$n = 20$	0.1445	0.4947	99.3284	0.0324
	$n = 50$	0.0125	0.3560	99.6106	0.0209
	$n = 100$	0.0041	0.1886	99.7908	0.0164
$\text{CGM}^{\text{LKH}}_{500}$	$n = 20$	0.0092	0.0262	99.8096	0.0052
	$n = 50$	0.0009	0.0205	99.9659	0.0051
	$n = 100$	0.0004	0.0221	99.9581	0.0045

Table 3.2: Relative Running Time of the Four Phases With and Without Exact Masking Strategy

a significantly larger beam width can mitigate the mentioned fluctuations, it comes at the cost of substantially increased computation time. However, since our beam search primarily serves as an assistance for the MCTS algorithm, minor fluctuations in accuracy for different small beam widths are negligible. Balancing runtime and accuracy is crucial for our approach, with the value function being the primary contributor to overall runtime (cf. Section 3.3.5). Figure 3.4 illustrates that the computation time associated with beam width 10 is relatively higher compared to the other beam widths. Consequently, we have opted to utilize a beam width of 5 for our primary experiments, as it has a favorable balance between prediction accuracy and computation time.

3.3.5 Computational Runtime Analysis

Table 3.2 provides an overview of the time distribution across the four phases in the $\text{CGM}^{\text{bs}}_{(500,5)}$ algorithm. Nearly all (over 99 %) of the computational time is consumed by the simulation phase. It needs to be mentioned that the computational time of the simulation phase is based on the runtime of the beam search as well as the evaluation of the neural network that is called for each simulation. In relative terms, the MCTS with a relaxed simulation phase remains time-intensive, but in absolute terms it is significantly faster. This suggests the ability of our MCTS heuristic to reduce the computation time by refining the value function.

3.3.6 Analysis of Predictions

We further conduct experiments where we distribute the p_{ij} values uniformly to obtain insights into how these predictions enhance the performance of the MCTS. Table 3.3 compares the performance of CGM^{LKH} for 200 to 1000 playouts with predictions p_{ij} obtained by the neural network to the performance of the MCTS approach without neural network-assisted decisions. The CCGCN predictions enhance the effectiveness of the MCTS, showing that the neural network is able to capture non-trivial instance characteristics and

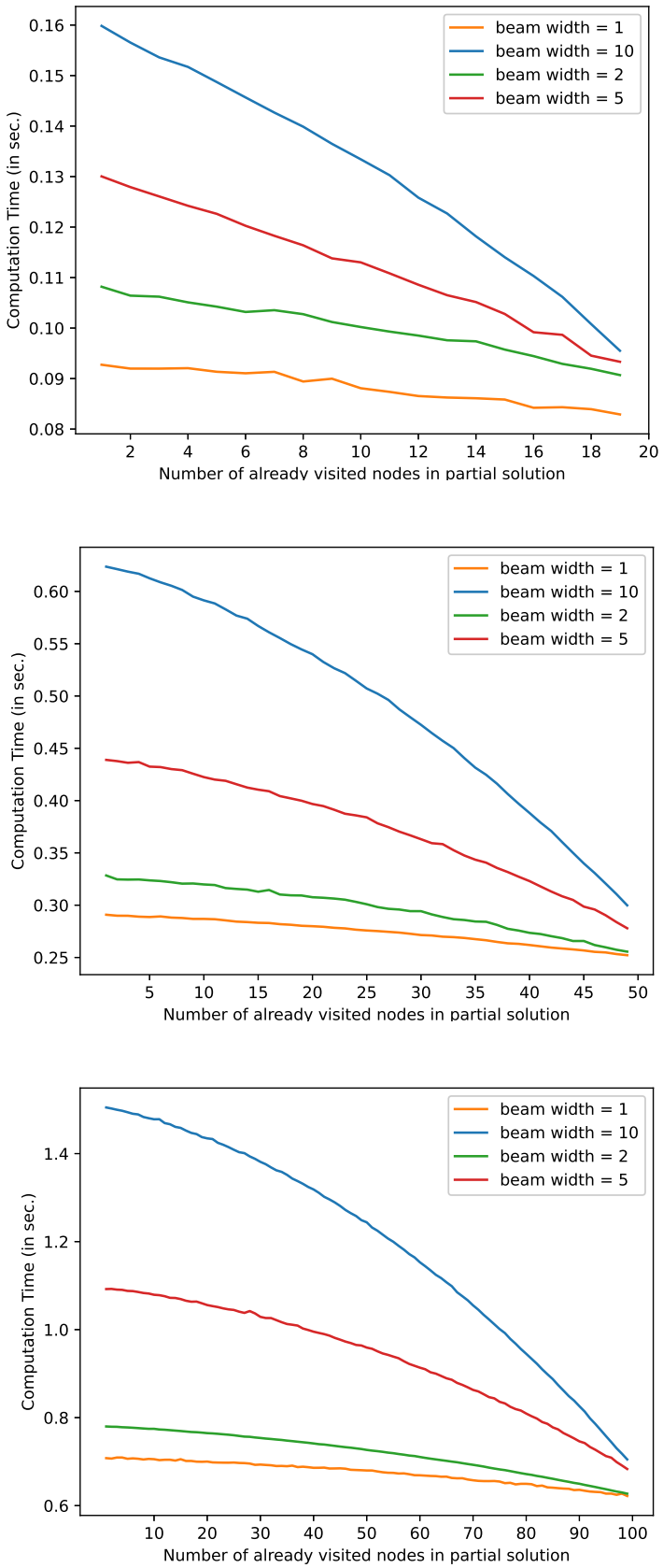


Figure 3.4: Calculation Time of Value Function for Different Problem Sizes

predictions	playouts	$n = 20$	$n = 50$	$n = 100$
GCN	200	5.95	10.12	15.04
	500	5.95	10.11	15.01
	1000	5.95	10.10	14.92
uniformly distributed	200	6.07	10.42	15.58
	500	6.01	10.35	15.49
	1000	5.98	10.32	15.37

Table 3.3: Mean Cost of CGM^{LKH} with and without Predictions Given by the GCN

demonstrating the possibility for improvement of classical approaches with neural network assistance.

3.4 Discussion

In this chapter, we presented a MCTS approach to solving the CVRPTW by using a deep neural network to guide the search procedure within the MCTS and balance the trade-off between exploration of the search tree and exploitation of the most promising paths within the tree. Furthermore, we applied the beam search, which was developed in Chapter 2, as the value function during the simulation phase of the MCTS. By using the iteratively built partial solutions during the search as a secondary input for the context-complemented graph convolutional network, we are able to incorporate the context of these partially developed solutions, specifically addressing the challenging time windows and capacity constraints, into the neural network's predictions of promising edges to guide the search process in the search tree. Encouraging results were achieved for smaller instances, demonstrating the potential of approaches that combine MCTS with AI techniques for routing problems with hard side constraints. We conducted tests with various hyperparameter configurations to assess their influence on the trade-off between computation time and solution quality. Computational results showed that the neural network-guided search was able to outperform the beam search as well as the heuristic used for creating the training data, demonstrating the network's ability to learn non-trivial characteristics from imperfect training data. An ablation study was conducted to show how the network's predictions improve the effectiveness of MCTS, demonstrating the possibility of improvement over classical approaches through neural network support. We further tested the concept of relaxed simulations during the simulation phase of the MCTS for the beam search value function to improve computation times. By allowing the neural network to handle hard constraints implicitly during the beam search by incorporating context into its predictions, the time spent in the simulation phase can be significantly reduced, while still obtaining high-quality solutions for small instances. However, for larger instances, this approach revealed its shortcomings, as inaccurate predictions by the neural network lead the MCTS into unpromising areas of the search tree.

Efficiently traversing a search tree for complex problems such as the CVRPTW re-

mains challenging. We have demonstrated that deep learning can accelerate the search process within the tree by predicting promising paths. However, to build solutions more efficiently and tackle problems like the CVRPTW, alternative techniques need to be explored. Quantum computing holds the promise of solving optimization problems faster than current technologies through specialized hardware, which in theory can perform some calculations exponentially faster than any modern classical computer. Although the current state of quantum computing hardware is not yet capable of handling optimization problems with more than a few variables, it is essential to explore theoretical methods for utilizing quantum computing hardware for combinatorial optimization problems. This ensures that when the hardware becomes widely available, the theoretical frameworks will be ready for implementation. For this reason, the next chapter is dedicated to investigating ways to leverage quantum computing for solving the CVRPTW.

Chapter 4

Solving the CVRPTW with Quadratic Unconstrained Binary Optimization

In recent years, specialized hardware has been developed to address combinatorial optimization problems, including quantum computers. Oftentimes, quantum and quantum-inspired algorithms are designed to solve quadratic unconstrained binary optimization (QUBO) problems, which are equivalent to Ising models, playing a significant role in physics [Luc14]. Many combinatorial optimization problems can be formulated as QUBOs using special reformulation techniques, hence, it is increasingly intriguing to explore ways to utilize specialized hardware for solving combinatorial optimization problems formulated as QUBOs. However, since these hardware systems can handle only a limited number of variables, it is essential to develop new scalable methods for integrating this hardware into a solving framework. This will enable us to efficiently solve larger instances despite the hardware limitations.

For this, in Section 4.1 we will formulate the CVRPTW as a QUBO problem and experiment with solving this formulation using specialized quantum-inspired computing hardware. We will apply a learned problem reduction to decrease the number of variables required to represent the CVRPTW as a QUBO. This reduction aims to solve larger instances on this hardware more effectively and to achieve better results.

In Section 4.2, we will develop a heuristic consisting of three phases that also uses quadratic unconstrained binary optimization to solve the CVRPTW. This heuristic is designed to leverage the strengths of QUBO hardware solvers, namely their efficiency in handling quadratic binary optimization problems with a lower number of variables. This approach aims to avoid the issues identified in Section 4.1 that arise from directly solving the CVRPTW as a QUBO.

We will conclude this chapter with Section 4.3, which presents some ideas and remarks on how to integrate QUBO solvers within a branch-and-bound framework to solve the CVRPTW exactly using quantum(-inspired) computing hardware.

4.1 Learned Problem Reduction to Solve the CVRPTW Using Quantum-Inspired Computing Hardware

Building on the work of [SGM22], we develop a new formulation for the CVRPTW as a quadratic unconstrained binary optimization problem. We work with the MILP formulation (1.5)-(1.15) presented in Section 1.3.1 based on edge representations, as this approach is best suited to benefit from our learning-based problem reduction. We derive a novel QUBO formulation for the CVRPTW and asymptotically calculate the number of variables needed to model this formulation. We then solve the CVRPTW on hardware specialized for solving QUBO problems, as described in Section 1.3.3. This section is based on the published work [Dor23].

4.1.1 QUBO Formulation of the CVRPTW

The CVRPTW imposes time window and capacity constraints that are expressed as inequality constraints. Generally speaking, inequalities are known to be particularly difficult to handle within QUBO models. In the following, we will see how to deal with inequalities in the context of QUBOs.

Recall from Section 1.3.2 that a QUBO is of the form

$$\min_{x\in\{0,1\}^{\tilde{n}\times\tilde{n}}} f = x^T Q x, \tag{4.1}$$

with $Q \in \mathbb{R}^{\tilde{n}\times\tilde{n}}$ being an upper triangular matrix and $x \in \{0,1\}^{\tilde{n}}$ being the binary decision vector. The goal is to find the vector x^* that minimizes f. Furthermore, recall that the MILP for the CVRPTW is of the form

$$\begin{aligned}
\min \quad & \sum_{i,j=0}^{n} c_{ij}x_{ij} \qquad \text{s.t.} && && (4.2)\\
& \sum_{i=0}^{n} x_{ij} = 1 && \forall j \in [n] && (4.3)\\
& \sum_{i=1}^{n} x_{0i} - \sum_{j=1}^{n} x_{j0} = 0 && && (4.4)\\
& \sum_{i=0}^{n} x_{ih} - \sum_{j=0}^{n} x_{hj} = 0 && \forall h \in V && (4.5)\\
& y_j \geq y_i - d_j x_{ij} - Z(1 - x_{ij}) && \forall i,j \in V, i \neq j && (4.6)\\
& 0 \leq y_i \leq Z && \forall i \in V && (4.7)\\
& s_j \geq s_i + c_{ij}x_{ij} - M(1 - x_{ij}) && \forall i,j \in V, i \neq j && (4.8)\\
& a_i \leq s_i \leq b_i && \forall i \in V && (4.9)\\
& \sum_{j=1}^{n} x_{0j} \leq K && && (4.10)
\end{aligned}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in V, i \neq j \tag{4.11}$$
$$s_i, y_i \in \mathbb{Z}_+ \qquad \forall i \in [n]. \tag{4.12}$$

Note that the number of vertices n does not equal the dimension of our binary decision vector x in (4.1), as the number of variables $\tilde{n}$ in the QUBO formulation includes binary variables representing the side constraints that are represented as integer variables in the MILP (4.2)-(4.12). In the following, we will elaborate on how to convert this MILP to a QUBO.

To obtain a QUBO formulation from this MILP, we have to transform this MILP into the general form

$$\min \quad \sum_{i=1}^{k} c_i z_i \qquad \text{s.t.} \tag{4.13}$$
$$\sum_{i=1}^{k} a_i z_i = b \tag{4.14}$$
$$z_i \in \{0,1\}, \tag{4.15}$$

as described in Section 1.3.2. Therefore, we have to change the inequalities (4.6) to (4.10) to equalities by introducing slack variables for each inequality. As an example, Inequality (4.6) can be rewritten as

$$y_i - y_j - d_j x_{ij} - Z(1 - x_{ij}) \leq 0.$$

By introducing an integer slack variable $\lambda_{i,j}^{4.6}$ for each edge (i,j), Inequality (4.6) is equivalent to

$$y_i - y_j - d_j x_{ij} - Z(1 - x_{ij}) + \lambda_{i,j}^{4.6} = 0.$$

This is done for all inequalities from (4.6) to (4.10), resulting in the inclusion of $\mathcal{O}(n^2)$ new integer variables. Note that the constraints in (4.7) and (4.9) are two inequalities per node, therefore two variables are needed for the lower and upper bound, respectively, denoted as $\lambda_{i,0}^{4.7}$ for the lower bound and $\lambda_{i,1}^{4.7}$ for the upper bound, respectively, and likewise for Constraint (4.9).

Next, we have to convert each integer variable, including the newly introduced slack variables, into its binary representation, as detailed in Section 1.3.2. Suppose we have an integer variable z that can take values between 0 and 16. To represent z in a binary form, we introduce a sufficient number of binary variables $z_m^b \in \{0,1\}$ such that each value between 0 and 16 can be represented by a combination of these binary variables. In this case, the binary representation of z would be

$$z = \sum_{m=0}^{4} 2^m z_m^b = z_0^b + 2z_1^b + 4z_2^b + 8z_3^b + 16z_4^b.$$

By taking into account the lower and upper bounds for each integer variable, we can lower the number of binary variables needed to represent the integer variable by applying the

binary expansion function B, defined in Equation (1.31). To do this, we have to define upper bounds for each newly included integer slack variable. For slack variable $\lambda_{i,j}^{4.6}$ for Inequality (4.6), Formula (1.32) gives us

$$\begin{aligned}\lambda_{i,j}^{4.6} \leq & -\min\{y_i^\ell, y_i^u\} - \min\{-y_j^\ell, -y_j^u\} \\ & - \min\{-d_j x_{ij}^\ell, -d_j x_{ij}^u\} - \min\{x_{ij}^\ell, x_{ij}^u\} + Z \\ \leq & \, 2Z + d_j.\end{aligned}$$

For the slack variable $\lambda_{i,j}^{4.8}$ for Constraint (4.8) we have

$$\begin{aligned}\lambda_{i,j}^{4.8} &\leq -\min\{s_i^\ell, s_i^u\} - \min\{-s_j^\ell, -s_j^u\} - \min\{c_{ij}x_{ij}^\ell, c_{ij}x_{ij}^u\} - \min\{x_{ij}^\ell, x_{ij}^u\} + M \\ &\leq -a_i + b_j + M.\end{aligned}$$

The slack variable $\lambda^{4.10}$ for Constraint (4.10) can clearly be bounded by K and the lower bounds for the slack variables $\lambda_{i,0}^{4.7}$ and $\lambda_{i,0}^{4.9}$ and upper bounds for the slack variables $\lambda_{i,1}^{4.7}$ and $\lambda_{i,1}^{4.9}$ are given by the constraints.

Now we can state the full quadratic binary polynomial f_Q^{CVRPTW} for modeling the CVRPTW. Applying the binary expansion function B (1.31) with the defined upper bounds for our integer variables and slack variables, the function can be stated as

$$f_Q^{\text{CVRPTW}} = P_1 \cdot f_{\text{obj}} + P_2 \cdot f_{\text{route}} + P_3 \cdot f_{\text{cap}} + P_4 \cdot f_{\text{tw}}, \tag{4.16}$$

with

$$\begin{aligned}
f_{\text{obj}} = & \sum_{i,j=0}^{n} c_{ij}x_{ij}, \\
f_{\text{route}} = & \sum_{j=1}^{n}\left(\sum_{i=0}^{n} x_{ij} - 1\right)^2 + \left(\sum_{i=1}^{n} x_{0,i} - \sum_{j=1}^{n} x_{j,0}\right)^2 + \sum_{h\in V}\left(\sum_{i=0}^{n} x_{i,h} - \sum_{j=0}^{n} x_{h,j}\right)^2, \\
f_{\text{cap}} = & \sum_{i\in V}\sum_{j\in V}\left(B(y_i,0,Z) - B(y_j,0,Z) - d_j x_{ij} - Z(1-x_{ij}) + B(\lambda_{i,j}^{4.6},0,2Z+d_j)\right)^2 \\
& + \sum_{i=1}^{n}\left(B(y_i,0,Z) - B(\lambda_{i,0}^{4.7},0,Z)\right)^2 \\
& + \sum_{i=1}^{n}\left(B(y_i,0,Z) + B(\lambda_{i,1}^{4.7},0,Z) - Z\right)^2 \\
& + \sum_{i=1}^{n}\left(B(s_i,a_i,b_i) - a_i - B(\lambda_{i,0}^{4.9},a_i,b_i)\right)^2 \\
& + \sum_{i=1}^{n}\left(B(s_i,a_i,b_i) - b_i + B(\lambda_{i,1}^{4.9},a_i,b_i)\right)^2 \\
& + \left(\sum_{j=1}^{n} x_{0,j} + B(\lambda^{4.10},0,K) - K\right)^2,
\end{aligned}$$

$$f_{\text{tw}} = \sum_{i\in V}\sum_{j\in V}\Big(B(s_i,a_i,b_i) - B(s_j,a_j,b_j) + c_{ij}x_{ij} - M(1-x_{ij}) + B(\lambda_{i,j}^{4.8},0,-a_i+b_j+M)\Big)^2.$$

The penalty constants $P_i \in \mathbb{R}_{\geq 0}, i \in [4]$, have to be adjusted accordingly, such that a violation of the constraints in f_{route}, f_{cap} or f_{tw} results in a larger increase of the function value than the decrease it might produces in the value of f_{obj}. Selecting penalty constants for the constraints greater than the optimal tour length provides a theoretical assurance that the QUBO solution with the lowest energy corresponds to a feasible solution. Computational experiments have shown that the performance from the QUBO hardware solver we applied is best when choosing the penalty constants sufficiently large, but not arbitrarily large. Fujitsu's DA [MTM+20] offers the functionality to automatically adjust the penalty coefficients during the optimization process, a description of that process is found in Section 1.3.3.

Utilizing the lower and upper bounds of integer variables for the binary expansions allows the prevention of excessively large coefficients for the newly introduced binary variables in the majority of cases. However, using these binary expansions is still problematic as it can cause difficulties for solvers to find the correct assignments [VPD19]. For example, if an integer variable's value needs to be changed from 16 (binary encoding: 10000) to 15 (binary encoding: 01111) during the optimization process, five bit switches are required. This becomes increasingly more challenging as the values of the integer variables increase, as it leads to large coefficients for the binary variables, further complicating the process of finding the correct assignment for each binary variable. In Section 4.1.3 we will see the impact of an increasing number of binary expanded variables on the solver's ability to identify solutions.

We are now able to determine the number of variables required for our QUBO formulation of the CVRPTW. For the binary representation of the integer and slack variables, let us look at Equation (1.31) again. For each integer variable z, the number of new binary variables added to the model is exactly

$$k_z = \lceil \log_2(z^u - z_l + 1)\rceil.$$

The integer variables s_i, y_i and slack variables $\lambda_{i,j}^{4.6}, \lambda_i^{4.7}, \lambda_{i,j}^{4.8}, \lambda_i^{4.9}, \lambda^{4.10}$ therefore require at most $\lceil \log_2(\max_i b_i - a_i + 1)\rceil$, $\lceil \log_2(Z+1)\rceil$, $\lceil \log_2(\max_i 2Z + d_i)\rceil$, $\lceil \log_2(\max_{ij} -a_i + b_j + M)\rceil$ and $\lceil \log_2(K+1)\rceil$ binary variables, respectively. If we define δ to be

$$\delta := \lceil \log_2(\max\{\max_{ij}\{-a_i+b_j+M\}, \max_i\{2Z+d_i\}, K+1\})\rceil, \tag{4.17}$$

then for every integer and slack variable at most δ binary variables are required for the binary encoding. Overall, we need $\mathcal{O}(n^2)$ variables to represent the x_{ij}. For the integer variables s_i, y_i we have $\mathcal{O}(n\delta)$ variables for the binary encoding. Since we have $\mathcal{O}(n^2)$ inequalities, we need $\mathcal{O}(n^2\delta)$ additional slack variables in binary form. Thus, in total $\mathcal{O}(n^2 + n^2\delta)$ variables are required for our QUBO formulation of the CVRPTW.

As pointed out earlier, some variables can be removed beforehand to lower the total number of variables needed. This holds for example if $a_i + c_{ij} > b_j$ for some $i, j \in V$ or we do not have a fully connected graph to begin with. In the first case, we can simply set $x_{ij} = 0$. In the latter case, the overall number of variables needed is reduced to $\mathcal{O}(|E| + |E|\delta)$.

4.1.2 Sparsity of Graphs

As the number of binary variables in the QUBO formulation of the CVRPTW increases rapidly due to the need for slack variables and binary representations of integer variables, as discussed in Section 4.1.1, our goal is to reduce the size of a specific problem instance. We do this by applying a threshold to the edge probabilities generated by the graph convolutional neural network presented in Chapter 2. This threshold should be set to sufficiently reduce the instance size, enabling the DA to handle larger instances while still finding good solutions. Therefore, we first have to evaluate the trade-off between the value of the probability threshold and the number of deleted edges as well as the number of remaining binary variables and inequalities and the resulting instance size.

Table 4.1 shows the mean number of deleted edges over 10000 instances for the different problem sizes and edge probability thresholds. One can see that even with a comparatively large threshold of 0.001, the neural network confidently excludes most of the edges. The percentage number of deleted edges for the same threshold increases with the problem sizes, since for larger graphs the percentage of edges irrelevant for the solution increases. A threshold of 0.00001 excludes at least 3/4 of the edges, for graphs with 100 nodes the neural network excludes all but 10 percent of the edges, which reduces the number of edges included in the graph from 10201 to just under 1000 edges on average.

	Problem size					
	$n = 20$		$n = 50$		$n = 100$	
threshold p	del. edges	%	del. edges	%	del. edges	%
10^{-3}	306.73	69.55	2169.42	83.41	9248.41	90.66
10^{-4}	279.42	63.36	2059.04	79.16	9019.67	88.42
10^{-5}	254.58	57.73	1966.48	75.60	8784.17	86.11

Table 4.1: Number and Percentage of Deleted Edges for Different Problem Sizes and Thresholds

These sparse graphs act as a learned problem reduction [SELW20]. Table 4.2 analyzes the impact of the problem reduction on the size of the instances for the DA. For the number of binary variables needed to represent integer variables, for our datasets an upper bound for δ in Equation (4.17) is given as $\delta = 10$. Note however that we can reduce the number of auxiliary binary variables needed by taking into account the upper and lower bound of each integer variable individually before initializing the auxiliary binary variables. As shown in Table 4.2, the learned problem reduction allows us to reduce the size of our instances significantly.

	Problem size								
	$n = 20$			$n = 50$			$n = 100$		
number of bits x	441			2601			10201		
number of bits s	200			500			1000		
number of bits y	200			500			1000		
inequalities	841			5101			20201		
threshold p	10^{-3}	10^{-4}	10^{-5}	10^{-3}	10^{-4}	10^{-5}	10^{-3}	10^{-4}	10^{-5}
deleted bits x	307	279	255	2169	2059	1966	9248	9020	8784
deleted ineqs	613	559	509	4339	4118	3933	18496	18039	17568
remaining bits	534	562	586	1432	1542	1635	2953	3181	3417
remaining ineqs	228	282	332	762	983	1168	1705	2162	2633

Table 4.2: Estimation of Instance Sizes for Digital Annealer after Applying Learned Problem Reduction

In the following, we will test how this affects the solving process of the DA in terms of solution quality.

4.1.3 Computational Experiments

Experiment Setup

For Fujitsu's DA we use the following parameters: In the energy function (4.16), P_1 is set to one and the penalties P_2, P_3 and P_4 are automatically incrementally adapted to the optimal value. This is done by multiplying them by a factor of 1.5, if the result, i.e. the objective value f_Q^{CVRPTW}, did not improve for 500 iterations. That way, the DA focuses on satisfying the side constraints before minimizing the objective f_{obj}. The time limit for the optimization process is set to one second per two bits. The selection of the parameter values for the automatic adaption is based on empirical experiments, which demonstrate that the parameters provide a good balance between the performance of the DA and the computational resources required by the optimization process. This amounts on average to a time limit of roughly 300 seconds for instances with 20 nodes and 800 seconds for the 50 node instances, since we set the time limit for the optimization process to one second per two bits and on average the number of remaining bits is around 600 and 1600 for the two problem sizes, respectively, as can be seen in Table 4.2. As the amount of time to run experiments on the DA was limited, we conducted the experiments on a smaller dataset only consisting of the Solomon benchmark instances from the set R [Sol87], which was also the baseline distribution for our data generation. The dataset consists of 115 and 46 instances for the problem sizes with 20 and 50 customers, respectively.

The preprocessing for the model is done on a machine with an Intel Core i7-8650U CPU @ 1.90GHz and one Nvidia Geforce GTX 1050 with 2GB RAM, the optimization is executed on Fujitsu's DA hardware [MTM+20].

Model	Problem size					
	$n = 20$			$n = 50$		
	cost	gap	time	cost	gap	time
GCNDA 10^{-3}	3.87	15.83	107.71	12.57	108.48	901.54
GCNDA 10^{-4}	3.92	17.33	113.52	12.32	104.37	907.40
GCNDA 10^{-5}	3.96	18.38	113.81	12.43	106.21	920.86
DA	3.97	18.70	116.52	12.33	104.48	914.58
$\text{CGM}^{\text{bs}}_{500,5}$	3.38	1.26	57.91	6.56	8.85	752.17
GCNBSSTH 10K	3.37	0.69	33.57	6.31	4.70	208.58
Gurobi	3.35	0.00	0.85	6.03	0.00	1962.92
GORT-GLS	3.34	0.00	10.00	6.04	0.03	30.00
LKH	3.34	0.00	7.67	6.03	0.00	12.04

Table 4.3: Mean Cost, Optimality Gap and Computation Time Per Instance

Computational Results

Table 4.3 shows the results of our experiments for solving the QUBO formulation (4.16) of the CVRPTW with the DA, which is called GCNDA in the following, compared to results obtained by our beam search heuristic from Chapter 2 with a beam width of 10000 and our Monte Carlo Tree Search approach from Chapter 3 with 500 rollouts and a beam width of 5 to have a similar runtime as the DA, as well as the baseline models, as described in Section 2.3.2. The different applied edge probability thresholds are stated behind the model name GCNDA. The model DA refers to solving our developed QUBO formulation (4.16) on the DA without applying the learned problem reduction.

The performance of the GCNDA method is inferior to that of GCNBSSTH and especially the baseline models, in terms of both run time and quality of results. While for instances with 20 nodes, the DA is able to find solutions with an optimality gap between 15.8% for the reduced instance with an edge probability threshold of 0.001 and 18.7% without the reduction, it has its difficulties for larger instances with 50 nodes to find a good solution, only finding solutions twice as costly as the best solution. Comparing the computation time between the DA and CPU approaches is difficult, as the DA lacks a termination criterion, meaning it continues to optimize until the given time limit is reached. This can lead to variable computation times and potentially idle periods, affecting the reported computation time. Given the substantial differences in underlying technology, making a direct comparison to CPU times is difficult.

However, the results from Table 4.3 demonstrate the effectiveness of the learned problem reduction method in improving solution quality by reducing the size of the considered graph for smaller instances. A performance improvement of around 3%, while also accelerating the optimization process, is a promising result. It is reasonable to assume that the decrease in computation time becomes more obvious if a higher time limit is given. As seen in Table 4.2, the higher the edge probability threshold used in the reduction, the fewer

integer variables are required. The risk of the best solutions not being obtained due to the removal of important edges during the reduction process is found to be negligible within the range of solution gaps presented in Table 4.3. As the size of the CVRPTW problem increases, the optimization of the number of integer variables in their binary representation becomes increasingly challenging. Since the learned problem reduction only applies to the decision variables x, whereas the number of integer variables y and s is not reduced, the effect of the reduction for larger instances is negligible.

The difficulties the DA encounters in finding good solutions, especially for larger instances, can be attributed to the following issue. Suppose that node i has the time window $[128, 255]$, which means the integer variable s_i that represents the arrival time of a vehicle at node i, can take values between 128 with a binary encoding of 10000000 and 255, with a binary encoding of 11111111. Furthermore, suppose that the correct solution includes visiting node j after node i, with j having a time window of $[300, 400]$ and the travel time c_{ij} between i and j being negligible. This means that for the best solution, the value of s_i is not relevant as long as it is in the range of $[128, 255]$. But the DA does not know that and may try to optimize s_i by going through all possible bit switches, in this case, $2^7 = 128$, and checking whether the objective value improves. As the number of integer variables that keep track of the side constraints increases significantly for larger instance sizes, the DA has problems finding good solutions. Therefore, given the current version of the DA and the resources available for our experimental study, the limitations of the approach of solving the CVRPTW directly on the DA have been reached.

Figure 4.1 visualizes the optimization process by the DA for two different instances. The x-axis refers to the solution time and the two y-axes indicate the energy of the objective function and the constraints (penalties), respectively. The DA starts with a condition where the energy of the objective function is zero and the energy of the penalties is positive. Then the solution is optimized by first decreasing the energy of the penalties to zero, which translates to finding a valid solution, and implies increasing the energy of the objective function. Then, the energy of the objective function is increased while the energy of the penalties remains at zero.

Discussion

The application of a deep neural network as a learned problem reduction for solving the CVRPTW via quadratic unconstrained binary optimization on Fujitsu's DA demonstrated the potential of using learned problem reductions in improving solution quality and reducing computation time on quantum-inspired computers. However, the large size of CVRPTW instances and the requirement for a significant number of integer variables to model the problem as a quadratic unconstrained binary optimization problem make it challenging to optimize for larger instances, even with the reduction of the instance size.

Therefore, in the following section, we will develop a more sophisticated heuristic that also utilizes quadratic unconstrained binary optimization, but avoids solving QUBO formulations of optimization problems including large integer variables as well as the handling of the time window and capacity constraints on the QUBO solving hardware.

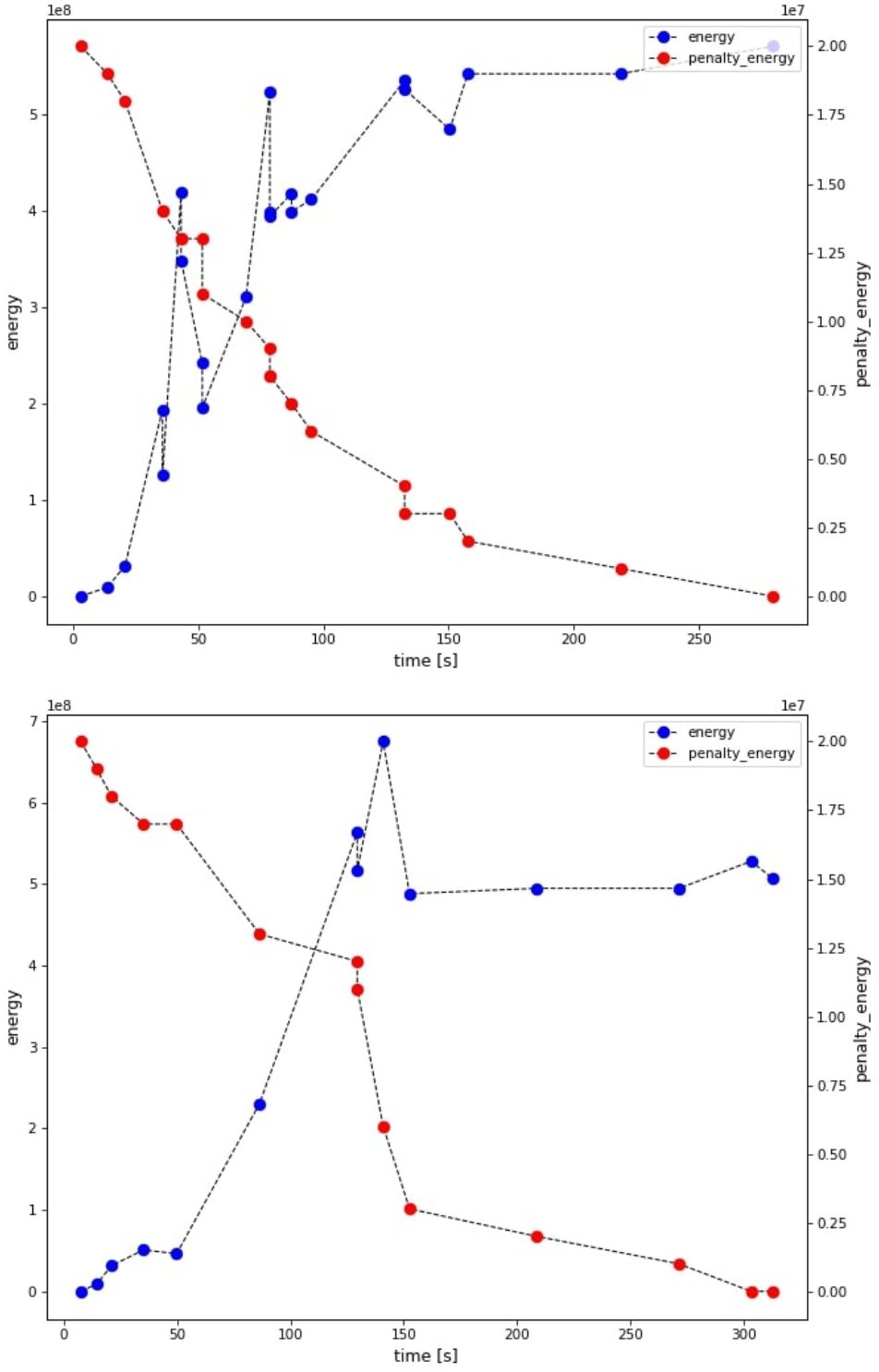


Figure 4.1: Visualization of the Optimization Process by the Digital Annealer for Two Different Instances

4.2 Three-Phase Heuristic

When solving the CVRPTW with quantum(-inspired) computing, we have seen in the previous section that a direct solution is impractical for larger instances, mainly because of the number of slack variables required to transform integer variables into binary variables and to transform inequalities into equations very quickly becomes too large to be handled efficiently by the current hardware. The applied learned problem reduction in Section 4.1 based on the predictions of the deep neural network led to performance improvements for smaller instances, but the limits of the hardware were quickly reached as the problem size increased, which is mainly caused by the side constraints of the CVRPTW, the capacity and time window constraints, which necessitate integer variables as well as inequalities.

On the other hand, it has also been demonstrated in the previous chapters that the graph convolutional deep neural network developed in Section 2.1 is well suited to extract the important properties from the graph representations to make high-quality predictions about the set of edges contained in the solution while taking the side constraints into account. The presented approaches, the beam search and the Monte Carlo Tree Search that utilize these predictions show that these predictions can be used to produce high-quality solutions.

In this section, we want to overcome the disadvantages of these two approaches, namely the lack of effective scalability to larger instances, and provide certain scalability so that even relatively large instances can also be solved to near optimality.

For this, we propose a hybrid approach to approximately solve the CVRPTW consisting of three individual phases. The proposed 3-phase heuristic combines quadratic unconstrained binary optimization hardware with a deep learning-assisted heuristic within a framework that offers scalability to larger instances. The goal is to leverage the strengths of QUBO-solving hardware while ensuring that the hardware limits are not exceeded. For this, we first use a deep learning-complemented QUBO formulation to cluster the set of vertices into subsets, generate sets of feasible candidate routes for each cluster employing the beam search that is then combined into a complete solution by a quadratic unconstrained binary set partition problem. The QUBOs are solved using Fujitsu's Digital Annealer (DA) [MTM+20]. We demonstrate that this 3-phase heuristic provides scalability to larger-sized instances and produces optimal results for smaller instances as well as near-optimal solutions for large instances while ensuring feasibility with respect to capacity and time window constraints.

4.2.1 Structure of the Three-Phase Heuristic

We have seen in Section 4.1 that solving the CVRPTW formulated as a QUBO directly using quantum(-inspired) computing hardware is not a feasible option because the number of binary variables required to represent an instance quickly surpasses the capabilities of dedicated hardware like the DA. Therefore, we propose a version of the *Cluster First, Route Second* method [DMR14] to solve the CVRPTW utilizing QUBO-solving hardware. Our 3-phase heuristic aims at leveraging the strengths of the DA [MTM+20] as well as the

graph convolutional neural network while providing scalability to larger instances. The three phases can be described as follows:

1. **Split** the set of vertices into a number of k clusters by solving a cluster optimization problem, complemented with weights given by a deep neural network, formulated as a quadratic unconstrained binary optimization problem on the DA. Repeat this for different values of k to diversify the set of clusters.

2. **Generate** candidate routes on each of the found clusters by applying the beam search (cf. Chapter 2) to create a set of different feasible candidate routes for the CVRPTW restricted to that cluster.

3. **Solve** a Set Partition Problem (SPP) [TV14] formulated as a QUBO on the DA to compose a complete solution by choosing a subset of the set of candidate routes generated in Phase 2 minimizing the total travel times.

This approach offers the advantage of significantly reducing the size of the QUBOs solved by the DA compared to solving a CVRPTW instance directly on the DA. Additionally, we can avoid the complexities associated with inequalities, as it turned out to be challenging for the DA in our case, as well as integer variables with large values by outsourcing the handling of those constraints to the heuristic employed in Phase 2. The number of decision variables is $\mathcal{O}(n)$ for the Clustering and $\mathcal{O}(n)$ for the SPP. The heuristic applied in Phase 2 is highly flexible, as it does not attempt to solve each cluster as a single-route instance but instead finds the best set of routes within each cluster and can build different sets of routes for each cluster in parallel. This flexibility allows us to vary the number of clusters in Phase 1 by a parameter λ that we call *cluster range*, forcing different cluster assignments and thereby creating a more diverse set of candidate routes for inclusion in Phase 3. In this way, we leverage the strengths of the DA, particularly its ability to handle binary quadratic unconstrained optimization problems with a manageable number of binary variables and delegate the handling of the hard capacity and time window constraints to a heuristic for more efficient processing. Algorithm 4 gives a high-level overview of the 3-phase heuristic. In the following sections, we describe each of the three phases in more detail.

4.2.2 Vertex Clustering

In each iteration of Phase 1, we split the set of vertices into a varying number of k subsets to vary the assignment of vertices to the clusters and subsequently obtain a more diverse set of candidate routes in Phase 2. To achieve this, we bound k from above by the minimal number of vehicles needed to solve the instance as a number of clusters exceeding this bound could potentially exclude the overall optimal solution to this instance by forcing more routes than necessary. The bound $\bar{k}$ is given as the sum over all demands divided by the capacity of a vehicle:

$$\bar{k} = \left\lceil \frac{\sum_{v \in V} d_v}{Z} \right\rceil .$$

Algorithm 4: 3-Phase Heuristic

Input: Graph $G = (V, E)$, upper bound $\bar{k}$ for the number of clusters, number of cluster range λ
Output: solution S

Initialize $C = \emptyset$
Evaluate neural network N for graph G to obtain edge probabilities p
for k in $\left[\bar{k} - \lambda, \ldots, \bar{k}\right]$ **do**
    clusters $\leftarrow$ `SolveClusterQUBO` (G, p, k)
    $C \leftarrow C \cup$ clusters
end
Initialize $R = \emptyset$
for c in C **do**
    routes $\leftarrow$ `BeamSearch` (c, p)
    $R \leftarrow R \cup$ routes
end
$S \leftarrow$ `SolveSPPQUBO` (R)
return S

We define our decision variables x_{vm} to be

$$x_{vm} = \begin{cases} 1 & \text{if vertex } v \text{ is assigned to cluster } m \\ 0 & \text{else.} \end{cases} \tag{4.18}$$

With this, we can formulate the clustering problem as a quadratic integer program as follows:

$$\begin{aligned} \min \quad & \sum_{m=1}^{k} \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{im} x_{jm} \quad \text{s.t.} \\ & \sum_{m=1}^{k} x_{im} = 1 \quad \forall i \in V, \\ & x_{im} \in \{0, 1\} \quad \forall i \in V, m \in [k], \end{aligned} \tag{4.19}$$

where we minimize the sum of travel times between all vertex pairs in each cluster under the constraint that each vertex belongs to exactly one cluster. The equivalent QUBO formulation is given as:

$$\min \quad \sum_{m=1}^{k} \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{im} x_{jm} + P \left(\sum_{i=1}^{n} \left(\sum_{m=1}^{k} x_{im} - 1 \right)^2 \right), \tag{4.20}$$

where the constraint is added as a penalty term with a penalty factor P. This QUBO is subsequently solved using the DA. In contrast to other formulations, we do not include

inequality constraints ensuring that the sum of the demands of each cluster does not exceed the capacity of the vehicles as well as taking into account the time windows within the clusters, as we do not solve each cluster with one route. This aligns with the strengths of the DA, as inequalities and integer variables cause difficulties. The purpose of our clustering approach is to split the instance into smaller instances to heuristically generate routes on these subsets to be included in the SPP.

We exclude the depot node from this formulation. Including the depot node only once fails to account for the fact that each cluster is solved as an independent CVRPTW instance in phase two. Consequently, we would need to include the depot node in each cluster by adding the constraint

$$x_{0m} = 1 \quad \forall m \in [k]$$

and summing from $i, j = 0$ in the objective function. However, this implies that for each node i in V there is exactly one m for which $x_{im} = 1 = x_{0m}$. Including the depot node would therefore mean adding the constant term $\sum_{i=1}^{n} c_{0i}$ to the objective function. Therefore, we can exclude the depot node from the beginning.

By augmenting the objective function with additional terms, we can incorporate additional information gained by the graph convolutional neural network (GCN) presented in Chapter 2 into the cluster decision process instead of relying solely on the travel times c_{ij}. For this purpose, for each edge (i, j) let p_{ij} be the probability to be included in the correct solution. Note that this associated probability value is directional, i.e., $p_{ij} \neq p_{ji}$. For the clustering procedure, however, the directions do not have to be considered and would even distort the result, as a large probability for one direction does not mean that the probability for the other direction is also large. We therefore take the maximum value of the two directions, because if the associated probability value in one of the directions is reasonable, it is to be assumed that the pair will be routed on the same route, i.e. should be assigned to the same cluster:

$$\overline{p_{ij}} = \max\{p_{ij}, p_{ji}\}.$$

Furthermore, let α_{dist} and α_{prob} denote the tuning parameters assigned to each part of the objective function summing up to 1, then the clustering problem can be formulated as follows:

$$\begin{aligned} \min \quad & \alpha_{\text{dist}} \sum_{m=1}^{k} \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{im} x_{jm} \\ & + \alpha_{\text{prob}} \sum_{m=1}^{k} \sum_{i=1}^{n} \sum_{j=i+1}^{n} (1 - \overline{p_{ij}}) x_{im} x_{jm} \quad \text{s.t.} \\ & \sum_{m=1}^{k} x_{im} = 1 \quad \forall i \in V, \\ & x_{im} \in \{0, 1\} \quad \forall i \in V, m \in [k]. \end{aligned} \tag{4.21}$$

A shortcoming of using these probabilities straightforwardly for clustering is that it may overlook important relationships between nodes. For example, if we have edges (v_1, v_2) and (v_2, v_3) with high probabilities, but the GCN assigns a low probability to the edge (v_1, v_3) because it recognizes the benefit of visiting v_2 between v_1 and v_3, we want to incorporate the insight that v_1 and v_3 should be assigned to the same cluster into the optimization problem. We do this by replacing the probabilities p_{ij} in (4.21) by *path probabilities*, which we define as:

$$p_{ij}^{\text{path}} := \max \left\{ \prod_{(v,w)\in P} p_{vw} : P \text{ is } i,j-\text{path} \right\}. \tag{4.22}$$

In the following examples, we will show how this affects the cluster building.

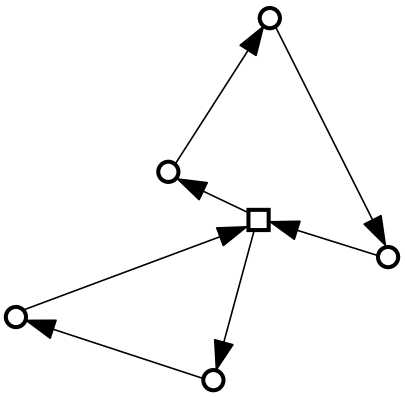


Figure 4.2: Example of a 5-node Instance with the Optimal Solution

Example 4.1. *Figure 4.2 shows an instance with 5 customers, which are shown as circles, and the depot, which is shown as a square. The correct solution consists of two routes, which means that two clusters must be formed. We now solve the clustering problem (4.20)*

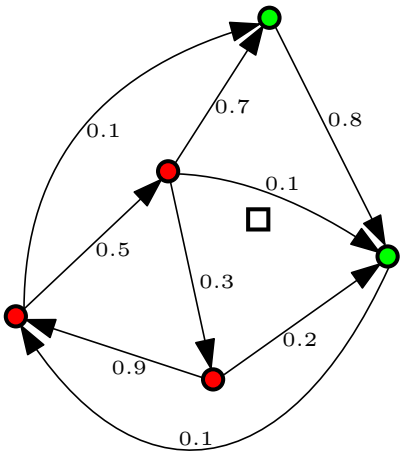


Figure 4.3: Cluster Assignment with Probabilities

with c_{ij} given as the edge weights as in Figure 4.3, where for the sake of simplicity, only

those edges are shown that have been assigned a significant probability by the neural network and the edges to or from the depot are omitted. Solving the clustering problem (4.20) yields the two clusters marked as red and green in Figure 4.3, respectively. We can see that one vertex is not assigned to the correct cluster compared to the optimal solution depicted in Figure 4.2. If we now transform the p_{ij} values into the path probabilities (4.22), some of

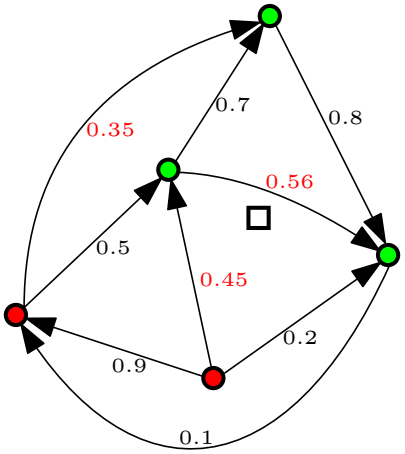


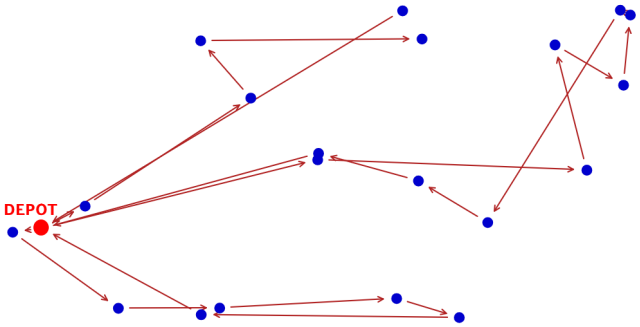
Figure 4.4: Cluster Assignment with Path Probabilities

the depicted edge values do change, which are marked red in Figure 4.4. By solving the clustering problem with the new edge weights p_{ij}^{path}, we do obtain the two correct clusters, again labeled in red and green, respectively.

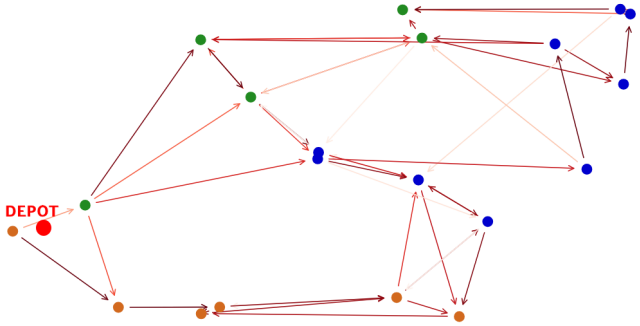
Figure 4.5 shows an example of the edge heatmaps after transforming the associated edge values p_{ij} to the path probabilities p_{ij}^{path} for an instance with 20 vertices. The optimal solution consists of three tours, corresponding to three clusters. These clusters are represented by nodes of different colors in Figure 4.5(b) and (c). In Figure 4.5(b), edges (i, j) with $p_{ij} \geq 0.25$ are shown, with darker red indicating larger values. Edges to and from the depot node, which is displayed in red, are omitted for clarity. In Figure 4.5(c), the transformed p_{ij}^{path} values are displayed, highlighting an increase in associated probability values, particularly for edges within the true clusters.

Figures 4.6 and 4.7 show an example solution of the Clustering Problem (4.21) for an instance with 50 nodes. Figure 4.6 shows the optimal solution on the left and the edge probability heatmap on the right, where only edges with an associated value greater than 0.25 are displayed. Figure 4.7 shows the clusters built by solving (4.21), where each cluster is depicted as a different node color, for the different values of k. As can be seen, no single solution constructs all clusters correctly, although the built clusters already exhibit high accuracy. However, by combining the three different cluster solutions, we achieve that each correct cluster is included as a subset. This also highlights that spatial arguments alone do not accurately represent the solution structures of CVRPTW instances.

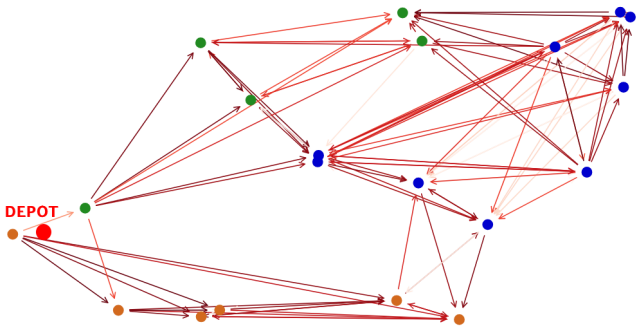
Instead of using the travel times as a criterion for the inclusion of two vertices in the same cluster, we can use another measure, the so-called *polar angular difference*. Based



(a) Optimal Solution

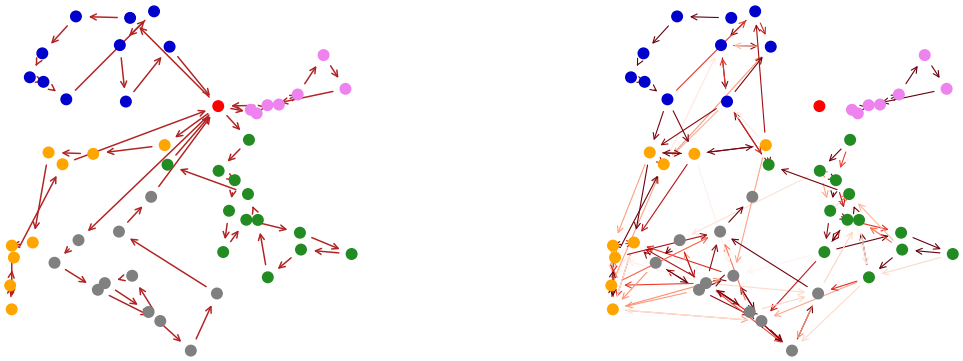


(b) Edge Probability Heatmap



(c) Path Probability Heatmap

Figure 4.5: Example Predictions and Transformation to Path Probabilities for 20 Node Instance



(a) Optimal Solution (b) Heatmap

Figure 4.6: Optimal Solution and Path Probability Heatmap for 50 Node Instance

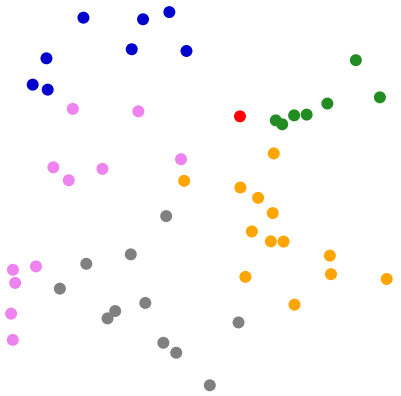
on the idea of the sweep heuristics ([GM74], [Sol87]), for any two vertices i and j, we can calculate the angle β_{ij} between the arrays going from the depot to i and j, respectively, and then minimize the sum of the polar angle differences in each cluster the same way as with the travel times c_{ij} in (4.21). Figure 4.8 illustrates the calculated polar angles to be used for minimization, where the square node represents the depot and a selection of the angles to be used is displayed.

4.2.3 Candidate Route Generation

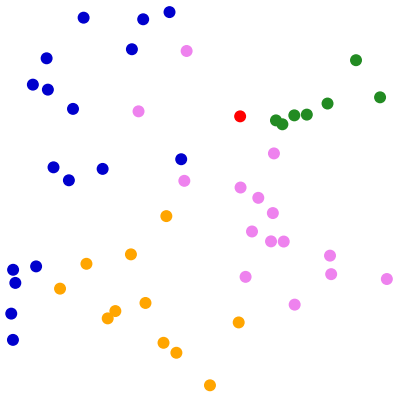
For each cluster $C_i, i \in [k]$, we solve the CVRPTW by employing the deep learning-assisted beam search developed in Chapter 2. Note that a CVRPTW solution for C_i can consist of several routes. Since the purpose of this phase is to generate candidate routes to be included in the SPP in Phase 3, which then selects routes from the set of generated candidate routes in Phase 2 to form a feasible solution to the complete instance, it is beneficial to not only create one set of candidate routes per cluster. Therefore, we apply a heuristic that is well suited to generate multiple sets of feasible routes with respect to the side constraints simultaneously and has a high accuracy on small instances as represented by the clusters. Each unique route in the set of candidate routes is then included as a variable in the SPP.

4.2.4 Set Partition Problem

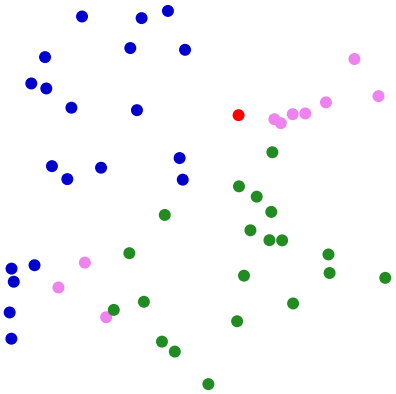
As described in [DMR14], the CVRPTW can be formulated as a Set Partition Problem (SPP), which we will define in the following. The SPP is the basis of the currently best exact method for the CVRPTW, the branch-cut-and-price algorithm [DMR14]. It uses the SPP as the master problem, as the solution of the linear relaxations of the SPP provides



(a) Clustering Problem Solution for 5 Clusters



(b) Clustering Problem Solution for 4 Clusters



(c) Clustering Problem Solution for 3 Clusters

Figure 4.7: Solution for Clustering Problem (4.20) for Different Number of Clusters

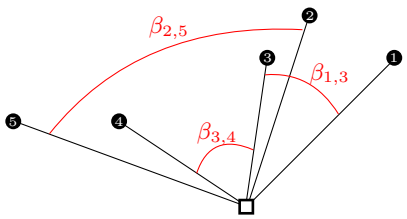


Figure 4.8: Example Angles for Inclusion in Polar Angular Difference Minimization

better lower bounds than, for example, the linear relaxation of the MILP (4.2).

For this, let R be the set of all feasible routes on subsets of vertices. For $r \in R$ denote with c_r the travel time of route r, let δ_{vr} be a binary coefficient that is equal to one if and only if vertex $v \in V$ is in route r. Define the decision variable y_r to be:

$$y_r = \begin{cases} 1 & \text{if route } r \text{ is used in the optimal solution,} \\ 0 & \text{else.} \end{cases} \tag{4.23}$$

Then the SPP can be formulated as an integer program

$$\begin{aligned} \min \quad & \sum_{r \in R} c_r y_r \quad \text{s.t.} \\ & \sum_{r \in R} \delta_{vr} y_r = 1, \quad v \in V \\ & y_r \in \{0,1\} \quad \forall r \in R, \end{aligned} \tag{4.24}$$

where we minimize the travel times of the chosen subsets of routes under the constraint that each vertex appears in exactly one route. For a given solution y of this problem we can link together all routes indicated by y to obtain a feasible solution to the instance with respect to the side constraints such that every node is visited and every route starts and ends at the depot, therefore representing a complete solution to the CVRPTW. This translates to the following QUBO formulation:

$$\min \quad \sum_{r \in R} c_r y_r + P \left(\sum_{v \in V} \left(\sum_{r \in R} \delta_{vr} y_r - 1 \right)^2 \right), \tag{4.25}$$

with P being a penalty factor. This QUBO problem is then solved by the DA. The optimal solution of (4.25) is a set of routes R^* such that for each vertex $v \in V$ (except for the depot) there is exactly one route $r^* \in R^*$ with $v \in r^*$ such that the sum of the travel times of the routes in R^* is minimal. With R being the set of all feasible routes, this corresponds to the optimal solution of the CVRPTW. But in general, finding the optimal solution for this problem is only possible for small instances if all feasible routes are included in R, as the number of routes grows exponentially fast. For our approach, we do not choose R as the set of all possible routes but only include the subset of routes generated as described in Section 4.2.3 in the consideration, such that the number of routes included in the SPP is easily controllable via the beam width b.

4.2.5 Computational Experiments

This section aims to evaluate the overall performance of our approach, comparing the results to other developed approaches in this thesis and state-of-the-art solution methods for the CVRPTW. Additionally, we conduct experiments on hyperparameter configuration.

weighting	$c+p$	$\beta+p$	$c+p^{\text{path}}$	$\beta+p^{\text{path}}$
(1.0, 0.0)	0.52	0.43	0.52	0.43
(0.8, 0.2)	0.54	0.49	0.71	0.64
(0.5, 0.5)	0.73	0.73	0.79	0.74
(0.2, 0.8)	0.75	0.67	**0.91**	0.85
(0.0, 1.0)	0.72	0.72	0.83	0.80

Table 4.4: Cluster Assignment Analysis For Different Parameters and Weights Solving (4.20) in Phase 1

4.2.6 Implementation and Hyperparameters

All models are implemented in Python 3.10 and run under Windows 10. The neural network architecture as well as the beam search are implemented as described in Section 2.3.1. The path probabilites p_{ij}^{path} are calculated by applying the Floyd-Warshall algorithm [Flo62] to the graph (G, ω) with the weight function $\omega : E \mapsto \mathbb{R}, \omega((i,j)) = -\log(p_{ij})$ to find shortest paths. With the weight function ω defined like this, the shortest path in (G, ω) corresponds to maximizing (4.22).

To find the best configuration of weights in the objective function (4.21), each combination of weights in $\{0.0, 0.1, \ldots, 0.9, 1.0\}$ such that the sum is one is tested on a dataset containing 50 instances for combinations of the travel times c, the edge probabilities p given by the neural network, the path probabilities p^{path} as well as the polar angles β. For each combination, we report the accuracy of the cluster solution by determining the percentage number of vertices that were assigned to the correct cluster in the solution of (4.20), where a value of 0.5 translates to 50% of the nodes were assigned to the correct clusters. A representative selection of the results is shown in Table 4.4. The best weighing of the objective function given by the manual experiments presented in Table 4.4 is given as $\alpha_{\text{dist}} = 0.2, \alpha_{\text{prob}} = 0.8$. The accuracy of determining clusters based solely on travel times or polar angle differences is lower because the time window constraints in CVRPTW significantly impact route feasibility, making spatial arguments less relevant. However, we observe that accuracy increases when using path probabilities rather than plain probabilities provided by the neural network combined with spatial arguments.

To set the hyperparameters of the beam search, namely the beam width, as well as the time limit for calculations using the DA, we manually conduct different experiments on small sets of instances for all problem sizes to test different impacts. First, given the correct clusters, we test the impact of beam width and time limit for the SPP optimization problem. The results are reported in Table 4.5. Second, given the clusters found by solving (4.20) with the DA and the routes built by the beam search for each cluster, we include the routes of the correct solution to find the average calculation time it takes the DA to find this solution when solving the SPP (4.25). The results are presented in Table 4.6.

In Table 4.5 we include the average number of routes included in the SPPP, representing the total number of unique routes across all clusters in Phase 2, which, in turn, directly

problem size	b	cand. routes	route factor	t_{SPP}	gap
$n = 20$	10	37.8	11.75	1	0.0179
				2	0.0179
				5	0.0179
	20	63.6	20.54	1	0.0103
				2	0.0074
				5	0.0074
	50	123.1	38.67	1	0.0091
				2	0
				5	0
$n = 50$	50	198.4	45.08	5	0.0326
				10	0.0297
				20	0.0297
	100	348.4	78.84	5	0.0298
				10	0.0105
				20	0.0105
	150	541.2	136.26	5	0.0128
				10	0
				20	0
$n = 100$	100	577.4	87.86	10	0.0649
				20	0.0608
				40	0.0605
				50	0.0605
	200	1345.46	158.66	10	0.0627
				20	0.0488
				40	0.0314
				50	0.0314
	300	2078.91	232.12	10	0.0483
				20	0.0224
				40	0
				50	0

Table 4.5: Accuracy Analysis for Phase 2 and 3

problem size	b	λ	$\lvert R\rvert$	average solving time for SPP (4.25)
20	50	2	126	2.46
50	150	3	518	5.05
50	300	3	1015	7.06
100	300	3	2162	22.63
100	400	3	3107	30.45

Table 4.6: Time Limit Analysis for Phase 3

correlates to the number of decision variables included in the SPP (4.25). We further list the average route factor, which we define to be the average of the number of unique routes built per cluster given by the beam search. Column b states the applied beam width, whereas t_{SPP} is the time limit in seconds given to the DA for solving the SPP. The gap column states the gap between the objective value of the found solution to the objective value of the best known solution. The reported runtimes are from a local machine and include the runtime of the beam search and the DA. For $n = 20$, we choose a cluster size range $\lambda = 2$, for the other problem sizes $\lambda = 3$.

With between 3 and 4 clusters built on average for instances with 20 nodes, the route factor 38.67 corresponds to 123 decision variables included. The beam width b may be larger than the route factor because the beam search may generate different solutions that include the same routes or different orderings of the same routes. These variations are treated as distinct solutions by the beam search, leading to a higher beam width. As this is done individually for each cluster, the number of candidate routes to be included in the SPP is subsequently larger than the beam width. The results show that even one to two seconds is enough to solve the SPP (4.25) to optimality for this problem size. We further observe that in this range of beam sizes, the difference in runtime is minimal. We, therefore, choose a beam width of 50 and a time limit of 5 seconds for the DA to account for the increase of variables when solving the clustering problem (4.20) for multiple values of k to be sufficient for further evaluation of the approach. With the results reported for instances with 50 and 100 nodes in Table 4.5, we choose a beam width of 150 and 300, respectively.

Table 4.6 shows the average calculation time it takes the DA to find the best solution for the SPP QUBO problem (4.25), where b denotes the beam width, λ is the cluster range, and $|R|$ is the number of decision variables included in (4.25). Based on the results, we choose a value of 5, 15, and 50 seconds for t_{SPP} for $n = 20, 50, 100$, respectively, to allow for some variance.

4.2.7 Computational Results

In Table 4.7, we compare our approach to the commercial exact MILP solver Gurobi version 10.0.0 [GO22] as well as to the best known exact solution method for vehicle routing problems, the branch-cut-and-price (BCP) algorithm ([SUP21], [EQSU23]). We further compare results to the heuristic LKH3 [Hel17] and Google's OR-Tools (GORT) version 9.5.2237 library [PF22], both of which frequently serve as heuristic baselines in related literature. The comparison is done by calculating the percentage gap between the objective value of the found solution to the objective value of the best known solution for each instance. The computing time for our 3-phase heuristic is measured as the time for the execution of all three steps without the communication and idle time with the DA. The values are averages over 100 instances.

For GORT, one can choose different configurations regarding the underlying local search heuristic. We conducted experiments for different settings of GORT, which have shown the guided local search (GLS) to be best suited for our needs. As a time limit is needed

for the GLS, we have chosen time limits of 10, 100, and 300 seconds for different problem sizes. Gurobi is applied to the 2-index MILP formulation (4.2)-(4.12) of the CVRPTW and is given a time limit of 10, 100, and 300 seconds per instance for the different problem sizes, respectively. The time limits are chosen to align with the runtime of our method. However, comparing these computation times is challenging, as they are heavily influenced by factors such as implementation efficiency (e.g., Python vs. C++) and the hardware used. In this case, the utilized hardware includes not only GPUs and CPUs but also specialized components, namely the Digital Annealer. We evaluate the baseline models on machines with two Intel Xeon E5-2680v3 @ 2.50GHz CPUs with 12 cores and 128GB RAM available at the High-Performance Computing Cluster of Hamburg University of Technology, while the execution of our 3-phase heuristic is done on a local machine with an Intel Core i7-1165G7 CPU @ 2.80GHz and 16GB RAM running on Windows 11. The reason for this is that communication with Fujitsu's DA is limited to selected local machines, which requires the entire heuristic, including the beam search, to be run on this machine. This in turn affects the runtime of our approach and complicates the comparison with the other approaches in terms of execution time. Therefore, we mark our time in italics in Table 4.7 and also report results for LKH3, GORT-GLS, and BCP on our local machine used for the 3-phase heuristic to provide some more comparability regarding the computational time. We furthermore list the results of the beam search (cf. Chapter 2) used to generate the candidate routes within each cluster applied to the whole instance with a beam width of 1000.

The evaluation of our approach as well as the baseline models is done on datasets containing 100 randomly generated instances following the same data distribution as described in Chapter 2.

Model	$n = 20$			$n = 50$			$n = 100$		
	cost	gap (%)	time	cost	gap (%)	time	cost	gap (%)	time
3-Phase Heuristic	5.95	0.00	*10.18*	10.22	0.30	*80.08*	15.10	1.45	*153.53*
Gurobi	5.95	0.00	0.60	10.27	1.63	51.61	17.92	18.47	300.00
BCP	5.95	0.00	1.07	10.10	0.00	10.68	14.89	0.00	119.87
GORT-GLS	5.95	0.00	10.00	10.13	0.33	100.00	15.12	1.56	300.00
LKH3	5.95	0.00	6.20	10.13	0.30	11.74	15.08	1.28	19.69
GORT-GLS local	5.95	0.00	20.00	10.16	0.58	100.00	15.68	3.99	300.00
LKH3 local	5.95	0.00	10.41	10.13	0.30	23.75	15.08	1.28	40.96
BCP local	5.95	0.00	3.31	10.10	0.00	19.30	14.89	0.00	227.17
BS 1k	6.28	5.47	7.24	11.73	16.14	33.51	18.63	25.12	98.79

Table 4.7: Mean Cost, Gap and Computation Time per Instance for Different Problem Sizes

Our 3-phase heuristic demonstrates optimal performance for small instance sizes. For instances with 50 nodes, the heuristic yields results comparable to GORT-GLS and LKH executed on a high-performance computer, while outperforming GORT-GLS on the same

local machine in terms of solution quality given the same computation time. For medium-sized instances, Gurobi already cannot solve most of the instances to optimality within the given time limit. The BCP method outperforms all heuristics for medium instances. However, it shows a larger relative increase in running time for large instances, whereas LKH and our heuristic exhibit much slower scaling in computation time.

For $n = 100$ nodes, our heuristic surpasses GORT-GLS in both running time and quality on both local and high-performance setups, albeit producing slightly inferior results compared to the LKH heuristic. Gurobi is generally unable to solve large instances within the time limit and has a significantly worse optimality gap compared to the heuristics. While BCP continues to produce the best results, our heuristic begins to outperform BCP in terms of running time on the local machine for large instances. This trend is likely to continue as the problem size increases, which shows the potential of our approach to offer better scalability.

4.2.8 Discussion

With our proposed 3-phase heuristic, we provide a proof of concept of how to use specialized quantum(-inspired) computing hardware such as Fujitsu's DA in combination with deep learning to solve combinatorial optimization problems with constraints that are difficult to solve directly with the current state of quantum(-inspired) computing hardware. Computational results show that this approach is promising, as it utilizes the strengths of quantum(-inspired) computing to provide better scalability than other state-of-the-art methods while yielding near-optimal solutions for larger instances.

Future research could focus on improving our approach in different ways. First, optimizing the computation time could involve implementing the heuristic more efficiently in C++ instead of using Python. However, optimizing the calculation time on the DA itself remains challenging, as no termination criterion can be set for the DA. Hence, the DA continues to optimize its found solution until the given time limit is reached. To account for some variance in the calculation time within the same problem size class, the calculation time on the DA typically includes some idle time.

Secondly, the structure of our approach allows each phase to be approached using different methods. For instance, one could compare different approaches for clustering the set of vertices in phase 1. Also, using different methods for finding solutions within each cluster and solving the set partitioning problem might improve the quality of the solutions or the computation time.

Our work demonstrates the potential of leveraging quantum-inspired computing in conjunction with deep learning for complex combinatorial optimization tasks. As quantum and quantum-inspired technologies continue to evolve, the integration of diverse computational techniques and hardware has the potential to provide even more efficient and versatile solutions for complex combinatorial optimization problems.

4.3 Ideas for Exact Approaches

In the previous sections, we have explored how quantum(-inspired) computing can be used to accelerate heuristic solution methods for the CVRPTW, and have conducted experiments to test the extent to which currently available quantum-inspired computing hardware is already capable of doing so. At the same time, it is also of interest to explore theoretical approaches for the use of quantum(-inspired) computation in exact methods, even if the current state of hardware is not yet advanced enough for practical application. In this section, we will investigate how quantum(-inspired) computation can be used in a branch-and-bound framework to determine exact solutions for the CVRPTW.

4.3.1 Branch and Bound

The Branch and Bound method, first proposed by [LD60], is an approach for solving optimization problems by systemically enumerating the candidate solutions within the search space and is the most commonly used tool for solving NP-hard optimization problems [Cla99]. For the CVRPTW, it is also the base for the best known exact approach, the Branch-Cut-and-Price method (cf. Section 1.4).

The set of all candidate solutions builds the root node of a search tree. During the process, the algorithm explores branches of the tree that correspond to subsets of the complete set of solutions. Branching is done by fixing decisions, which exclude subsets of the candidate solution set. Before a branch is explored, it is checked whether this branch can produce a better solution than the best feasible solution found so far, the so-called *upper bound*. For this purpose, a *lower bound* is estimated for this branch that restricts the value of the solution that can be found in this branch. If this lower bound is worse than the global upper bound, this branch is not examined further, as no better solution can be found here. Therefore, an efficient execution of the Branch and Bound algorithm depends on finding good lower and upper bounds in the process.

Our approach for solving the CVRPTW with a Branch and Bound algorithm using quantum-inspired computing builds a solution sequentially by adding one node to the solution in each layer of the search tree. This structure enables us to apply bounding strategies that make use of reduced bounding problems with respect to the partial solution. The Branch and Bound algorithm can be executed as follows.

The root node of our Branch and Bound search tree is initialized with the partial solution $P_0 = [0]$. At each step of the algorithm, one search tree node is selected and evaluated to obtain a lower bound. Afterward, if the lower bound does not exceed the upper bound, we apply a branching strategy and add child nodes to our selected search tree node, which represents adding one vertex to the partial solution represented by its parent search tree node. These new nodes are initialized with the lower bound from its parent. Therefore, we can use this value to select the next node, for which a lower bound is calculated. Otherwise, if the lower bound exceeds the global upper bound, we prune this branch.

For the upper bound, we periodically apply the DA to solve the QUBO formulation

presented in Section 4.1.1. For this, we select the search tree node holding the partial solution with the best lower bound as a starting point for evaluation to improve the upper bound faster and subsequently achieve a faster termination by pruning a greater amount of branches. The partial solution is represented in the QUBO formulation by fixing the corresponding decision variables x_{ij} to 1 if (i, j) is contained in the partial solution and to 0 for all (i, j') with $j' \neq j$, respectively. Likewise, the integer variables s and y from the QUBO formulation are fixed accordingly. In this way, the size of the QUBO formulation decreases the longer the partial solution becomes. At the beginning, the upper bound is set to ∞ or calculated using a heuristic.

Algorithm 5 provides a high-level overview of the above-described Branch and Bound algorithm for the CVRPTW with a classical branching strategy applied, but we will elaborate on different branching strategies in Section 4.3.5.

Next, we describe different strategies to obtain upper and lower bounds, choose the next node to branch on, examine various branching strategies and elaborate on preprocessing and masking strategies to accelerate the optimization process.

4.3.2 Upper Bounds

Obtaining upper bounds during the search process accelerates the search by allowing us to prune the search tree when the lower bound at a node surpasses the best upper bound. At the beginning, an upper bound is calculated using a heuristic. In the Branch and Bound algorithm, a new upper bound is naturally found when reaching a leaf of the search tree, indicating a complete solution. The solution value is then compared to the best upper bound, and if it is lower, is saved as the new best solution and subsequently used for pruning decisions. In this case, all previously calculated lower bounds are compared to the new upper bound and pruned accordingly. As reaching a leaf in the search tree typically occurs only later in the search process, introducing strategies to obtain strong upper bounds early in the search process can expedite the algorithm and lead to faster termination.

Quadratic Unconstrained Binary Optimization

We can employ the DA (cf. Section 4.1) to solve the QUBO formulation (4.16) with its corresponding matrix Q for the CVRPTW via quantum-inspired computing. This offers two ways to integrate the DA into our Branch and Bound algorithm for finding upper bounds.

On the one hand, the DA can be applied to find upper bounds for subtrees of our search tree. For this, we can fix the values of some of the variables. Each node in the search tree holds information imposed by the branching decisions from the root node to that specific node. As we build our solution sequentially, each edge in the search tree corresponds to appending a vertex to the partial solution. Therefore, each search tree node holds a partial solution $P = [v_1, \ldots, v_\ell]$ as well as a set of decision variables with fixed values. For all edges $e = (v_{i-1}, v_i)$, with $i \in \{2, \ldots, \ell\}$, that are contained in the partial solution, the

Algorithm 5: Branch and Bound for the CVRPTW

```
Input: Instance G, period t
Output: best solution s and its cost
Initialize upper_bound ← ∞
Initialize best_route ← []
Initialize initial_route ← [0]
Initialize initial_load ← 0
Initialize initial_cost ← 0
Initialize stack ← [(initial_route, initial_load, initial_cost)]
while stack is not empty do
    (current_route, current_load, current_cost) ← stack.pop()
    if all customers are visited and current_cost < upper_bound then
        upper_bound ← current_cost
        best_route ← current_route
    end
    else
        for each unvisited customer c ∉ current_route do
            if adding c to current_route is feasible w.r.t. constraints then
                new_route ← current_route + c
                new_load ← current_load + c.demand
                new_cost ← current_cost + cost(current_route[-1], c)
                if lower_bound(new_route) < upper_bound then
                    stack.append((new_route, new_load, new_cost))
                end
            end
        end
    end
    if current_time == kt for some k ∈ ℤ then
        choose route r from stack with minimal cost
        new_upper_bound, new_route ← DigitalAnnealer(G, r)
        if new_upper_bound < upper_bound then
            upper_bound ← new_upper_bound
            best_route ← new_route
        end
    end
end
return best_route, upper_bound
```

values of x_e are set to 1. Likewise, variables $x_{e'}$ are set to 0 for those edges e' that are already excluded because of branching decisions or because $e' = (u', v)$ and there exists $e = (u, v)$ with $x_e = 1$, i.e. customer v has already been visited and therefore all other edges that enter v are excluded. Specifically, for a partial solution $P = [v_1, \dots, v_\ell]$ and $k \in \{1, \dots, \ell - 1\}$, if v_k is not the depot node, we set the following decision variables:

$$x_{v_k j} = \begin{cases} 1 & \text{if } j = v_{k+1} \\ 0 & \text{else.} \end{cases} \tag{4.26}$$

Additionally, a partial solution inherits context information regarding the current position, time and load of the partially built route, which is stored at the search tree node. This context is easily computed within the search tree by utilizing the context from its parent node, it is only necessary to examine the last added edge to obtain the current context. The integer variables s and y from the QUBO formulation are fixed accordingly. In this way, the size of the QUBO formulation decreases the longer the partial solution becomes.

By including the context of the partial solution, we can fix decision variables to zero whenever the edge (v_ℓ, j) violates the side constraints for $j \notin P$. Optimizing the resulting reduced QUBO results in an upper bound for all subproblems sharing these fixed variables, and therefore can serve as an upper bound for this subtree.

Alternatively, due to its underlying optimization process, the DA can incorporate a *guidance configuration* as a starting point for its search. This guidance configuration allows us to set values of decision variables as guidance for the DA to start its optimization process. Note that, in contrast to the first approach to apply the DA by fixing the variables corresponding to a partial solution, setting the values of these decision variables in the guidance configuration does not imply that their values are fixed and that these variables are not directly involved in the optimization process. Instead, these assigned values serve as indicators instructing the DA with a starting point for its search process.

This can be utilized in two different ways. First, in each call of the DA, we can use the best feasible integer solution found up to that point as the guidance configuration. This solution can be obtained from a leaf in the search tree or from the last iteration of the Annealer's search process. Second, similar to the lower bound strategies, we can select the search tree node with the best lower bound and utilize the partial solution as the initial guidance configuration. For each node added to the solution, we define the initial starting point of the DA's search process by assigning the corresponding decision variable a value of one in our guidance configuration. Starting its optimization process with this information, the DA generates a valid solution that serves as an upper bound for the entire search process.

Since the optimization process of the DA requires communication time to the hardware in addition to computing time, it is necessary to limit the number of applications of the DA during the execution of the Branch and Bound algorithm. This is achieved by specifying the time period t as a hyperparameter, which specifies the time interval in which the DA is applied.

4.3.3 Lower Bounds

Since a lower bound is calculated for each newly added search tree node, it is essential that the determination of the lower bound is not computationally demanding to maintain a reasonable computational approach. Furthermore, while finding a feasible lower bound is straightforward (e.g., zero is always a valid lower bound), such trivial bounds do not help in accelerating the algorithm as they do not help in pruning branches where the lower bound exceeds the upper bound. Therefore, the goal is to find lower bounds that are as large as possible, ideally close to the feasible solution, to effectively prune non-promising branches and accelerate the search process.

In the following, we describe the approach of utilizing the QUBO formulation for determining lower bounds using semidefinite programming as well as other methods for finding good lower bounds in a reasonable amount of time that fit into our Branch and Bound framework of sequentially adding vertices to partial solutions.

Semidefinite Programming

We utilize the QUBO matrix Q of the CVRPTW that is used to solve the problem via quantum-inspired computing (cf. Section 4.1), to also determine lower bounds. By changing the domain of the decision variables x from $\{0,1\}$ to $[0,1]$, we transform the problem into a convex optimization problem that is subsequently solved using semidefinite programming (SDP) techniques. This offers the advantage of formulating the problem only once as a QUBO matrix, which, when combined with the quantum-inspired computing method for obtaining upper bounds, eliminates the need for translating formulations and results between the lower bound and upper bound strategies.

For this, let $Q \in \mathbb{R}^{\tilde{n}\times\tilde{n}}$ be the upper triangular matrix of the QUBO formulation (4.16) of a CVRPTW instance and let $z = \{0,1\}^{\tilde{n}}$ be the vector holding the decision variables. Note that $\tilde{n}$ is greater than $|V|^2$, as we have to transform the decision variables x_{ij}, y_i, s_i from the formulation (4.2) and slack variables to single-indexed binary variables for our QUBO formulation. The objective is to find the vector $z \in \{0,1\}^{\tilde{n}}$ that minimizes the energy function $E(z)$:

$$\min_{z\in\{0,1\}^{\tilde{n}}} E(z) = \min_{z\in\{0,1\}^{\tilde{n}}} z^T Q z.$$

Denote by $\mathbb{S}^n$ the space of all $n \times n$ real symmetric matrices. We define the matrix inner product for this space to be the Frobenius inner product:

Definition 4.2 (Frobenius inner product)**.** Let $A, B \in \mathbb{S}^n$. Then

$$\langle A, B\rangle_{\mathbb{S}^n} := A \circ B = \sum_{i,j\in[n]} a_{ij}b_{ij} = \mathrm{Tr}(AB^T).$$

Definition 4.3 (Semidefinite program)**.** A *Semidefinite Program* (SDP) is of the form

$$\min_{X\in\mathbb{S}^n} \quad C \circ X \quad \text{subject to}$$
$$A_i \circ X = b_i, \quad i = 1, \ldots, m$$

$$X \succeq 0,$$

with $X, C, A_1, \ldots, A_m \in \mathbb{S}^n$. The last condition, $X \succeq 0$, is an order relation that states X is positive semidefinite (PSD). A matrix X is PSD if, for any real-valued nonzero vector v, the value $v^T X v$ is non-negative.

Now, let $zz^T = \mathbf{Z} = (\mathbf{z}_{i,j})_{1 \leq i,j \leq \tilde{n}} \in \mathbb{R}^{\tilde{n} \times \tilde{n}}$.

Lemma 4.4. *The above-defined matrix* $\mathbf{Z}$ *is positive semidefinite and symmetric.*

Proof. The symmetric property can easily seen by multiplicating. To show that $\mathbf{Z}$ is PSD, let $v \in \mathbb{R}^{\tilde{n}}$ be any real-valued nonzero vector. Then

$$v^T \mathbf{Z} v = (v^T z)(z^T v) = (v^T z)^2 \geq 0.$$

□

We are interested in finding lower bounds for the energy function $E(z)$, i.e. we want to find values LB^{SDP}, such that

$$LB^{\text{SDP}} \leq E(z).$$

Recall that the matrix Q from our original QUBO formulation is upper triangular. Since $\mathbf{Z}$ is symmetric, without changing the result we can transform Q to be symmetric by changing the entries of Q to $\frac{q_{ij}+q_{ji}}{2}$ for $i, j \in [\tilde{n}]$. It then follows that

$$z^T Q z = \mathbf{Z} \circ Q = \text{Tr}(\mathbf{Z} Q^T)$$

and, since $\mathbf{Z}$ is symmetric, this is equal to $\text{Tr}(\mathbf{Z}Q)$. Since the energy function can be expressed as $z^T Q z$, a lower bound for the energy function can be obtained by optimizing the relaxed problem using an SDP:

$$\begin{aligned} LB^{\text{SDP}} = \min_{\mathbf{Z} \in \mathbb{S}^{\tilde{n}}} \ & \text{Tr}[\mathbf{Z}Q] \quad \text{subject to} \\ & \mathbf{Z} \succeq 0 \\ & 0 \leq \mathbf{z}_{i,j} \leq 1 \quad \forall i, j \in [\tilde{n}]. \end{aligned} \tag{4.27}$$

Using this approach, we are able to also calculate lower bounds at nodes in the search tree that hold partial solutions, i.e. $z_i \in \{0, 1\}$ for some subset $i \in S \subset [\tilde{n}]$, by pre-determining the corresponding values in $\mathbf{Z}$, since the convex PSD constraint intersected with linear constraints, half spaces or hyperplanes, remains a convex set.

For this, let $F = \{(i, j) \mid i, j \in [\tilde{n}], \mathbf{z}_{i,j} = 0\}$ and $I = \{(i, j) \mid i, j \in [\tilde{n}], \mathbf{z}_{i,j} = 1\}$ denote the sets of indices corresponding to edges that are already excluded and included by a partial solution, respectively. Then the SDP is given as:

$$\begin{aligned} LB^{\text{SDP}} = \min_{\mathbf{Z} \in \mathbb{S}^{\tilde{n}}} \ & \text{Tr}[\mathbf{Z}Q] \quad \text{subject to} \\ & \mathbf{Z} \succeq 0 \end{aligned} \tag{4.28}$$

$$0 \leq \mathbf{z}_{i,j} \leq 1 \quad \forall i,j \in [\tilde{n}]$$
$$\mathbf{z}_{i,j} = 1 \quad \forall (i,j) \in I$$
$$\mathbf{z}_{i,j} = 0 \quad \forall (i,j) \in F.$$

The resulting objective value serves as a lower bound for our energy function because of the relaxation of the domain to the interval $[0,1]$. The non-integer values of the decision variables can be used for branching decisions (cf. Section 4.3.5).

Assignment Problem

By the design of our Branch and Bound algorithm, at each layer of the search tree, we add nodes to partial solutions. This sequential approach allows us to derive lower bounds and make branching decisions by adapting the TSP lower bound strategy from [Chr72]. To apply TSP lower bounds to our problem, the CVRPTW is transformed into a TSP by defining an appropriate auxiliary graph. For this, let $P = [v_1, \ldots, v_\ell]$ be the partial solution at the search tree node for which we want to find a lower bound. We define a TSP instance by creating an auxiliary graph $G_P = (V_P, E_P)$, where the vertex set V_P is initialized as the vertices that are not included in P together with the depot. We fully connect G_P and initialize the edge weight

$$c_{ij}^P = \begin{cases} c_{ij} & \text{if } i \neq j \\ \infty & \text{else} \end{cases}$$

to exclude cycles of length 1, with c being the cost matrix of the original problem instance. Dropping the subtour elimination constraints from the TSP yields an Assignment Problem (AP):

$$\begin{aligned} \text{(AP)} \quad LB^{\text{AP}} := \min \quad & \sum_{i,j=0}^{n} c_{ij}^P x_{ij} \qquad \text{s.t.} && (4.29) \\ & \sum_{i=0}^{n} x_{ij} = 1 && \forall j \in [n] \\ & \sum_{j=0}^{n} x_{ij} = 1 && \forall i \in [n] \\ & \sum_{i=1}^{n} x_{0i} - \sum_{j=1}^{n} x_{j0} = 0 \\ & x_{ij} \in \{0,1\} && \forall i,j \in V. \end{aligned}$$

This is an integer linear program, but it can be efficiently solved through standard linear programming techniques by removing the integrality constraints. The resulting optimal solution will always be an integer solution because the constrained matrix of the linear program is totally unimodular [Reb74]. Moreover, the AP can be solved in $\mathcal{O}(n^3)$ by

means of an efficient implementation of the Hungarian algorithm [Mun57], which can be reduced to $\mathcal{O}(mn + n^2 \log n)$ with m being the number of edges [FT87].

As our auxiliary graph G_P contains the depot, solving the AP results in an assignment with two edges connected to the depot. As solutions to the CVRPTW consist of several different routes, the depot is usually connected to more than two nodes. To account for this and to subsequently improve the lower bound, we add several dummy depots to G_P. Since we do not know the correct number of routes for the optimal solution of our CVRPTW instance in advance, we use an estimation. For this, we adapt the lower bounds on the number of vehicles for CVRPTWs developed in [AGM14] to obtain a good estimate. We then determine how many already built routes exist in P and subtract this to obtain the number of dummy depots $\bar{k}$ we are allowed to add to our auxiliary problem. The dummy depots $\{v_{d_1}, \ldots, v_{d_{\bar{k}}}\}$ with edges to all other nodes are added to G_P with the following edge weights for $v \in \{v_{d_1}, \ldots, v_{d_{\bar{k}}}\}$:

$$c^P_{vj} = \begin{cases} c_{0j} & \text{if } j \notin \{0, v_{d_1}, \ldots, v_{d_{\bar{k}}}\} \\ \infty & \text{else,} \end{cases}$$

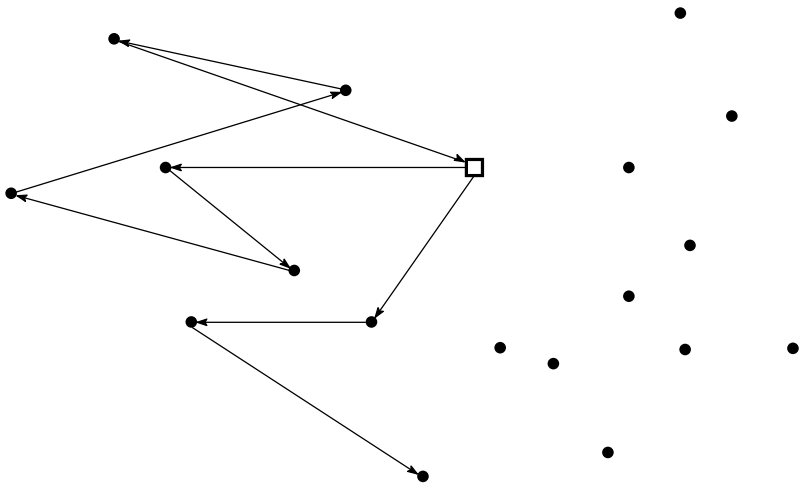
$$c^P_{iv} = \begin{cases} c_{i0} & \text{if } i \notin \{0, v_{d_1}, \ldots, v_{d_{\bar{k}}}\} \\ \infty & \text{else.} \end{cases}$$

By setting these edge weights for the dummy depots, we ensure that no depots are directly connected in the solution of the AP. Furthermore, as we work with partial solutions, assume that the last visited node v_ℓ in P is not the depot. To obtain valid lower bounds, we also have to factor in that only one more edge connected to v_ℓ is allowed as well as one edge already leaves the depot. For that, we add another dummy node v_x to G_P with:

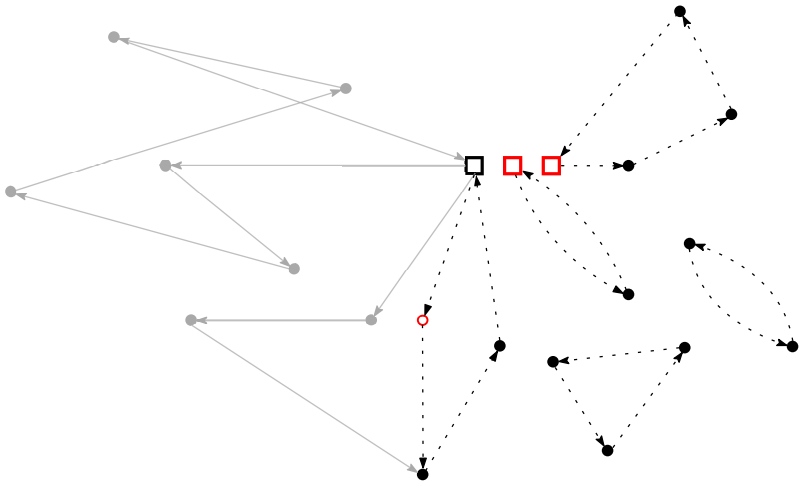
$$c^P_{ij} = \begin{cases} 0 & \text{if } (i,j) = (v_x, v_\ell) \text{ or } (i,j) = (0, v_x) \\ \infty & \text{else.} \end{cases}$$

This ensures that in the solution of the AP, only one edge with a weight greater than zero is leaving v_ℓ and entering the depot, respectively. Solving the AP for $G_P = (V_P, E_P, c^P)$ with $V_P = (V \setminus P) \cup \{v_{d_1}, \ldots, v_{d_{\bar{k}}}, v_x\}$ yields a valid lower bound for the CVRPTW.

Example 4.5. *Figure 4.9(a) shows an instance of the CVRPTW that is already partially solved with the partial solution displayed as the edges included in the plot. The depot node is marked as a square, whereas the customer nodes are shown as circles. The partial solution consists of one completed route visiting five nodes and one partially constructed route. Figure 4.9(b) shows the auxiliary graph G_P, with the partial solution still shown in grey for clarity. The red squares are two dummy depots, because the lower bound on the number of vehicles equals four for this instance, and the dummy node v_x is shown as the red circle. The sum of the edge weights from the edges contained in the solution is added to obtain the lower bound LB^{AP}.*



(a) Instance with partial solution



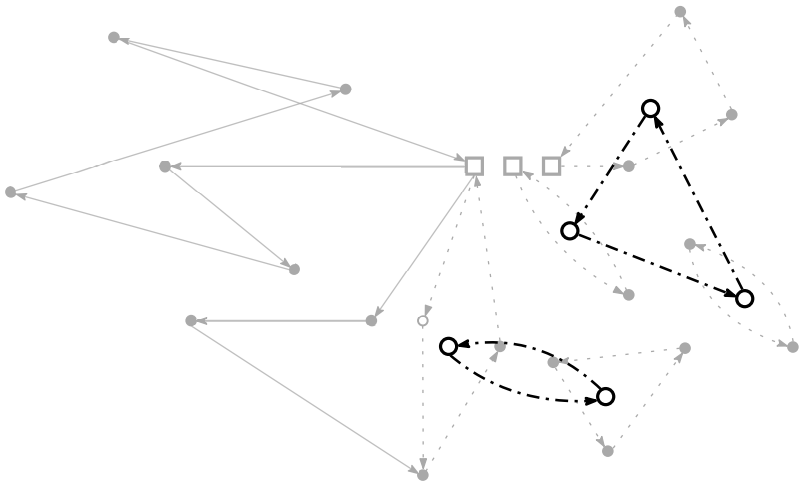
(b) Solution of Assignment Problem on auxiliary graph G_P

Figure 4.9: Solution of Assignment Problem Considering Partial Solution

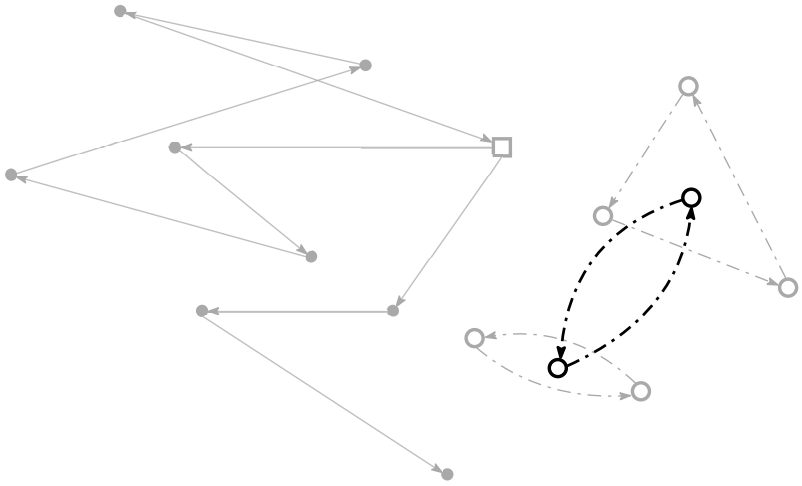
Recursive Assignment Problem

Christofides [Chr72] introduced an improvement to the assignment bound for the TSP. Typically, the solution to the AP contains cycles. To address this, he developed a recursive method for incorporating the necessary costs to connect these cycles. Using the Hungarian method, the AP is solved and the reduced cost matrix, which is calculated during the Hungarian algorithm [Mun57], is stored. Then, the cycles of the solution are contracted, where a single node replaces the nodes in a cycle. For these new nodes, a new cost matrix is derived from the reduced matrix, considering the minimal costs between cycles. Then, the cost matrix is compressed by replacing each element c_{ij} with $\min\{c_{ij}, \min_k\{c_{ik} + c_{kj}\}\}$ to maintain the triangle inequality. Using this updated matrix, the assignment problem is solved again. The optimal value obtained is then added to the current lower bound. This

is repeated until the solution consists of only one cycle, at which point the calculated lower bound LB^{RAP} is returned.



(a) Solution of first recursion



(b) Solution of second recursion

Figure 4.10: Solution of Recursive Assignment Problem

Example 4.6. *Figure 4.10 displays the recursive application of the assignment problem. The solution to the assignment problem in Figure 4.9(b) contains five cycles. These are contracted to single nodes and, together with the reduced and compressed cost matrix, the assignment problem is solved again, which yields the solution as shown in Figure 4.10(a). The reduced edge weights of the five edges, which are contained in the solution, are added to the lower bound* LB^{AP} *obtained from the solution in Figure 4.9(b). As the number of cycles of this solution is still greater than one, the same procedure is applied again to obtain the solution displayed in Figure 4.10(b), and the reduced edge weights are added to the lower bound. Since this solution consists of only one cycle, the recursion terminates*

and the calculated lower bound LB^{RAP} is returned.

Minimal Spanning Trees

In the paper of Held and Karp [HK71], minimal spanning trees are used to obtain lower bounds for the TSP. More specifically, they use minimum-weight 1-trees, which are trees that consist of a vertex attached with two edges to a spanning tree. For this, let T^* be the weight of the optimal TSP solution and W^* be the weight of a minimum-weight 1-tree on the vertex set $\{1\dots,n\}$ with the depot node 0 attached with two distinct minimal weight edges. Because a TSP solution is a special 1-tree, where every vertex has degree two, W^* can serve as a lower bound for the TSP. Moreover, if the minimum-weight 1-tree is a TSP tour, it is a tour of minimum weight. A minimal spanning tree can be obtained efficiently by applying for example the Kruskal algorithm [Kru56] with a complexity of $\mathcal{O}(n^2 \log n)$. As described in Section 4.3.3, in the context of solving the CVRPTW and having a partial solution $P = [v_1, \dots, v_\ell]$, we can transform our instance on the set of unvisited vertices to a TSP instance and therefore obtain valid lower bounds by finding minimum-weight 1-trees. For this, the cost matrix needs to be symmetric, therefore we set $\bar{c}_{ij} = \min(c_{ij}, c_{ji})$.

Held and Karp [HK71] describe an iterative method, called the subgradient ascend method, to tighten the lower bound found via minimum-weight 1-trees by altering the original graph to generate different minimum-1-trees in each iteration. The method is based on the idea of penalizing vertices in the minimum-weight 1-tree that do not have a degree of two such that the degrees of all vertices get iteratively closer to two and therefore closer to a feasible tour. For this, at each iteration a weighted cost matrix $(\gamma_{ij})_{1\leq i,j\leq n}$ is calculated by setting the edge weights to $\gamma_{ij} = \bar{c}_{ij} + \pi_i + \pi_j$, with π being the penalty vector. Let W_k be the weight of the k-th 1-tree and d_{ik} the degree of vertex i in the k-th 1-tree, it then holds that

$$T^* + 2\sum_{i=1}^{n} \pi_i \geq \min_k \left\{ W_k + \sum_{i=1}^{n} \pi_i d_{ik} \right\}, \tag{4.30}$$

since every vertex in the TSP tour has degree two. By defining $W^*(\pi)$ to be the weight of the minimum-weight 1-tree with penalty vector π, $\nu_i = d_i - 2$ and $\omega(\pi) = W^*(\pi) + \pi\nu$, we have that (4.30) is equivalent to

$$T^* \geq \omega(\pi). \tag{4.31}$$

As this is an infinite family of lower bounds, the best of these bounds is given as

$$LB^{\mathrm{MST}} := \max_{\pi}(\omega(\pi)). \tag{4.32}$$

The subgradient ascent method iteratively changes the penalty vector π to obtain better lower bounds from this family by approaching degree two for every vertex in the 1-tree. For this, let ν_i^m be the degree minus two of vertex i in the 1-tree found in iteration m. The penalty for the next iteration is set to $\pi_i^{m+1} = \pi_i^m + t\nu_i^m$, with t being an appropriately chosen step size. This decreases the weight of edges connected to vertices with a degree

of one and increases the weight of edges connected to vertices with a degree greater than two. The penalty vector is initialized as the $n-$dimensional vector containing only zeros.

4.3.4 Preprocessing and Masking Strategies

As shown in [KLMS05], preprocessing can be used to narrow the solution space before the actual optimization starts to accelerate the optimization process. This can be done among other things by determining the value for a subset of the decision variables beforehand.

The lower bounds obtained by the assignment problem (AP) and its variant, the recursive assignment problem (RAP) are valid, but they do not account for side constraints, such as time windows and capacity limitations, which results in a larger search space. To address this and improve the lower bounds obtained through the AP, we can preprocess the problem instance by applying a masking scheme to the cost matrix. The time window constraints exhibit both static and dynamic characteristics, with certain edges consistently invalid and others depending on the specific arrival time at the node within a partial solution. In contrast, capacity constraints are primarily dynamic. The static time window constraints can be used for preprocessing the instance before starting the Branch and Bound algorithm by assigning an edge weight of ∞ to invalid edges. Moreover, at each node in the search tree, we can consider the context of the partial solution to apply an additional masking scheme. Assuming that the last visited node v_ℓ in the partial solution is not the depot, we can calculate the exact visit time for this node and determine the remaining vehicle capacity. With this information, we can set edge weights to ∞ for all edges (v_ℓ, j) that do not conform to the context-specific constraints. This also implies that the edge selected in the AP solution, departing from node v_ℓ, is valid for our CVRPTW and can be used for branching decisions. As solutions are constructed sequentially, we can store the context at each tree node, ensuring that computing the context for their child nodes is computationally efficient. Experiments show that this preprocessing significantly enhances the lower bounds obtained by the AP.

4.3.5 Branching Strategies

Branching at a search tree node divides the search space into smaller spaces, creating two or more instances that represent usually disjoint subsets of the original problem. Classical branching strategies usually consider branching on the decision variables, which consists of selecting the fractional variable that would produce the largest increase in the value of the objective function if it was set to either zero or one, and then branching in both directions (cf. [LN83], [TV14]). Other strategies include branching on resource windows or, depending on the formulation of the CVRPTW, branching on the number of vehicles [KLMS05]. These branching strategies have in common that the selection of the variable to branch on relies on either the expected impact of fixing this variable or the calculated reduced costs in the framework of pricing subproblems. In our sequential Branch and Bound approach for the CVRPTW, the branching decision involves choosing the path to follow when incrementally constructing a partial solution until complete and feasible

solutions are obtained. Therefore, only sequential branching decisions are considered.

In the following, we discuss different strategies to make sequential branching decisions in our Branch and Bound search tree depending on the lower bound strategy applied. For this, we assume to have selected a search tree node to branch on which contains the partial solution $P = [v_1, \ldots, v_\ell]$.

Branching on Flow Variables

When obtaining lower bounds via the SDP (4.28), we can define the set of candidate variables to branch on as:

$$B_{\text{SDP}} = \left\{ x_{ij} : x_{ij}^{\text{SDP}} \in (0,1), (i,j) \in E, i = v_\ell, j \notin P \right\},$$

with x_{ij}^{SDP} denoting the value assigned to the variable in the SDP solution. That is, we select the edge to branch on from the set of edges corresponding to those edges leaving v_ℓ that have fractional values in the lower bound solution. In general, if there is more than one candidate in B_{SDP}, we want to select an edge with a fractional value that is neither close to 0 nor 1 to have a greater effect on the lower bound through branching. A common heuristic when selecting the edge to branch on is to select the variable that maximizes $c_{ij} \left(\min \left\{ x_{ij}, 1 - x_{ij} \right\} \right)$ (cf. [KLMS05]). Therefore, when branching on edges is applied, at each node of the search tree an edge is selected to extend the current partial path and two descendant nodes are generated: the first node is associated with the inclusion of the selected edge in the solution (i.e. $x_e = 1$), while in the second node, the edge is excluded (i.e. $x_e = 0$).

Branching on Customers

In the case that the lower bound is not obtained by the SDP approach (4.28), but by the (recursive) assignment problem (4.29) or the minimal spanning trees method (4.32), we do not have relaxed decision variables with fractional values to include in our branching decision.

Instead, we can utilize the solution of the lower bound obtained by solving the assignment problem. If the masking scheme as detailed in Section 4.3.4 is applied, the edge (v_ℓ, u), which is included in the (recursive) assignment problem solution for the lower bound, does comply with the side constraints. Therefore, selecting this edge as a branching decision is valid with respect to our CVRPTW problem. The candidate set is given as:

$$B_{\text{AP}} = \left\{ x_{ij} : x_{ij}^{\text{AP}} = 1, (i,j) \in E, i = v_\ell, j \notin P \right\},$$

which does contain exactly one variable. Similar to branching on the flow variables, we then create two descendants in the search tree corresponding to the inclusion and exclusion of this variable, respectively.

If the current subproblem does not contain a partially built route, e.g. at the root node or when in the previous step a route has been closed by arriving at the depot, in the context of solving the CVRP sequentially within a branch and bound algorithm, it has

been proven beneficial to select the next edge to start a new route as the unvisited node with the highest demand [Mil95]. For the CVRPTW, it is advantageous to rather select the edge going to the unserved customer with the earliest time window, as capacity is significantly less constraining in the CVRPTW [KLMS05]. We apply this strategy within the binary B_{SDP} and B_{AP} strategies.

Additionally, we apply a deep learning-assisted branching strategy that is based on probabilities p_e associated with edge each $e \in E$. For this, we utilize the graph convolutional neural network that is also applied for the problem reduction of the QUBO instance as detailed in Section 4.1. Given a threshold $\bar{p}$, let

$$B_{\mathrm{GCN}} = \{x_{ij} : e = (i,j) \in E, i = v_\ell, j \notin P, p_e > \bar{p}\},$$

be the set of decision variables corresponding to those edges leaving v_ℓ to unvisited nodes, whose associated probability exceeds the threshold. As a binary branching strategy we can now choose the decision variable $x_{ij} \in B_{\mathrm{GCN}}$ with the highest probability and use this to create two branches the same way as described before.

In most of the literature, only binary branching strategies are considered, although developing non-binary branching strategies is generally possible (cf. [KLMS05]). Let $T \in \mathbb{N}$ be a parameter determining the number of branches we want to create for each branching decision. Then, we can select the $T-1$ elements of B_{GCN} with the highest associated value, and generate T new nodes in the Branch and Bound search tree. Each of the first $T-1$ nodes corresponds to the inclusion of one of the selected edges as the next visited customer for our partial solution, while in the last node, those $T-1$ edges are excluded. This ensures a division of the search space into smaller spaces without excluding any solution, maintaining the possibility of finding the exact solution. However, if we do not include the last branch excluding all edges building their own branch, we build significantly fewer branches in the process, as all children in this branch are discarded directly. However, this comes with the downside of losing the property of being an exact solving method as not the complete solution space is being searched, resulting in a heuristic approach that we denote with B_{GCNH}.

4.3.6 Implementation and Experiments

Although we have learned in Section 4.1 that with the current state of quantum(-inspired) computing hardware it is generally not feasible to apply it as the bounding function for finding upper bounds, as described in Section 4.3.2, due to computational time and time contingency limits, we proceeded to implement the Branch and Bound algorithm as detailed in this section. Additionally, we conducted experiments to test the influence of different lower bound strategies.

Implementation

The Branch and Bound algorithm is implemented in parallel with Python 3.10 using the PyBnB package to allow for parallel evaluation of numerous search tree nodes for lower

lower bound method	branching	# tree nodes	# pruned branches	optimality gap
SDP	B_{SDP}	*95152*	*3287*	*0.1819*
MST	B_{GCN}	4892016	1903347	0
AP	B_{AP}	6903829	1423855	0
RAP	B_{AP}	12039203	920481	0
RAP	B_{GCN}	4120728	2794791	0
RAP	B_{GCNH}	278601	99808	0.0287

Table 4.8: Branch and Bound Comparison of Different Branching and Bounding Strategies

bounds while simultaneously applying the DA [MTM+20] to calculate upper bounds. For the search tree, we use Python's heap queue algorithm to maintain a priority queue, determining which search tree node to explore next. The priority queue is structured as a binary tree where each parent node has a value (i.e., a lower bound) less than or equal to that of any of its children. This configuration allows for efficient sorting of the queue while maintaining the correct ordering after each iteration when a better upper bound is found. This enables us to delete all nodes from the heap queue that now have a lower bound exceeding the new best upper bound. The calculation of a new upper bound starts after 100000 lower bounds have been calculated, which is a manually chosen hyperparameter to manage the available time contingent on the DA. For solving the SDPs, we use CVXPY [DB16] and found the SDPA solver (cf. [YFK03], [YFF+12], [Nak10], [KKMY11]), an interior-point method, to be best suited for our application. We choose 100 iterations for the minimal spanning trees lower bound method.

For the branching strategy B_{GCN}, we choose $T = 4$, which corresponds to introducing three branches for the three most promising edges to be included, respectively, while the fourth branch excludes all three edges. Additionally, we conduct experiments where we discard this fourth branch, resulting in significantly fewer branches and a heuristic approach.

Computational Experiments

We test the different configurations of our Branch and Bound algorithm on test instances with 20 nodes as detailed in Section 2.3.2 with a time limit of five hours to allow for the algorithm to terminate for each configuration to compare the effectiveness of different bounding and branching strategies albeit longer computation times. In Table 4.8 we report the number of simulated search tree nodes and the number of branches pruned due to their lower bound exceeding the current upper bound indicating the effectiveness of the different lower bound strategies, as well as the solution quality in terms of the gap to the optimal solution.

The results indicate that the recursive assignment problem is the most effective method for determining lower bounds in this context. This effectiveness stems from the number of branches we were able to prune, allowing us to exclude more search tree nodes before simulation and thus reducing the overall number of nodes that needed to be simulated.

On the other hand, the available solvers for solving semidefinite programming problems struggle with efficiently solving our specific SDP (4.28) for finding lower bounds. The results for this configuration are italicized because the algorithm did not terminate within the time limit for this configuration. As evidenced by the small number of simulated nodes, the relatively long computation times required to solve the SDP (4.28) for lower bounds resulted in only a few pruning decisions, which typically occur more frequently deeper in the search tree. The optimality gap of approximately 18% is due to the heuristic used initially to obtain an upper bound. In practice, this inefficiency undermines the theoretical advantages of using the same QUBO formulation as for the upper bound. Given the large number of simulated search tree nodes for which a lower bound needs to be calculated, faster computation times for the lower bound are more beneficial.

Table 4.8 further demonstrates that employing the B_{GCN} branching strategy significantly reduces the number of simulated tree nodes. This is because generating multiple branches for the most promising edges within the same branching step, rather than just two branches for including or excluding a specific edge, proves to be more advantageous. Also, applying the heuristic branching strategy B_{GCNH} significantly reduces the number of search tree nodes investigated. This reduction results in improved computation times while still providing close to optimal solutions for smaller instances.

For this approach to be more efficient and result in an exact method applicable to larger instances, the hardware for solving QUBOs needs to handle instances with a larger number of variables more efficiently. Additionally, more efficient SDP solvers are necessary.

In summary, while our approach demonstrates potential in reducing computational complexity through effective pruning strategies and heuristic methods, the current limitations of available hardware and solvers highlight the need for further advancements. Improvements in both QUBO-solving hardware and SDP solvers will be crucial for making this method a viable option for application.

Chapter 5

Conclusion

This thesis explored the integration of deep learning methods into heuristics to address routing problems, focusing on the development and evaluation of innovative methodologies combining artificial intelligence and heuristic search techniques to solve the capacitated vehicle routing problem with time windows. The contributions of this work demonstrate the potential for AI-enhanced approaches to provide competitive solutions for complex combinatorial optimization problems, even in the presence of hard constraints such as time windows and capacity limitations.

For this, we designed a deep graph convolutional neural network to analyze problem instances and predict edge probabilities for each edge in the complete graph how likely it is that this edge is included in the correct solution. This deep learning model served as a tool to guide heuristic methods. By using the network's predictions, we developed a limited-width breadth-first tree search that sequentially constructed solutions. In each step, partial solutions were evaluated based on the network's output, ensuring that only the most promising branches were further explored while maintaining feasibility with respect to the side constraints. Computational results demonstrated that while this method achieved promising results for small instances, its scalability was limited. For larger problem sizes, the tree search became less effective, failing to reliably identify optimal or high-quality solutions and lagging behind state-of-the-art heuristics.

To address these shortcomings, we developed a Monte Carlo tree search algorithm. For this, we integrated techniques to balance the exploration of the whole search tree and the exploitation of the most promising areas of the search tree and leveraged the graph convolutional neural network's output in two ways: simulating the objective value of partial solutions through a limited-width breadth-first tree search and weighting search tree edges by their predicted probabilities. This iterative process allowed the Monte Carlo tree search to gather contextual information on partially constructed solutions, which were incorporated as a secondary input to the neural network to predict promising search paths more effectively. Computational results showed significant improvements over the original limited-width breadth-first tree search, with better solution quality and enhanced scalability to larger instances. An ablation study confirmed that the network's predictions enhanced the performance of the Monte Carlo tree search, validating the potential of neural

networks to improve heuristic-based methods. Nonetheless, the approach's computational runtime and scalability still fell short compared to highly specialized heuristics.

In the final phase of this research, we investigated the use of quantum-inspired computing hardware to address routing problems. By modeling the capacitated vehicle routing problem with time windows as a quadratic unconstrained binary optimization (QUBO) problem, we explored how quantum-inspired computing hardware specialized in solving QUBOs could be utilized to find solutions efficiently. Due to hardware limitations, a graph convolutional neural network was employed as a learned problem reduction, excluding the majority of edges of the complete graph to reduce the size of the problem representation. While this approach achieved comparable results to the limited-width breadth-first search for small instances, scalability remained an issue for larger problems, as the number of variables needed for representing the side constraints quickly exceeded the hardware's capabilities.

To overcome these limitations, we developed a three-phase heuristic that combines quantum-inspired computing hardware with deep learning-assisted heuristics. This method involved a deep learning-complemented QUBO formulation to cluster the set of vertices into subsets, generating sets of candidate routes for each cluster using deep learning-assisted tree searches, and finally combining these routes into a complete solution through a QUBO-based set partitioning approach. With this structure, the challenge of handling the side constraints, which proved difficult for the quantum-inspired computing hardware, was delegated to a heuristic tree search. This approach allowed the specialized hardware to focus on its strengths, solving quadratic binary problems with a smaller number of variables. The resulting three-phase heuristic demonstrated scalability to larger instances while maintaining high quality and feasibility with respect to the side constraints. This proof of concept highlighted how quantum-inspired computing and AI techniques can be combined to address complex combinatorial optimization problems. Computational experiments showed that the three-phase heuristic provided similar scalability as other state-of-the-art methods and produced near-optimal solutions for larger instances.

The findings of this thesis highlight the potential of integrating AI with heuristic methods to tackle complex optimization problems. Future research could explore several directions. Deep learning, a rapidly evolving field, offers opportunities to improve prediction models by leveraging different neural network architectures. In particular, reinforcement learning holds great promise for enhancing prediction quality in combinatorial optimization tasks. Combining reinforcement learning models with unsupervised or self-supervised learning could enable the development of models that generalize well across diverse problem instances. This could reduce the dependence on large labeled datasets and in combination with heuristics improve solution quality and computational efficiency for combinatorial optimization problems.

Moreover, each phase in the proposed three-phase heuristic offers opportunities for improvement. For instance, alternative clustering algorithms could be explored to improve the partitioning of vertices in the initial step. Similarly, different hardware and software for solving QUBOs may yield further improvements. Expanding the applicability to other

combinatorial optimization problems with hard constraints could also demonstrate the generalizability of the proposed methods.

In summary, this thesis demonstrated the potential of combining neural networks, heuristic algorithms, and emerging computing technologies. While there is still room for improvement, the proposed approaches lay a promising groundwork for addressing complex routing problems and advancing the field of combinatorial optimization.

Bibliography

[ABH13] Hakim Akeb, Adel Bouchakhchoukha, and Mhand Hifi. A beam search based algorithm for the capacitated vehicle routing problem with time windows. In Maria Ganzha, Leszek Maciaszek, and Marcin Paprzycki, editors, *Federated Conference on Computer Science and Information Systems 2013*, pages 329–336, 2013.

[AGM14] Sohaib Afifi, Rym Nesrine Guibadj, and Aziz Moukrim. New lower bounds on the number of vehicles for the vehicle routing problem with time windows. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming. CPAIOR 2014*, volume 8451 of *Lecture Notes in Computer Science*, pages 422–437. 2014.

[BCGL07] Gerardo Berbeglia, Jean-François Cordeau, Irina Gribkovskaia, and Gilbert Laporte. Static pickup and delivery problems: a classification scheme and survey. *TOP*, 15(1):1–31, 2007.

[BCM07] Roberto Baldacci, Nicos Christofides, and Aristide Mingozzi. An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts. *Mathematical Programming*, 115(2):351–385, 2007.

[BD17] Dimitris Bertsimas and Jack Dunn. Optimal classification trees. *Machine Learning*, 106(7):1039–1082, 2017.

[BDD06] Natashia Boland, John Dethridge, and Irina Dumitrescu. Accelerated label setting algorithms for the elementary resource constrained shortest path problem. *Operations Research Letters*, 34(1):58–68, 2006.

[BG05] Olli Bräysy and Michel Gendreau. Vehicle routing problem with time windows, part i: Route construction and local search algorithms. *Transportation Science*, 39(1):104–118, 2005.

[BGK+20] Michał Borowski, Paweł Gora, Katarzyna Karnas, Mateusz Błajda, Krystian Król, Artur Matyjasek, Damian Burczyk, Miron Szewczyk, and Michał Kutwin. New hybrid quantum annealing algorithms for solving vehicle routing problem. In Valeria V. Krzhizhanovskaya, Gábor Závodszky, Michael H.

Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João Teixeira, editors, *Computational Science – ICCS 2020*, volume 12142 of *Lecture Notes In Computer Science*, pages 546–561. 2020.

[BL17] Xavier Bresson and Thomas Laurent. Residual gated graph convnets. *Preprint*, arXiv:1711.07553, 2017.

[BLP21] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[BMR11] Roberto Baldacci, Aristide Mingozzi, and Roberto Roberti. New route relaxation and pricing strategies for the vehicle routing problem. *Operations Research*, 59(5):1269–1283, 2011.

[BMR12] Roberto Baldacci, Aristide Mingozzi, and Roberto Roberti. Recent exact algorithms for solving the vehicle routing problem under capacity and time window constraints. *European Journal of Operational Research*, 218(1):1–6, 2012.

[BPL+16] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural Combinatorial Optimization with Reinforcement Learning. *Preprint*, arXiv:1611.09940, 2016.

[BPPU18] Teobaldo Bulhões, Artur Pessoa, Fábio Protti, and Eduardo Uchoa. On the complete set packing and set partitioning polytopes: Properties and rank 1 facets. *Operations Research Letters*, 46(4):389–392, 2018.

[BS86] Edward K. Baker and Joanne Schaffer. Solution improvement heuristics for the vehicle routing and scheduling problem with time window constraints. *American Journal of Mathematical and Management Sciences*, 6:261–300, 1986.

[BVH07] Russell Bent and Pascal Van Hentenryck. Randomized adaptive spatial decoupling for large-scale vehicle routing with time windows. In Anthony Cohn, editor, *Proceedings of the 22nd national conference on Artificial intelligence*, volume 2, pages 173–178, 2007.

[BVH10] Russell Bent and Pascal Van Hentenryck. Spatial, temporal, and hybrid decompositions for large-scale vehicle routing with time windows. In David Cohen, editor, *Principles and Practice of Constraint Programming – CP 2010*, volume 6308 of *Lecture Notes In Computer Science*, pages 99–113, 2010.

[CCD19] Luciano Costa, Claudio Contardo, and Guy Desaulniers. Exact branch-price-and-cut algorithms for vehicle routing. *Transportation Science*, 53(4):946–985, 2019.

[Chr72] Nicos Christofides. Technical note - bounds for the travelling-salesman problem. *Operations Research*, 20:1044–1056, 1972.

[Cla99] Jens Clausen. Branch and bound algorithms - principles and examples - technical note. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

[CM14] Claudio Contardo and Rafael Martinelli. A new exact algorithm for the multi-depot vehicle routing problem under capacity and route length constraints. *Discrete Optimization*, 12:129–146, 2014.

[Cou07] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In David Hutchison, editor, *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. 2007.

[CT19] Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché Buc, and Emily B. Fox, editors, *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 6281–6292, 2019.

[DAT20] Arthur Delarue, Ross Anderson, and Christian Tjandraatmadja. Reinforcement learning with combinatorial actions: An application to vehicle routing. In Hugo Larochelle, Marc'Aurelio Ranzato, Raja Hadsell, Maria Florina Balcan, and Hsuan-Tien Lin, editors, *Proceedings of the 34th International Conference on Neural Information Processing Systems*, volume 34, pages 609–620, 2020.

[DB16] Steven Diamond and Stephen Boyd. Cvxpy: a python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research*, 17:2909–2913, 2016.

[DC07] Rodolfo Dondo and Jaime Cerdá. A cluster-based optimization approach for the multi-depot heterogeneous fleet vehicle routing problem with time windows. *European Journal of Operational Research*, 176(3):1478–1507, 2007.

[DK24] Jorin Dornemann and Tobias Klein. Graph convolutional neural network assisted monte carlo tree search for the capacitated vehicle routing problem with time windows. *Accepted for publication in a special volume of the AIRO Springer Series*, 2024+.

[DLH08] Guy Desaulniers, François Lessard, and Ahmed Hadjar. Tabu search, partial elementarity, and generalized k-path inequalities for the vehicle routing problem with time windows. *Transportation Science*, 42(3):387–404, 2008.

[DMR14] Guy Desaulniers, Oli B.G. Madsen, and Stefan Ropke. The vehicle routing problem with time windows. In Paolo Toth and Daniele Vigo, editors, *Vehicle*

Routing: Problems, Methods, and Applications, volume 2, pages 119–159. 2014.

[Dor23] Jorin Dornemann. Solving the capacitated vehicle routing problem with time windows via graph convolutional network assisted tree search and quantum-inspired computing. *Frontiers in Applied Mathematics and Statistics*, 9, 2023.

[DR59] G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Management Science*, 6(1):80–91, 1959.

[Dro94] Moshe Dror. Note on the Complexity of the Shortest Path Models for Column Generation in VRPTW. *Operations Research*, 42(5):977–978, 1994.

[DSKT24] Jorin Dornemann, Salwa Shaglel, Martin Kliesch, and Anusch Taraz. Scalable 3-phase-heuristic for solving the capacitated vehicle routing problem with time windows using quantum-inspired computing and deep learning. *Accepted for publication in Lecture Notes in Computer Science*, 2024+.

[DW87] Richard Durbin and David Willshaw. An analogue approach to the travelling salesman problem using an elastic net method. *Nature*, 326(6114):689–691, 1987.

[EQSU23] Najib Errami, Eduardo Queiroga, Ruslan Sadykov, and Eduardo Uchoa. Vrp-solvereasy: A python library for the exact solution of a rich vehicle routing problem. *INFORMS Journal on Computing*, 2023.

[ES10] Nasser A. El-Sherbeny. Vehicle routing with time windows: An overview of exact, heuristic and metaheuristic methods. *Journal of King Saud University - Science*, 22(3):123–131, 2010.

[FDGG04] Dominique Feillet, Pierre Dejax, Michel Gendreau, and Cyrille Gueguen. An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems. *Networks*, 44(3):216–229, 2004.

[FGR07] Dominique Feillet, Michel Gendreau, and Louis-Martin Rousseau. New refinements for the solution of vehicle routing problems with branch and price. *INFOR: Information Systems and Operational Research*, 45(4):239–256, 2007.

[FJ81] Marshall L. Fisher and Ramchandran Jaikumar. A generalized assignment heuristic for vehicle routing. *Networks*, 11(2):109–124, 1981.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345–345, 1962.

[FRG+19] Sebastian Feld, Christoph Roch, Thomas Gabor, Christian Seidel, Florian Neukart, Isabella Galter, Wolfgang Mauerer, and Claudia Linnhoff-Popien.

A hybrid solution method for the capacitated vehicle routing problem using a quantum annealer. *Frontiers in ICT*, volume 6, 2019.

[FST20] Jonas K. Falkner and Lars Schmidt-Thieme. Learning to solve vehicle routing problems with time windows through joint attention. *Preprint*, arXiv:2006.09100, 2020.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[FTST22] Jonas K. Falkner, Daniela Thyssens, and Lars Schmidt-Thieme. Large neighborhood search based on neural construction heuristics. *Preprint*, arXiv:2205.00772, 2022.

[GCC+20] Lei Gao, Mingxiang Chen, Qichang Chen, Ganzhong Luo, Nuoyi Zhu, and Zhixin Liu. Learn to design the heuristics for vehicle routing problem. *Preprint*, arXiv:2002.08539, 2020.

[GKD22] Fred W. Glover, Gary A. Kochenberger, and Yu Du. Quantum bridge analytics i: A tutorial on formulating and using qubo models. *Annals of Operations Research*, 314:141–183, 2022.

[GLMT99] Michel Gendreau, Gilbert Laporte, Christophe Musaraganyi, and Éric D. Taillard. A tabu search heuristic for the heterogeneous fleet vehicle routing problem. *Computers and Operations Research*, 26(12):1153–1173, 1999.

[GM74] Billy E. Gillett and Leland R. Miller. A heuristic algorithm for the vehicle-dispatch problem. *Operations Research*, 22(2):340–349, 1974.

[GN07] Keshri Ganesh and T.T. Narendran. Cloves: A cluster-and-search heuristic to solve the vehicle routing problem with delivery and pick-up. *European Journal of Operational Research*, 178(3):699–717, 2007.

[GO22] L.L.C. Gurobi Optimization. Gurobi optimizer reference manual. *https://www.gurobi.com*, 2022.

[Hel17] Keld Helsgaun. An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems: Technical report. *Roskilde Universitet*, 2017.

[HK71] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1(1):6–25, 1971.

[HNN+20] Ramkumar Harikrishnakumar, Saideep Nannapaneni, Nam H. Nguyen, James E. Steck, and Elizabeth C. Behrman. A quantum annealing approach

for dynamic multi-depot capacitated vehicle routing problem. *Preprint*, arXiv:2005.12478, 2020.

[HT85] John Hopfield and David Tank. "neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52(3):141–152, 1985.

[ID05] Stefan Irnich and Guy Desaulniers. Shortest path problems with resource constraints. In Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon, editors, *Column Generation*, pages 33–65. 2005.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on International Conference on Machine Learning*, volume 37, page 448–456, 2015.

[IWT+19] Hirotaka Irie, Goragot Wongpaisarnsin, Masayoshi Terabe, Akira Miki, and Shinichirou Taguchi. Quantum annealing of vehicle routing problem with time, state and capacity. In Claudia Feld, Sebastianand Linnhoff-Popien, editor, *Quantum Technology and Optimization Problems. QTOP 2019*, volume 11413 of *Lecture Notes in Computer Science*, pages 145–156. 2019.

[JLB19a] Chaitanya K. Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *Preprint*, arXiv:1906.01227, 2019.

[JLB19b] Chaitanya K. Joshi, Thomas Laurent, and Xavier Bresson. On learning paradigms for the travelling salesman problem. *Preprint*, arXiv:1910.07210, 2019.

[JPSP08] Mads Jepsen, Bjørn Petersen, Simon Spoorendonk, and David Pisinger. Subset-row inequalities applied to the vehicle-routing problem with time windows. *Operations Research*, 56(2):497–511, 2008.

[KAHL22] Matthew Kowalsky, Tameem Albash, Itay Hen, and Daniel Lidar. 3-regular three-XORSAT planted solutions benchmark of classical and quantum heuristic optimizers. *Quantum Science and Technology*, 7(2):025008, 2022.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *Preprint*, arXiv:1412.6980v9, 2014.

[KCK+20] Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. In Hugo Larochelle, Marc'Aurelio Ranzato, Raja Hadsell, Maria Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 21188–21198, 2020.

[KDM+99] Niklas Kohl, Jacques Desrosiers, Oli B. G. Madsen, Marius M. Solomon, and Francois Soumis. 2-path cuts for the vehicle routing problem with time windows. *Transportation Science*, 33(1):101–116, 1999.

[KKMY11] Sunyoung Kim, Masakazu Kojima, Martin Mevissen, and Makoto Yamashita. Exploiting sparsity in linear and nonlinear matrix inequalities via positive semidefinite matrix completion. *Mathematical Programming*, 129(1):33–68, 2011.

[KLMS05] Brian Kallehauge, Jesper Larsen, Oli B.G. Madsen, and Marius M. Solomon. Vehicle routing problem with time windows. In Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon, editors, *Column Generation*, pages 67–98, 2005.

[Koh90] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[KPS92] Yiannis A. Koskosidis, Warren B. Powell, and Marius M. Solomon. An optimization-based heuristic for vehicle routing and scheduling with soft time window constraints. *Transportation Science*, 26(2):69–85, 1992.

[Kru56] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. 2006.

[KvHGW22] Wouter Kool, Herke van Hoof, Joaquim Gromicho, and Max Welling. Deep policy dynamic programming for vehicle routing problems. In Pierre Schaus, editor, *CPAIOR: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, volume 13292 of *Lecture Notes In Computer Science*, pages 190–213, 2022.

[KvW19] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *7th International Conference on Learning Representations*, 2019.

[KW18] Yoav Kaempfer and Lior Wolf. Learning the multiple traveling salesmen problem with permutation invariant pooling networks. *Preprint*, arXiv:1803.09621, 2018.

[LD60] Alisa H. Land and Alison G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497, 1960.

[LK73] Song Lin and Brian W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, 1973.

[LLE04] Jens Lysgaard, Adam N. Letchford, and Richard W. Eglese. A new branch-and-cut algorithm for the capacitated vehicle routing problem. *Mathematical Programming*, 100(2):423–445, 2004.

[LN83] Gilbert Laporte and Yves Nobert. A branch and bound algorithm for the capacitated vehicle routing problem. *Operations-Research-Spektrum*, 5(2):77–85, 1983.

[LS02] Gilbert Laporte and Frédéric Semet. Classical heuristics for the capacitated vrp. In Paolo Toth and Daniele Vigo, editors, *The Vehicle Routing Problem*, pages 109–128. 2002.

[LSL90] Chung-Lun Li and David Simchi-Levi. Worst-case analysis of heuristics for multidepot capacitated vehicle routing problems. *ORSA Journal on Computing*, 2(1):64–73, 1990.

[Luc14] Andrew Lucas. Ising formulations of many np problems. *Frontiers in Physics*, volume 2, 2014.

[LZY20] Hao Lu, Xingwen Zhang, and Shuang Yang. A learning-based iterative method for solving vehicle routing problems. In *8th International Conference on Learning Representations*, 2020.

[MCF+77] Mark F. Medress, Franklin S. Cooper, Jim W. Forgie, Cordell Green, Dennis H. Klattand, Michael H. O'Malley, Edward P. Neuburg, Allen Newell, Raj Reddy, and Barry Ritea. Speech understanding systems: Report of a steering committee. *Artificial Intelligence*, 9:307 – 316, 1977.

[Mil95] Donald L. Miller. A matching based exact algorithm for capacitated vehicle routing problems. *ORSA Journal on Computing*, 7(1):1–9, 1995.

[MTM+20] Satoshi Matsubara, Motomu Takatsu, Toshiyuki Miyazawa, Takayuki Shibasaki, Yasuhiro Watanabe, Kazuya Takemoto, and Hirotaka Tamura. Digital annealer for high-speed solving of combinatorial optimization problems and its applications. In Huazhong Yang and K. T. Tim Cheng, editors, *Proceedings of the 25th Asia and South Pacific Design Automation Conference. ASPDAC 2020*, page 667–672, 2020.

[Mun57] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[Nak10] Maho Nakata. A numerical evaluation of highly accurate multiple-precision arithmetic version of semidefinite programming solver: Sdpa-gmp, -qd and -dd. *International Symposium on Computer-Aided Control System Design*, pages 29–34, 2010.

[ND86] Gordon F. Newell and Carlos F. Daganzo. Design of multiple-vehicle delivery tours—i a ring-radial network. *Transportation Research Part B: Methodological*, 20(5):345–363, 1986.

[NOST18] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence V. Snyder, and Martin Takác. Reinforcement learning for solving the vehicle routing problem. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, and Nicolo Cesa-Bianchi, editors, *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 9861–9871, 2018.

[NVBB17] Alex Nowak, Soledad Villar, Afonso S. Bandeira, and Joan Bruna. Revised note on learning algorithms for quadratic assignment with graph neural networks. *Preprint*, arXiv:1706.07450, 2017.

[ODH08] Alexander Ostertag, Karl F. Doerner, and Richard F. Hartl. A variable neighborhood search integrated in the popmusic framework for solving large scale vehicle routing problems. In María J. Blesa, Christian Blum, Carlos Cotta, Antonio J. Fernández, José E. Gallardo, Andrea Roli, and Michael Sampels, editors, *Hybrid Metaheuristics*, pages 29–42, 2008.

[ODH+09] Alexander Ostertag, Karl F. Doerner, Richard F. Hartl, Eric Taillard, and Philippe Waelti. Popmusic for a real-world large-scale vehicle routing problem with time windows. *Journal of the Operational Research Society*, 60(7):934–943, 2009.

[Ouy07] Yanfeng Ouyang. Design of vehicle routing zones for large-scale distribution systems. *Transportation Research Part B: Methodological*, 41(10):1079–1093, 2007.

[PCDU17] Diego Pecin, Claudio Contardo, Guy Desaulniers, and Eduardo Uchoa. New enhancements for the exact solution of the vehicle routing problem with time windows. *INFORMS Journal on Computing*, 29(3):489–502, 2017.

[PF22] Laurent Perron and Vincent Furnon. Or-tools routing library. *https://developers.google.com/optimization/*, 2022.

[PGDR09] Eric Prescott-Gagnon, Guy Desaulniers, and Louis-Martin Rousseau. A branch-and-price-based large neighborhood search algorithm for the vehicle routing problem with time windows. *Networks*, 54(4):190–204, 2009.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein,

Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché Buc, and Emily B. Fox, editors, *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 8024–8035, 2019.

[Pot93] Jean-Yves Potvin. *The Traveling Salesman Problem: A Neural Network Perspective*. Universite de Montreal, Centre de recherche sur les transports, 1993.

[Pou20] Julie Poullet. Leveraging machine learning to solve the vehicle routing problem with time windows. *Master Thesis, Massachusetts Institute of Technology, Sloan School of Management, Operations Research Center*, 2020.

[PPP+17] Diego Pecin, Artur Pessoa, Marcus Poggi, Eduardo Uchoa, and Haroldo Santos. Limited memory rank-1 cuts for vehicle routing problems. *Operations Research Letters*, 45(3):206–209, 2017.

[PPPU17] Diego Pecin, Artur Pessoa, Marcus Poggi, and Eduardo Uchoa. Improved branch-cut-and-price for capacitated vehicle routing. *Mathematical Programming Computation*, pages 61–100, 2017.

[PR93] Jean-Yves Potvin and Jean-Marc Rousseau. A parallel route building algorithm for the vehicle routing and scheduling problem with time windows. *European Journal of Operational Research*, 66(3):331–340, 1993.

[PR07] David Pisinger and Stefan Ropke. A general heuristic for vehicle routing problems. *Computers and Operations Research*, 34(8):2403–2435, 2007.

[Pug14] Sergejs Pugacs. A clustering approach for vehicle routing problems with hard time windows. *Master Thesis, Universidade Nova de Lisboa*, 2014.

[PWZ20] Bo Peng, Jiahai Wang, and Zizhen Zhang. A deep reinforcement learning algorithm using dynamic attention model for vehicle routing problems. In Kangshun Li, Wei Li, Hui Wang, and Yong Liu, editors, *Artificial Intelligence Algorithms and Applications*, pages 636–650, 2020.

[QLLM12] Mingyao Qi, Wei-Hua Lin, Nan Li, and Lixin Miao. A spatiotemporal partitioning approach for large-scale vehicle routing problems with time windows. *Transportation Research Part E: Logistics and Transportation Review*, 48(1):248–257, 2012.

[Reb74] Kenneth R. Rebman. Total unimodularity and the transportation problem: a generalization. *Linear Algebra and its Applications*, 8(1):11–24, 1974.

[Rus95] Robert A. Russell. Hybrid heuristics for the vehicle routing problem with time windows. *Transportation Science*, 29(2):156–166, 1995.

[RVO+14] Eleanor G. Rieffel, Davide Venturelli, Bryan O'Gorman, Minh B. Do, Elicia M. Prystay, and Vadim N. Smelyanskiy. A case study in programming a quantum annealer for hard operational planning problems. *Quantum Information Processing*, 14(1):1–36, 2014.

[Rø12] Stefan Røpke. Branching decisions in branch and-cut-and-price algorithms for vehicle routing problems. *Presentation in Column Generation*, 2012.

[Sav85] Martin W. P. Savelsbergh. Local search in routing problems with time windows. *Annals of Operations Research*, 4(1):285–305, 1985.

[SC17] Alex Syrichas and Alan Crispin. Large-scale vehicle routing problems: Quantum annealing, tunings and results. *Computers and Operations Research*, 87:52–62, 2017.

[SdSSB19] Maria Amélia Lopes Silva, Sérgio Ricardo de Souza, Marcone Jamilson Freitas Souza, and Ana Lúcia C. Bazzan. A reinforcement learning-based multi-agent framework applied for solving routing and scheduling problems. *Expert Systems with Applications*, 131:148–171, 2019.

[SELW20] Yuan Sun, Andreas Ernst, Xiaodong Li, and Jake Weiner. Generalization of machine learning for problem reduction: a case study on travelling salesman problems. *OR Spectrum*, 43(3):607–633, 2020.

[SGM22] Özlem Salehi, Adam Glos, and Jarosław Adam Miszczak. Unconstrained binary models of the travelling salesman problem variants for quantum optimization. *Quantum Information Processing*, 21, 67(2), 2022.

[ŚGSM22] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, 2022.

[SGT+09] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[Sha98] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming - CP98*, pages 417–431. 1998.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John

Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[Sol87] Marius M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265, 1987.

[SPL22] Whei Yeap Suen, Matthieu Parizy, and Hoong Chuin Lau. Enhancing a QUBO solver via data driven multi-start and its application to vehicle routing problem. In Jonathan E. Fieldsend, editor, *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, page 2251–2257, 2022.

[SRV18] Pranavi Singanamalan, Dharma Reddy, and Perni Venkataramaiah. Solution to a multi depot vehicle routing problem using k-means algorithm, clarke and wright algorithm and ant colony optimization. *International Journal of Applied Engineering Research*, 13:15236–15246, 2018.

[SUP21] Ruslan Sadykov, Eduardo Uchoa, and Artur Pessoa. A bucket graph–based labeling algorithm with application to vehicle routing. *Transportation Science*, 55(1):4–28, 2021.

[SWMU19] Masataka Sao, Hiroyuki Watanabe, Yuuichi Musha, and Akihiro Utsunomiya. Application of digital annealer for faster combinatorial optimization. *Fujitsu Scientific and Technical Journal*, 55:45–51, 2019.

[TDR+21] Tony Tran, Minh Do, Eleanor Rieffel, Jeremy Frank, Zhihui Wang, Bryan O'Gorman, Davide Venturelli, and John Christopher Beck. A hybrid quantum-classical approach to solving scheduling problems. In Jorge A. Baier and Adi Botea, editors, *Proceedings of the 9th International Symposium on Combinatorial Search*, volume 7, pages 98–106, 2021.

[TV02] Éric D. Taillard and Stefan Voss. Popmusic — partial optimization metaheuristic under special intensification conditions. In Celso C. Ribeiro and Pierre Hansen, editors, *Essays and Surveys in Metaheuristics*, volume 15 of *Operations Research/Computer Science Interfaces*, pages 613–629. 2002.

[TV14] Paolo Toth and Daniele Vigo. *Vehicle Routing: Problems, Methods, and Applications*. 2nd edition, 2014.

[vA20] Joran van Apeldoorn. A quantum view on convex optimization. *PhD Thesis, Institute for Logic, Language and Computation, University of Amsterdam*, 2020.

[VCC+18] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. *Preprint*, arXiv:1710.10903, 2018.

[VFJ15] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28, pages 2692–2700, 2015.

[VPD19] Tomáš Vyskočil, Scott Pakin, and Hristo N. Djidjev. Embedding inequality constraints for quantum annealing optimization. In Sebastian Feld and Claudia Linnhoff-Popien, editors, *Quantum Technology and Optimization Problems*, pages 11–22, 2019.

[WSL21] Yunli Wang, Sun Sun, and Wei Li. Hierarchical reinforcement learning for vehicle routing problems with time windows. In Luiza Antonie and Pooya Moradian Zadeh, editors, *Proceedings of the Canadian Conference on Artificial Intelligence*, 2021.

[YFF+12] Makoto Yamashita, Katsuki Fujisawa, Mituhiro Fukuda, Kazuhiro Kobayashi, Kazuhide Nakata, and Maho Nakata. Latest developments in the sdpa family for solving large-scale sdps. In Miguel F. Anjos and Jean B. Lasserre, editors, *Handbook on Semidefinite, Conic and Polynomial Optimization*, pages 687–713. 2012.

[YFK03] Makoto Yamashita, Katsuki Fujisawa, and Masakazu Kojima. Implementation and evaluation of sdpa 6.0 (semidefinite programming algorithm 6.0). *Optimization Methods and Software*, 18(4):491–505, 2003.