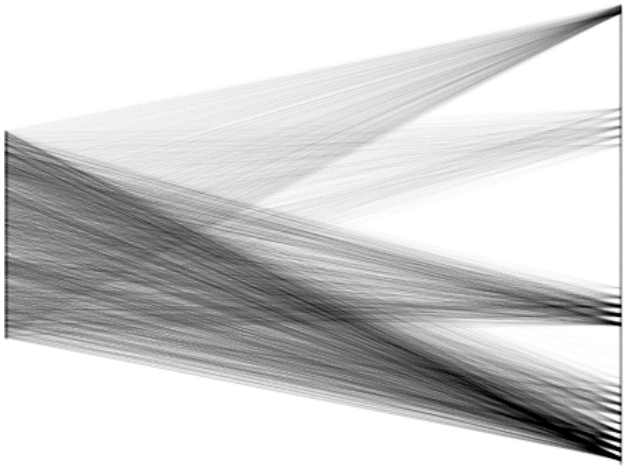

Guiding Column Generation using Deep Reinforcement Learning

Trainee and Training Device Optimization

Master's thesis in Complex Adaptive Systems

ANTON LINDÉN

DEPARTMENT OF PHYSICS

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Master's thesis 2023

Guiding Column Generation using Deep Reinforcement Learning

Trainee and Training Device Optimization

Anton Lindén

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2023

Guiding Column Generation using Deep Reinforcement Learning
Trainee and Training Device Optimization
Anton Lindén

Supervisor: Adam Wojciechowski, Jeppesen
Examiner: Mats Granath, Department of Physics

Master's Thesis 2023
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Example of a Linear Program encoded as a bipartite graph. Nodes on the left are variable nodes and nodes on the right are constraint nodes. Edges exist when a variable contributes to a constraint.

Typeset in LaTeX
Gothenburg, Sweden 2023

Guiding Column Generation using Deep Reinforcement Learning
Trainee and Training Device Optimization
Anton Lindén
Department of Physics
Chalmers University of Technology

Abstract

Many optimization problems can be formulated as a Integer Linear Program (ILP), which is an optimization problem that involves minimizing or maximizing a linear objective function subject to linear constraints and integrality requirements. Some examples include train scheduling, airline crew scheduling and production planning. ILP models with an exponentially growing set of variables are often solved using an algorithm known as Column Generation (CG). CG iteratively improves the objective function value by generating new variables, or columns, without considering every possible variable in the ILP model. This thesis was performed together with Jeppesen, a Boeing company, and investigated the possibility of using Deep Reinforcement Learning (DRL) to generate new variables in CG for a scheduling problem for airline pilots. Results show that it is possible to teach an agent a policy that slightly improves the quality of the generated variables in this specific problem. However, it is still unclear whether or not the benefits of using DRL outweighs the extra effort of setting up and training an agent.

Keywords: Scheduling Problem, Integer Linear Programming, Column Generation, Graph Neural Networks, Deep Reinforcement Learning.

Acknowledgements

I would like to thank Jeppesen for giving me this opportunity to write my Master's thesis with them. More specifically I would like to thank the Manpower team for teaching me about the Trainee and Training Device Optimization solver, and my supervisor Adam Wojciechowski for all the advice, discussions and support. I would also like to thank my examiner Mats Granath for his valuable feedback.

Anton Lindén, Gothenburg, August 2023

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

CG	Column Generation
DRL	Deep Reinforcement Learning
GNN	Graph Neural Network
ILP	Integer Linear Programming
LP	Linear Programming
MLP	Multi Layer Perceptron
RMP	Restricted Master Problem
TTDO	Trainee and Training Device Optimization

Contents

List of Figures

List of Tables

1
Introduction

This thesis was performed together with Jeppesen in Gothenburg. Jeppesen is a fully owned subsidiary of the Boeing Company, offering software solutions for aviation, including commercial aviation. One among the many solutions they provide, is a software solution for creating training schedules for airline pilots which is called the Trainee and Training Device Optimization (TTDO) solver.

Airline pilots are typically qualified for one aircraft type at a time. They are required to perform training on a yearly basis in order to maintain their aircraft qualifications [29]. Training consists of various training events such as attending lessons in class rooms and flying in flight simulators. The training is supposed to mimic real-life conditions as close as possible, and pilots are therefore paired together as captain and first officer. Training generally ends with an examination event in a flight simulator. The flight simulators used for training are very expensive resources and scheduling of these is important to ensure they are used efficiently. Airlines also want to minimize training duration since they lose manpower when pilots are away performing training. This leaves us with an optimization problem where the objective is to train pilots as efficiently as possible with the available resources (class rooms and flight simulators).

Jeppesen's TTDO solver aims to find an optimal set of rollouts for the pilots, where a rollout is a sequence of training events. A small example of a set with two rollouts is illustrated in Figure 1.1.

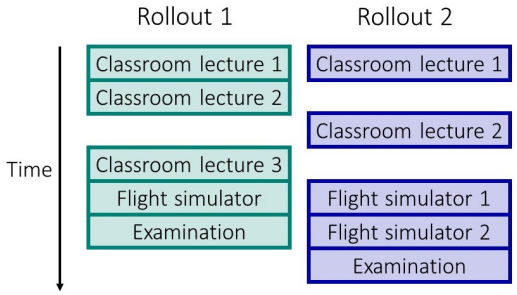

Figure 1.1: Example of a set containing two rollouts.

In reality it is hard to find an optimal set of rollouts since we quickly end up with a combinatorial explosion if there are many pilots in need of training. Most problems

similar to the TTDO problem are NP-hard and exact solutions are intractable. Instead, heuristic and approximation algorithms are used instead to find a good enough solution quickly [20].

The TTDO problem can be modeled as an Integer Linear Program (ILP), which is an optimization problem that involves minimizing or maximizing a linear objective function subject to linear constraints and integrality requirements. A wide variety of problems can be formulated and solved using ILPs (see [22]). Some examples include:

- Train scheduling.
- Airline crew scheduling.
- Production planning.
- Telecommunications.

As mentioned previously, the TTDO problem is hard to solve since we end up with a combinatorial explosion, or an exponentially growing set of variables in the ILP model. Jeppesen's approach to this is to utilize an algorithm known as Column Generation (CG) which iteratively improves the objective function value by generating new variables, or columns. The advantage with CG is that we don't have to solve the ILP with the full set of variables, instead we iteratively build up a subset of variables by only adding new variables which have potential to improve the objective function value. Details about the TTDO problem, ILPs and CG are described in Chapter 2.

CG allows for the use of various algorithms and techniques to generate new variables. A common technique is to add new variables that have the most negative reduced cost (see Section 2.1.1), but other more advanced algorithms can also be used, for example Deep Reinforcement Learning (DRL). Using DRL as part of a CG framework has, for instance, previously been done by C.Chi et al. [6]. Their DRL agent was tested on two canonical problems - the Cutting Stock Problem and the Vehicle Routing Problem with Time Windows. The agent was able to converge faster in both problems compared to a greedy strategy where variables with the most negative reduced cost were added. In other words, if one had to terminate CG earlier, their DRL algorithm would result in a better (smaller) objective function value as compared to what the greedy strategy would achieve. This thesis will attempt to use a similar DRL approach as [6], but applied to Jeppesen's TTDO problem instead.

1.1 Aim and Research Questions

The aim of this thesis is to investigate the possibility of using a DRL algorithm for generating variables in the TTDO solver. This alternative solver might be able to increase the quality of the generated variables, as was shown by [6], or at the very least provide an alternative solution. The following research questions will be answered to fulfill the aim:

- How viable is it to use DRL for generating variables in the TTDO solver?
- Is it possible to increase the quality of the generated variables in the TTDO solver using DRL?

1.2 Delimitations

This thesis will only attempt to use DRL for generating new variables in the TTDO solver, even though there are various other algorithms and techniques (see [4, 25, 1]) that can be used that also show potential to increase the quality of generated variables. The work in this thesis is inspired by previous work done by [6].

Furthermore, Jeppesen's TTDO solver is designed to create training schedules for airline pilots based on input data containing information about the training demand and the available resources, among other things. This thesis will only work with a single dataset from one specific airline. Due to confidentiality agreements, details regarding the dataset or the airline will not be covered here.

1.3 Ethical aspects

It is important to consider ethical aspects when using software for creating work-schedules for humans, especially when Machine Learning algorithms are involved [30]. Scheduling can influence a worker's total hours, pay, mental health, and ability to make plans in their personal lives. Machine Learning algorithms also have a risk of creating a black-box model, where it is uncertain how or why we arrived at a proposed solution [3]. More specifically, DRL algorithms utilize deep Neural Networks which can consist of a lot of trainable parameters depending on the complexity of the task at hand. They are typically always black-box models, unless dealing with very simple neural networks, and should therefore be used responsibly. Nevertheless, here we apply Machine Learning to the restricted task of solving an optimization problem where there is no disadvantage in finding a better solution.

2
Theory

This chapter initially provides theory about Linear Programs and Column Generation (CG), which is useful for understanding how the Training Device Optimization (TTDO) solver works. The TTDO solver is then described in detail followed by theory about Deep Reinforcement Learning.

2.1 Linear Programs

A Linear Program (LP) is an optimization problem that involves minimizing or maximizing a linear objective function subject to linear constraints [22]. The general form of an LP is presented in (2.1), where c_i, a_{ij} and b_j are constants and x_i are variables.

$$\begin{aligned} \text{minimize} \quad & \sum_{i \in I} c_i x_i, \\ \text{subject to} \quad & \sum_{i \in I} a_{ij} x_i \leq b_j, \quad \forall j \in J, \\ & x_i \geq 0, \quad \forall i \in I. \end{aligned} \tag{2.1}$$

LPs can be solved using various algorithms such as Primal simplex, Dual simplex and the Interior-point method. Nowadays commercial solvers use several algorithms simultaneously and simply return the solution from the algorithm which finishes first. This is known as Concurrent Optimization [14]. However, LPs with an exponentially growing set of variables are hard to solve with an exact solution using the aforementioned algorithms, and other algorithms such as Column Generation are typically called for instead, see Section 2.2 for details about Column Generation.

2.1.1 Duality

Duality deals with pairs of LPs and the relationship between their solutions [12]. The pairs are known as the Primal and the Dual LP [12]. By calling (2.1) our Primal, we obtain the corresponding Dual as:

$$\begin{aligned} \text{maximize} \quad & \sum_{j \in J} b_j u_j, \\ \text{subject to} \quad & \sum_{j \in J} a_{ij} u_j \geq c_i, \quad \forall i \in I, \\ & u_j \geq 0, \quad \forall j \in J. \end{aligned} \tag{2.2}$$

Note that (2.2) turns into a maximization problem with new variables, u_j, known as dual variables. The dual variable values, often simply called dual values, are

multipliers associated with the constraints in (2.1). They represent the sensitivity of the objective function value with respect to changes in the right-hand side of the constraints. A higher dual value indicates that a constraint has a greater impact on the objective function value [18].

Another fundamental concept of Linear Programming is the reduced cost, $\overline{c_i}$, of a variable. A positive reduced cost is the amount by which c_i would have to be reduced before it would be possible for a corresponding variable, x_i, to assume a positive value in the optimal solution. It is defined as:

$$\overline{c_i} = c_i - \sum_{j \in J} a_{ij} u_j \tag{2.3}$$

A negative reduced cost indicates that increasing the corresponding variable, x_i, will improve the objective function value. If all variables have non-negative reduced costs the solution is optimal.

2.1.2 Integer Linear Programs

A Integer Linear Program (ILP) is an optimization problem that involves minimizing or maximizing a linear objective function subject to linear constraints *and* integrality requirements. Where integrality requirement means that the variable can only be assigned integer values. The general form of an ILP is the same as (2.1) except that we further constrain our variables such that:

$$x_i \in \mathbb{Z}, \quad \forall i \in I. \tag{2.4}$$

Some problems can even model the integer variables as binary variables for convenience. Many decision-making problems can be modeled with binary variables since they can represent whether or not a certain action is taken.

ILPs are generally solved by first removing the integrality requirements which gives us an LP instead. This is known as an LP-relaxation. LPs can be solved more easily using the algorithms mentioned earlier in Section 2.1. The solution to the LP gives us a lower bound for a minimization problem, or upper bound for a maximization problem, on the optimal objective function value of the ILP [11]. The integrality requirements are then reintroduced using a branch-and-bound framework. The idea is to iteratively branch and resolve the LP until all the variables are integers, which gives us a solution to the ILP [13].

2.2 Column Generation

Column Generation (CG) is an algorithm used to iteratively solve LPs with an exponentially growing set of variables, or columns. The concept behind CG is to solve LPs with only a subset of all the variables instead of considering all of the variables explicitly. We do this by formulating a Restricted Master Problem (RMP),

which is a decomposed version of the original problem, and consists of only a subset of variables. Any solution to RMP is a feasible solution for the original problem. The RMP is typically initialized using heuristics or domain knowledge. New variables are then iteratively added to the RMP in order to improve the objective function value [17]. Each iteration of CG involves two steps:

1. Solve the current RMP in order to determine the current optimal objective function value and the current dual values.
2. Generate new variables which have potential to improve the objective function value (negative reduced cost), and add them to the RMP.

New variables are generated using a problem specific sub-problem [17]. The Sub-problem generally uses the dual values from the solution to the current RMP in order to generate variables with a negative reduced cost. Various stop conditions can be used such as iterating a fixed amount of times or stopping when the Sub-problem is no longer able to generate variables with a negative reduced cost. An illustration of the CG process is shown in Figure 2.1.

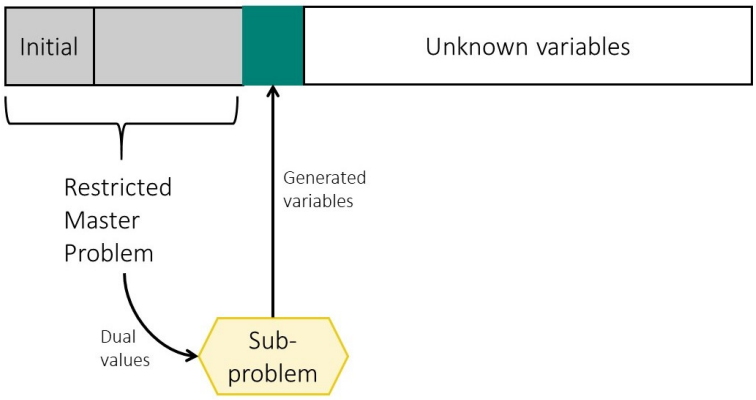

Figure 2.1: Illustration of the process of Column Generation. New variables are iteratively added to the Restricted Master Problem in order to improve the objective function value.

2.3 Trainee and Training Device Optimization Problem

Jeppesen's Trainee and Training Device Optimization (TTDO) problem is modeled as an ILP with binary decision variables that indicate if a certain rollout should be included in the solution or not. Pilots are generally paired as captain and first officer during training to mimic real-life conditions as close as possible. The TTDO solver takes this into account by assigning rollouts to *pairs* of captain and first officer, known as a natural pair:

(captain, first officer)

This is of course not always possible, and sometimes rollouts are assigned to *unnatural* pairs as well:

(captain, captain), (first officer, first officer)

We can also model a single pilot as a stand-in participant if pairing is not possible, but we generally want to use natural pairings as much as we can because it is most cost efficient. The TTDO problem when modeled as an ILP is presented in (2.5) below and the denotations can be found in Table 2.1.

Table 2.1: Denotations for the ILP formulated in (2.5)

I	Set of unique rollouts.
S	Set of resource slots (available times for class rooms and flight simulators).
D	Set of training demands.
c_i	Cost of rollout i.
x_i	Decision variable ($= 1$ if rollout i is used, $= 0$ otherwise).
I_s	Set of rollouts occupying resource slot s.
I_d	Set of rollouts satisfying demand d.

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i \in I} c_i x_i, \\
\text{subject to} \quad & \sum_{i \in I_s} x_i \;\leq\; 1, \qquad \forall s \in S, \\
& \sum_{i \in I_d} x_i \;=\; 1, \qquad \forall d \in D, \\
& x_i \;\in\; \{0,1\}, \quad \forall i \in I.
\end{aligned}
\tag{2.5}
$$

The first constraint in (2.5) ensures that the resource slots are not double-booked and the second constraint ensures we try to satisfy the training demand. The cost c_i is representative of the quality of a rollout and depends on various factors such as total training duration and number of rest days (see Section 2.3.1). Figure 2.2 and Table 2.2 below show a small example of a TTDO problem with 4 resource slots and 2 training demands modeled as an ILP.

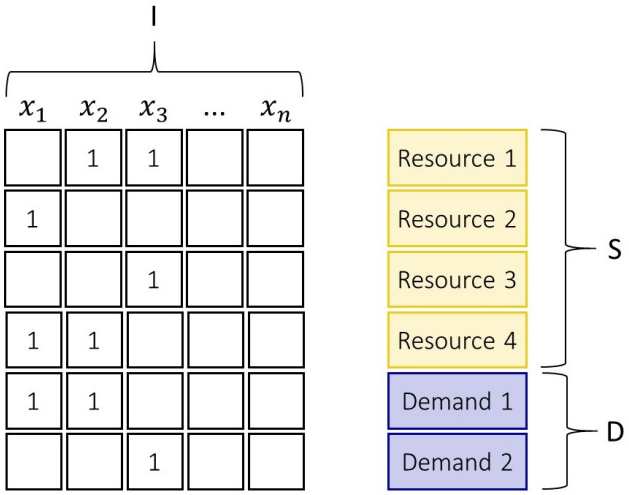

Figure 2.2: Example of a Trainee and Training Device Optimization problem modeled as a Integer Linear Program. There are 2 training demands which we want to cover using the 4 available resource slots.

Table 2.2: Occupied resource slots and satisfied demands for the variables: x_1, x_2 and x_3, as shown in Figure 2.2.

$$
\begin{aligned}
I_{s=1} &= \{2, 3\} \\
I_{s=2} &= \{1\} \\
I_{s=3} &= \{3\} \\
I_{s=4} &= \{1, 2\} \\
I_{d=1} &= \{1, 2\} \\
I_{d=2} &= \{3\}
\end{aligned}
$$

In this case $x_1 = 1$, $x_2 = 0$ and $x_3 = 1$ could be a potential solution since both training demands are satisfied and no resource slots are double-booked. From Figure 2.2 we also observe that the number of variables increases exponentially when the set of resource slots, S, increases.

2.3.1 Costs and Rules for Generated Rollouts

The quality of a training schedule is determined by the total cost of the rollouts, as seen in (2.5) above. A lower total cost is associated with a higher quality schedule, which is why the formulated ILP is a minimization problem. Cost c_i for a certain rollout i can depend on various factors, for example:

- Total training duration.
- Number of days off.
- Training events that start very early in the morning or very late in the evening (results in a penalty).
- Number of location changes between various training facilities.

Rollouts are typically also constrained by a set of rules that must be satisfied, for example:

- Students can only train for a maximum of 5 days in a row.
- Students must rest at least 16h between training events.

Both the cost function and the set of rules can be changed/tuned to satisfy the airline in question.

2.3.2 Solver

Jeppesen's TTDO solver is based on CG, see Section 2.2. The RMP is initialized using heuristics and is a LP-relaxation of (2.5). The solver has two phases:

1. Iterate CG for n iterations.
2. Reintroduce integrality requirements with a branch-and-bound framework.

The TTDO solver's Sub-problem for generating new variables involves a cost and rule driven shortest-path search. From now on we simply refer to this Sub-problem as the variable generator. The concept behind the generator is to build sequences of training events, or rollouts, by searching a graph with training events represented as nodes. The generator repeats the shortest-path search for all pairs of pilots, which results in a set of rollouts, or variables.

2.4 Neural Networks

A neural network is a computational model inspired by the structure of the human brain. It is built using many computational units, known as neurons. These neurons generate outputs as a function of their inputs. The outputs can be modified by adjusting weights and biases, which enables us to train a network of interconnected neurons to produce a desired output [19][28]. The terms weights and biases are commonly referred to as the parameters of a neural network. Some applications (see [15]) of neural networks include:

- Image classification.
- Speech recognition.
- Natural language processing.
- Recommender systems.
- Generative models.

2.4.1 Multi Layer Perceptrons

Multi Layer Perceptrons (MLPs) are a fundamental class of neural networks. They are built using several layers of neurons and information is passed in one direction through the network. A MLP consists of an input layer, one or more hidden layers (intermediate layers), and a final output layer. The computation in each neuron

involves a weighted sum of the inputs from the previous layer, a bias and an activation function, as formulated in (2.6). The input layer, denoted $y^{(0)}$ in this case, does not have any computations.

$$y_i^{(l+1)} = \sigma(\sum_j w_{ij}^{(l+1)} y_j^{(l)} + b_i^{(l+1)}) \tag{2.6}$$

where w_{ij} represents the weight between neuron j and neuron i, b_i is the bias, and σ is the activation function. The activation function introduces nonlinearity. An example of a MLP with two hidden layers is illustrated in Figure 2.3.

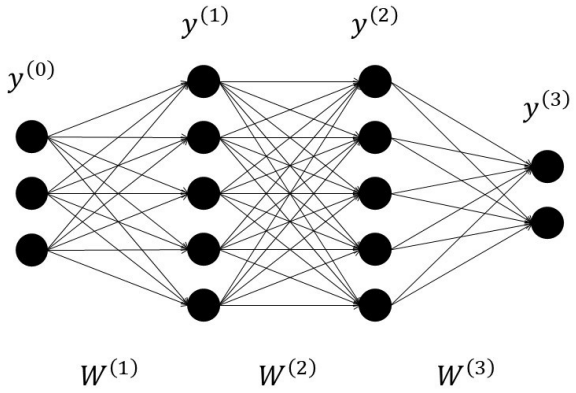

Figure 2.3: Example of a Multi Layer Perceptron with two hidden layers and weights $W^{(l)}$ for layer l.

2.4.2 Graph Neural Networks

Graph Neural Networks (GNNs) are a class of neural networks specifically designed to operate on graph-structured data. Graph-structured data is typically defined as a set of nodes, edges connecting the nodes, and a feature vector for each node. GNNs generally rely on a message-passing framework, where each node in the graph receives information, or messages, from its neighbours and updates its own features based on the information received. Note that MLPs pass information through each layer in a sequential manner, while GNNs aggregate information in each node iteratively [2]. This allows GNNs to work with graphs of varying sizes.

We use the relatively simple feature update rule from [5] as an example. For a single node i in the graph we aggregate the features from its neighbours, $j \in \mathcal{N}(i)$, and apply a linear transformation with weights W_1. We also apply a linear transformation to its own features with weights W_2. The updated features for node i are then given by (2.7):

$$x_i' = \sigma(W_2 x_i + W_1 \sum_{j \in \mathcal{N}(i)} x_j) \tag{2.7}$$

where x_i and x_j are the feature vectors for node i and j respectively, and σ is the activation function. This update rule is also illustrated in Figure 2.4 below.

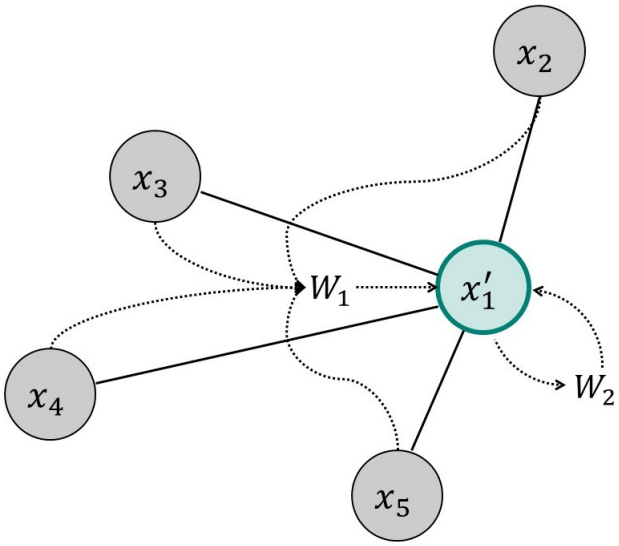

Figure 2.4: Feature update for a single node x_1 using the GraphConv update rule from [5].

2.4.3 Training

Neural networks are generally trained using an algorithm known as backpropagation, which adjusts the network parameters according to some error measure of the output signal, often called loss function. Backpropagation relies on updating each parameter using gradient descent [7]. If we let $e(y, t)$ denote an error measure between the output signal y and the targets t we get the following update rule for the weights:

$$w_{ij} = w_{ij} - \alpha \frac{\delta e(y,t)}{\delta w_{ij}} \tag{2.8}$$

where α is known as the learning rate. The learning rate plays a vital role in the training of a neural network as it significantly affects the performance. Biases are updated in the same fashion as the weights.

2.5 Reinforcement Learning

The theory in this section is based on *"Playing Atari with Deep Reinforcement Learning"* [31]. Reinforcement Learning is a sub-field of Machine Learning. The idea behind Reinforcement Learning is to make use of an agent that interacts with an environment in a sequence of actions, state transitions and rewards, as illustrated in Figure 2.5.

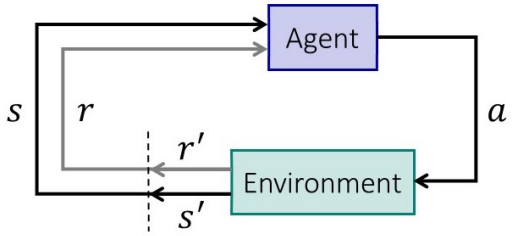

Figure 2.5: Underlying concept of Reinforcement Learning, where we have an agent interacting with an environment in a sequence of actions, state transitions and rewards.

where

- **State (s)** Current state of the environment that the agent perceives.
- **Reward (r)** Feedback signal provided by the environment.
- **Action (a)** Decision or choice made by the agent in response to the current state of the environment.
- **State-transition probability (p)** Probability of transitioning from s to s' using action a.

These components give us a Markov Decision Process [8] under the assumption that p does not depend on earlier states and actions, or in other words, there is no memory. The goal of the agent is to learn a policy that maximizes cumulative rewards, e.g. play an Atari game as well as possible. We assume future rewards are discounted by a factor of γ. The discount factor γ controls the trade-off between immediate rewards and potential future rewards.

The value of taking a specific action while being in a specific state can be defined using an action-value function, denoted as $Q(s, a)$. The optimal action-value function, denoted as $Q^*(s, a)$ instead, would give us the maximum cumulative discounted reward achievable by following any strategy after taking some action a. The optimal action-value function can be expressed using the Bellman equation:

$$Q^*(s, a) = \sum_{s' \in S} p(s', r'|s, a)[r' + \gamma max_{a'} Q^*(s', a')] \tag{2.9}$$

where $max_{a'} Q^*(s', a')$ is the maximum action-value achievable in the next state s' over all possible actions a', and $p(s', r'|s, a)$ is the state-transition probability. In Reinforcement Learning we typically want to estimate the action-value function $Q(s, a) \approx Q^*(s, a)$, so that our agent can learn an effective policy. The action-value function can be estimated using a non-linear function approximator, such as a deep neural network. This gives us a variant of Reinforcement Learning known as Deep Reinforcement Learning (DRL).

2.5.1 Deep Reinforcement Learning

A neural network can serve as a function approximator for the action-value function, $Q(s, a)$. Some benefits of using neural networks as function approximators is that they can capture more complex state representations and have the ability to generalize. They can be trained by minimizing an error measure:

$$e = (t - Q(s, a))^2 \tag{2.10}$$

where $t = r' + \gamma max_{a'} Q(s', a')$ is the target value according to (2.9). DRL also utilizes a technique known as experience replay [32] where we store the agent's actions, state transitions and rewards in a buffer, denoted B, such that:

$$B = e_1, e_2, ..., e_N \tag{2.11a}$$

$$e_n = (s_n, a_n, s'_n, r'_n) \tag{2.11b}$$

where e_n is an experience from the agent's interactions with the environment, and N is the buffer size. The idea is to randomly sample experiences from the buffer after each step in the environment, and use these experiences to train our neural network. This ensures we keep training on past experiences and avoid getting stuck in some local optimum. Experience replay also allows for greater data efficiency since we can sample and reuse the same experiences multiple times.

We also want to encourage exploration in the early stages of training, and let our agent exploit a learned policy in the later stages of training [32]. DRL typically handles the balance between exploration and exploitation by using an epsilon-greedy strategy. The epsilon-greedy strategy makes our agent take random actions with probability ϵ. This probability is gradually reduced over the course of the training, which leads to increasingly deterministic actions.

3

Methods

This chapter presents the suggested Deep Reinforcement Learning (DRL) approach to the Trainee and Training Device Optimization (TTDO) problem. An overview of the framework is presented first, followed by details about the DRL algorithm.

3.1 Framework

Jeppesen's TTDO solver is based on Column Generation (CG) and has two phases (see Section 2.3.2). The idea behind the DRL framework suggested here is to call the TTDO solver's variable generator **twice**, and let an agent decide which of the two sets of variables to add to the Restricted Master Problem (RMP). The variable generator gives us a unique set of variables each time it is called since it can only generate a specific variable once. This means we can obtain two different sets of variables, and hopefully teach an agent a policy that chooses the most appropriate set. Figure 3.1 shows a high-level illustration of the DRL framework.

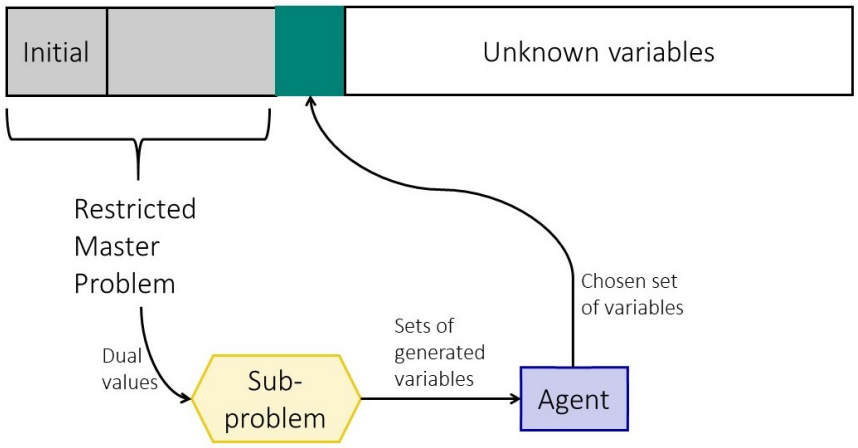

Figure 3.1: Suggested Deep Reinforcement Learning framework for generating new variables in Column Generation. The Sub-problem returns two sets of variables for an agent to decide which set to add to the Restricted Master Problem.

3.2 Deep Reinforcement Learning Formulation

In this section we formulate CG as a Markov Decision Process with states, actions and rewards.

3.2.1 State Space and Actions

The state represents the information that the agent has about the environment. In our case the environment is the CG process, similar to the environment designed by [6], but for the TTDO solver instead. The information passed to the agent is the current RMP, along with the suggested variables that we want to add. The RMP is always a LP, and it can be encoded as a bipartite graph [24]. A bipartite graph is defined as a set of nodes which is divided into two disjoint subsets such that there exists no edges between nodes belonging to the same subset [33]. We encode a LP with two node types: variable nodes $\mathcal{V}$ and constraint nodes $\mathcal{C}$. An edge (v, c) exists between a node $v \in \mathcal{V}$ and a node $c \in \mathcal{C}$ when variable v contributes to constraint c. The suggested variables from the variable generator are included in $\mathcal{V}$. Appendix A.3 shows a bipartite graph which was created after 10 iterations of CG with Jeppesen's TTDO solver.

Node features were designed for both the variable nodes and the constraint nodes in order to encode richer information in the bipartite graph. Some of the features were inspired by [6], others were designed specifically for the TTDO problem. Refer to Appendix A.1 and A.2 for a detailed description of all the node features.

Considering we want to let an agent decide between two sets of variables at each iteration of CG, we represent a state as two bipartite graphs, as shown in Figure 3.2 below.

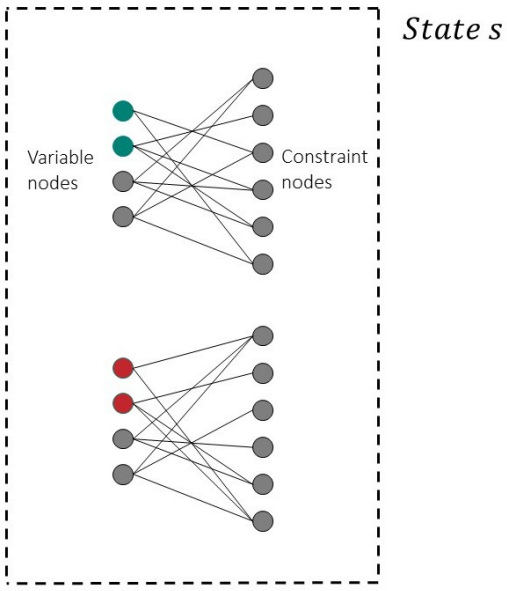

Figure 3.2: A state is represented by two bipartite graphs that encode the current Restricted Master Problem along with the suggested variables that we want to add. Green nodes are the first set of suggested variables and red nodes are the second set of suggested variables.

Our agent is given two actions at each state: add the first set of suggested variables or the second set of suggested variables. Transitioning from $s \rightarrow s'$ is done by adding the chosen set of variables to the RMP and resolving it, which gives us the next iteration of CG where we obtain two new sets of suggested variables, and two new bipartite graphs, s'.

As our states consist of two bipartite graphs, it is natural to use a Graph Neural Network (GNN) as an approximator for the action-value function, $Q(s, a)$, used in DRL. We want to obtain action-values for each of the bipartite graphs so we can choose which of the two sets of suggested variables to add to the RMP. Using a GNN also allows us to take advantage of message-passing between variable nodes, $\mathcal{V}$, and constraint nodes, $\mathcal{C}$, and hopefully be able to capture their interactions with each other. The GNN which was used in this thesis is described in more detail in Section 3.3.

3.2.2 Reward Function

The reward (3.1) is the change in the RMP objective function value in each iteration of CG.

$$r' = |\text{obj}' - \text{obj}| \tag{3.1}$$

where obj' is the objective function value of the RMP after adding the chosen set of variables and resolving it. Maximizing cumulative rewards is equivalent to improving the RMP objective function value as much as possible over the entire CG process.

3.3 Graph Neural Network

A Graph Neural Network (GNN) was used as a function approximator for the action-value function, Q(s, a). The input to the GNN is a bipartite graph which represents the encoded state of the current RMP, along with the suggested variables. The feature update is done separately for the two node types, as was done in [6]. For variable nodes we aggregate the features from neighbouring constraint nodes and apply a non-linear activation function (ReLU). Constraint nodes follow the same procedure, but instead we aggregate the features from neighbouring variable nodes. This can be seen as message-passing from constraint nodes to variable nodes, and vice versa. We end up with two matrices that represent the updated features of each node, often called a feature map, with sizes:

$$\begin{array}{ll} (\dim(\mathcal{V}), x'_v) & \text{for variable nodes} \\ (\dim(\mathcal{C}), x'_c) & \text{for constraint nodes} \end{array}$$

where x'_v and x'_c are the updated feature vectors for the variable nodes and the constraint nodes respectively. The feature maps are then pooled across the nodes using mean pooling, which in turn gives us feature vectors for each node type, or an embedding of each node type. As a final step we concatenate the two feature vectors and pass them through a Multi Layer Perceptron (MLP) with a single output neuron. This gives us an action-value for *one* bipartite graph. Figure 3.3 below shows a high-level illustration of our GNN.

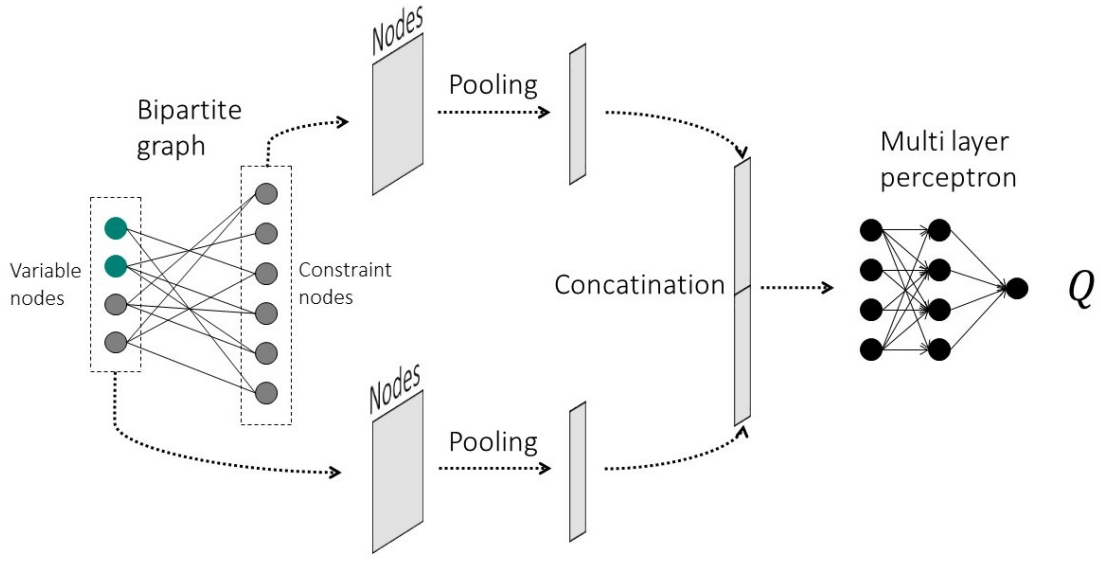

Figure 3.3: Graph Neural Network for obtaining an action-value from a bipartite graph. Green nodes are suggested variables for adding to the Restricted Master Problem.

3.4 Training

The agent was trained by interacting with the CG process of the TTDO solver for a number of episodes. Each episode consists of iterating CG for n iterations and then restarting the TTDO solver. Each iteration of CG involves adding a set of variables to the RMP and resolving it, resulting in a state transition and a reward. Experience replay was used [32] with mean squared error (2.10) between the action-values $Q(s, a)$ and $\hat{Q}(s, a)$, where $\hat{Q}(s, a)$ is a separate GNN, known as a target network, that stabilizes training by providing a fixed target for the main GNN [31]. The target network was updated using soft updates [9]. An epsilon-greedy strategy was used for encouraging exploration during the early stages of training. The hyperparameters used for training the agent are listed in Appendix A.4.

The agent was trained on the first 20 iterations of CG for 500 episodes. The TTDO solver can be run for more than 20 iterations of CG, resulting in even lower objective function values (exponentially decaying). However, training our agent on more than 20 iterations of CG would take too long and was not viable due to several reasons:

- The bipartite graphs representing the encoded state of the RMP become too large eventually.
- Some limitations in hardware.
- Sub-optimal coding.

The agent's progress during training can be viewed in Appendix A.6.

3.4.1 Offline Environment

The architecture of the GNN, such as the number of layers, number of neurons, and the operator for updating node features, was designed in a separate offline environment. In this case we refer to an offline environment as an environment that is detached from the TTDO solver and does not involve DRL either. Bipartite graphs were first sampled from the TTDO solver using random actions, giving us a reasonably sized dataset to work with. The bipartite graphs were then labeled by the objective function value obtained from solving the RMP. This allows us to define a regression task where we can train our GNN, as presented in Section 3.3, on the obtained dataset using mean squared error between output and label:

$$e = (y - y_{\text{expected}})^2 \tag{3.2}$$

where y is the output from the GNN and y_{expected} is the corresponding label. The dataset was partitioned into training and validation sets as customary. The performance of the GNN was then monitored and the design of the GNN was adjusted accordingly. The training progress for this regression task can be found in Appendix A.5.

3.5 Performance Evaluation

The trained agent's performance was evaluated in 3 different ranges of CG iterations:

- $0 - 10$ iterations.
- $10 - 20$ iterations.
- $10 - 100$ iterations.

The agent's performance was benchmarked against two strategies, denoted Simple 1 & 2 for convenience:

- **Simple 1** Always add the first set of suggested variables. This is equivalent to Jeppesen's original TTDO solver.
- **Simple 2** Always add the second set of suggested variables.

where the two sets of suggested variables are the ones described in Section 3.1. These benchmarks allow us to assess the differences between the two sets of suggested variables, and to see whether or not our agent is able to leverage them both. The first benchmark, Simple 1, also allows us to compare our agent's performance with the original TTDO solver, since the original TTDO solver calls the variable generator only once.

Moreover, performance was measured by the Linear Program (LP) objective function values at each iteration of CG. The Integer Linear Program (ILP) objective function values, which are obtained after using branch-and-bound for reintroducing integrality requirements, were also measured once the CG iterations were complete. The LP objective function values show the performance of each strategy during CG, which is where our agent is trained, and the ILP objective function values are just an extra metric indicating the quality of the final training schedule.

The reason for not always deploying our agent in the initial 10 CG iterations is because there is no noticeable difference between any of the 3 strategies (Agent, Simple 1, Simple 2) in this range. The RMP is still relatively small and adding any set of variables results in a large improvement of the objective function value. However, this behaviour is most likely related to the dataset used in this thesis, and doesn't necessarily always have to be true.

The agent's performance was evaluated in 100 iterations of CG even though it was only trained on the first 20 iterations in order to test its ability to generalize.

4

Results

This chapter presents the experimental results from the Deep Reinforcement Learning (DRL) approach to the Trainee and Training Device Optimization (TTDO) problem. An agent was trained according to Section 3.4 and thereafter evaluated in 3 different ranges of Column Generation (CG) iterations, as described in Section 3.5.

4.1 0-10 Iterations

Figure 4.1 below shows a convergence plot which was obtained after running the TTDO solver using each of the 3 different strategies - **Agent**, **Simple 1** and **Simple 2** for 10 iterations of CG. Note that the result is only for a single instance of the problem. There was only minor variance between the strategies and instances and the order, i.e best to worst, of the LP objective function values always remained the same.

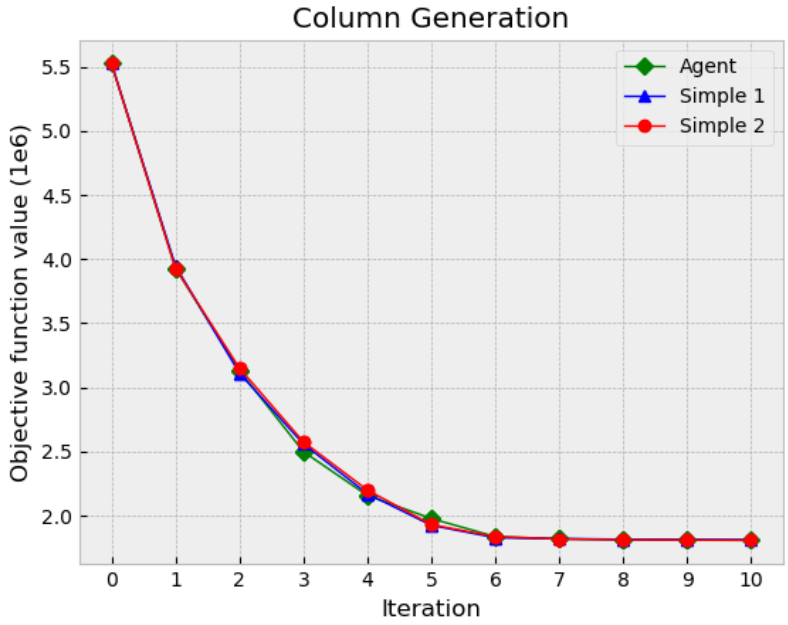

Figure 4.1: Column Generation for 10 iterations. The agent's performance is benchmarked against **Simple 1:** always add the first set of suggested variables, and **Simple 2:** always add the second set of suggested variables.

Table 4.1: Objective function values after 10 iterations of Column Generation. The Integer Linear Program (ILP) objective function value is obtained after using branch-and-bound for reintroducing integrality requirements.

	LP objective function value [1e6]	ILP objective function value [1e6]
Agent	1.8086	1.8544
Simple 1	1.8128	1.8414
Simple 2	1.8087	1.8499

4.2 10-20 Iterations

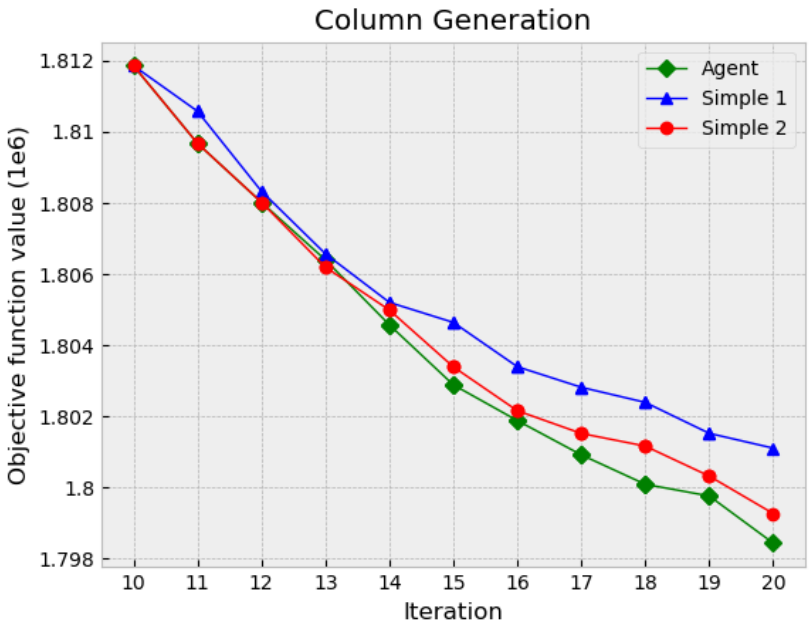

Figure 4.2: Column Generation for 20 iterations.

Table 4.2: Objective function values after 20 iterations of Column Generation.

	LP objective function value [1e6]	ILP objective function value [1e6]
Agent	1.7984	1.8121
Simple 1	1.8011	1.8152
Simple 2	1.7993	1.8178

4.3 10-100 Iterations

Figure 4.3 below shows a convergence plot which was obtained after running the TTDO for 100 iterations of CG in order to test the agents ability to generalize. There was only minor variance between the strategies and instances for the first 20 iterations of CG, but further iterations showed increasingly more variance. For example, the order of the LP objective function values after 30+ iterations looked to be random.

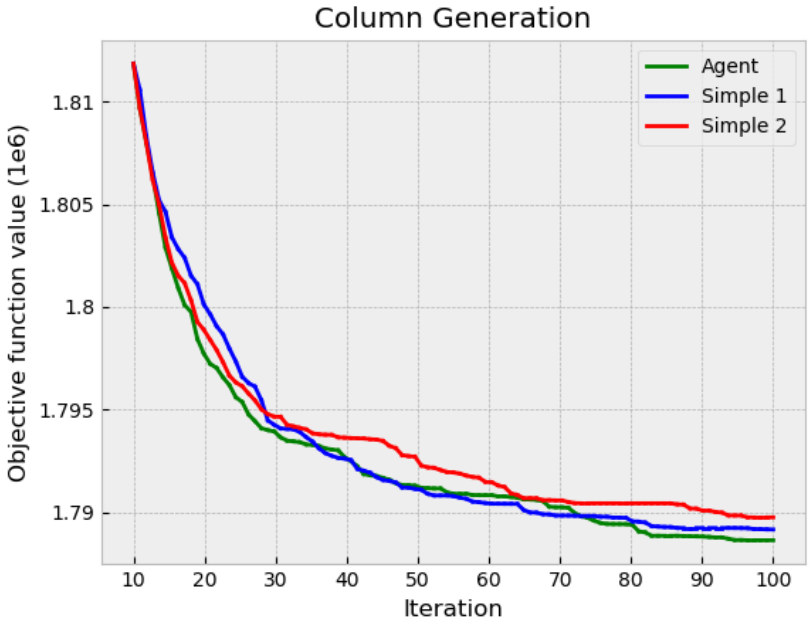

Figure 4.3: Column Generation for 100 iterations.

Table 4.3: Objective function values after 100 iterations of Column Generation.

	LP objective function value [1e6]	ILP objective function value [1e6]
Agent	1.7886	1.8055
Simple 1	1.7892	1.7971
Simple 2	1.7898	1.8079

5
Discussion

The aim of this thesis was to investigate the possibility of using a Deep Reinforcement Learning (DRL) algorithm for generating variables in Jeppesen's Trainee and Training Device Optimization (TTDO) solver. The suggested DRL approach was able to achieve slightly improved final LP objective function values in all 3 evaluation cases, as seen in Table 4.1, 4.2, 4.3. However the improvement is very minor and it is unclear whether or not this is due to limitations in the DRL algorithm, or if it is simply not possible to leverage the variable generator any more than what our results show. Furthermore, Figure 4.1 does not seem to indicate faster convergence, while Figure 4.2 clearly does show faster convergence. Recall that the results in [6] always showed faster convergence from their agent. Limitations in our DRL algorithm could for example be:

- Not enough capacity in the Graph Neural Network.
- Poor design of the Graph Neural Network.
- Insufficient training.
- Badly designed reward function.
- Not enough or poorly chosen node features for the bipartite graphs.

On the other hand, even though Jeppesen's variable generator gives us a unique set of variables each time it is called, the discrepancy between the sets of variables might be too small for our agent to be able to leverage any more than what our results show, thus also resulting in very minor improvements.

Figure 4.3, which shows the convergence plot for 100 iterations of CG, does not indicate that our agent was able to generalize to CG iterations beyond the first 20 which it was trained on. As can be observed, if we were to terminate CG after 60 iterations then Jeppesen's original TTDO solver would have resulted in a better final LP objective function value for the specific problem instance.

Another remark is that this thesis work did not have time to gather any statistics from the performance evaluation. The trained agent was just deployed a few times to check consistency, but one should preferably deploy it multiple times and track detailed statistics in order to allow for a better analysis of the results.

6
Conclusion

The research questions in this thesis work were to investigate the viability of using Deep Reinforcement Learning (DRL) for generating variables in the Trainee and Training Device Optimization (TTDO) solver, and to see if we could increase the quality of the generated variables. Results show that it is possible to teach an agent a policy that slightly improves the final LP objective function values, and results in faster convergence in at least 1 of the 3 evaluation cases. We do gain traction during training so our agent is learning something, but the overall benefits are not that great considering that the DRL approach requires an upfront computational cost in the form of training the agent. I think a DRL approach to the TTDO problem would be more viable if it was proven that the agent was able to generalize - both to CG iterations beyond the ones the agent was trained on, but also to new datasets. Unfortunately, this thesis work only tested a single dataset from one specific airline due to time constraints.

6.1 Future work

The training process of the agent was quite insufficient in this thesis work. The agent was trained using CPUs and the bipartite graphs representing the encoded state of the RMP become very large eventually (roughly 10k nodes) resulting in far too slow training times. This could be addressed by using GPUs and coding more efficiently.

Moreover, the discrepancy between the sets of variables seems be too small for our agent to be able to leverage any more than what our results show. One could try using different variable generators that gives us more discrepancy. It is also possible to try increasing the action space by letting the agent choose from more than two sets of variables.

Bibliography

[1] A. Demiriz, K. P. Bennett, J. Shawe-Taylor, "Linear Programming Boosting via Column Generation", https://link.springer.com/article/10.1023/A:1012470815092, Accessed: 2023-05-11

[2] A Menzli, "Graph Neural Network and Some of GNN Applications: Everything You Need to Know", https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications, Accessed: 2023-05-23

[3] B. Brozek, M. Furman, M. Jakubiec, B. Kucharzyk, "The black box problem revisited. Real and imaginary challenges for automated legal decision making", https://link.springer.com/article/10.1007/s10506-023-09356-9, Accessed: 2023-05-27

[4] C.Wang, C.Guo, X.Zuo, "Solving multi-depot electric vehicle scheduling problem by column generation and genetic algorithm", https://www.sciencedirect.com/science/article/pii/S1568494621006955, Accessed: 2023-05-11

[5] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, M. Grohe, "Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks", https://arxiv.org/abs/1810.02244, Accessed: 2023-05-27

[6] C.Chi, A.Aboussalah, E.Khalil, J.Wang, Z.Sherkat-Masoumi, "A Deep Reinforcement Learning Framework for Column Generation", https://www.researchgate.net/publication/361134972_A_Deep_Reinforcement_Learning_Framework_For_Column_Generation, Accessed: 2023-02-16

[7] D. E. Rumelhart, G. E Hinton, R. J Williams, "Learning internal representations by error propagation", https://ieeexplore.ieee.org/document/6302929, Accessed: 2023-05-23

[8] Deep Learning Wizard, "Markov Decision Processes (MDP) and Bellman Equations", https://www.deeplearningwizard.com/deep_learning/deep_reinforcement_learning_pytorch/bellman_mdp/, Accessed: 2023-05-26

[9] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, M. Riedmiller, "Deterministic Policy Gradient Algorithms", https://www.researchgate.net/publication/280752017_Deterministic_Policy_Gradient_Algorithms, Accessed: 2023-05-29

[10] F. S. Hillier and G. J. Lieberman, "The Origins of Operations Research", in *Introduction to Operations Research*, pp. 1–2, 2010

[11] G. L. Nemhauser, L. A. Wolsey, "The Primal and Dual Simplex Algorithms", in *Integer and Combinatorial Optimization*, pp. 30-40, 1999

[12] G. L. Nemhauser, L. A. Wolsey, "Duality", in *Integer and Combinatorial Optimization*, pp. 28-29, 1999

[13] Gurobi Optimization, "Mixed Integer Programming Basics", https://www.gurobi.com/resources/mixed-integer-programming-mip-a-primer-on-the-basics/, Accessed: 2023-04-09

[14] IBM, "algorithm for continuous linear problems", https://www.ibm.com/docs/en/icos/22.1.0?topic=parameters-algorithm-continuous-linear-problems, Accessed: 2023-05-16

[15] I Goodfellow, Y. Bengio, A Courville, "Deep Learning", https://www.deeplearningbook.org/, Accessed: 2023-05-22

[16] Jeppesen, a Boeing Company, "Commercial Aviation", https://ww2.jeppesen.com/commercial-aviation-market/, Accessed: 2023-02-16

[17] J. Desrosiers and M. E. Lübbecke, "A primer in column generation", in *Column Generation*, G. Desaulniers, J. Desrosiers, and M. M. Solomon, pp. 1–32, 2005

[18] K. G. Murty, "Duality in Linear Programming", in *Linear Programming*, pp. 182-185, 1983

[19] K. Gurney, "Real and Artificial Neurons", in *An Introduction to Neural Networks*, pp. 7-24

[20] L. A. Wolsey, "Heuristic Algorithms", in *Integer Programming*, pp. 203-216, 1998

[21] L. A. Wolsey, "Branch and Bound", in *Integer Programming*, pp. 91-106, 1998

[22] L. A. Wolsey, "Formulations", in *Integer Programming*, pp. 1-17, 1998

[23] L. A. Wolsey, "Column Generation Algorithms", in *Integer Programming*, pp. 185-201, 1998

[24] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, A. Lodi, "Exact Combinatorial Optimization with Graph Convolutional Neural Networks", https://arxiv.org/abs/1906.01629,Accessed: 2023-05-23

[25] M. Morabit, G. Desaulniers, A. Lodi, "Machine-Learning–Based Column Selection for Column Generation", https://pubsonline.informs.org/doi/abs/10.1287/trsc.2021.1045, Accessed: 2023-05-11

[26] pytorch-geometric, "Creating Message Passing Networks", https://pytorch-geometric.readthedocs.io/en/latest/tutorial/create_gnn.html, Accessed: 2023-05-23

[27] P. Velickovic, G. Cucurull, A. Casanova, ARomero, P. Lio, Y. Bengio, "Graph Attention Networks", https://arxiv.org/abs/1710.10903, Accessed: 2023-05-24

[28] R. E. Uhrig, "Introduction to Artificial Neural Networks", https://ieeexplore.ieee.org/document/483329, Accessed: 2023-05-22

[29] Transportstyrelsen, "Typ- och klassutbildning", https://www.transportstyrelsen.se/sv/luftfart/Certifikat-och-utbildning/piloter/typ--och-klassutbildning/, Accessed: 2023-04-01

[30] Technical University of Munich, "Algorithmic Scheduling in Industry", https://www.ieai.sot.tum.de/wp-content/uploads/2022/06/Research-Brief_Algorithmic-Scheduling-in-Industry_

Technical-and-Ethical-Aspects_June2022_FINAL.pdf, Accessed: 2023-05-27

[31] V.Mnih, K.Kavukcuoglu, D.Silver, A.Graves, I.Antonoglou, D.Wierstra, M.Riedmiller, "Playing Atari with Deep Reinforcement Learning", https://arxiv.org/abs/1312.5602, Accessed: 2023-04-10

[32] V.Mnih et al, "Human-level control through deep reinforcement learning", https://www.nature.com/articles/nature14236, Accessed: 2023-05-22

[33] Wolfram Mathworld, "Bipartite Graph", https://mathworld.wolfram.com/BipartiteGraph.html, Accessed: 2023-05-23

[34] Y.Bengio, A.Lodi, A.Prouvost, "Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon", https://arxiv.org/abs/1811.06128, Accessed: 2023-04-19

A

Appendix

A.1 Variable Node Features

- **Cost of rollout**
- **Reduced cost**
- **RMP variable** Binary value indicating if the variable already belongs to the current Restricted Master Problem.
- **Suggested variable** Binary value indicating if the variable is a newly generated variable, and not yet added to the Restricted Master Problem.
- **Previous solution values** The variable values from the previous solution to the Restricted Master Problem. Suggested variables have this feature set to 0.
- **Natural pair** Binary value indicating if the rollout is for a natural pair of pilots or not.
- **Connectivity** Number of constraint nodes connected to the variable. This feature represents the resource slots utilized by the rollout, along with the covered demands.
- **Training duration** Time between the start of the first training event and the end of the last training event, for the corresponding rollout.
- **Number of training events**

A.2 Constraint Node Features

- **Dual value**
- **Connectivity** Number of variable nodes that contribute to the constraint.
- **Demand constraint** Binary value.
- **Resource constraint** Also a binary value.
- **Start hour** Time of day when the corresponding training event starts. Demand constraints have this feature set to 0.

A.3 Bipartite Graph Example

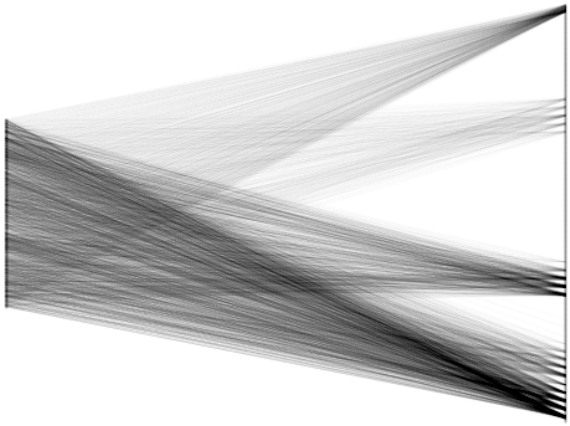

Figure A.1: Example of a Linear Program encoded as a bipartite graph. Nodes on the left are variable nodes and nodes on the right are constraint nodes. Edges exist when a variable contributes to a constraint.

A.4 Hyperparameters

Table A.1: Hyperparameters used during training of the agent.

Optimizer	Adam
Learning rate α	1e-2
Initial exploration rate ϵ	1
Final exploration rate ϵ	1e-2
Discount factor γ	0.99
Target network update rate τ	1e-3
Experience replay size	1000
Batch size	32

A.5 Training in Offline Environment

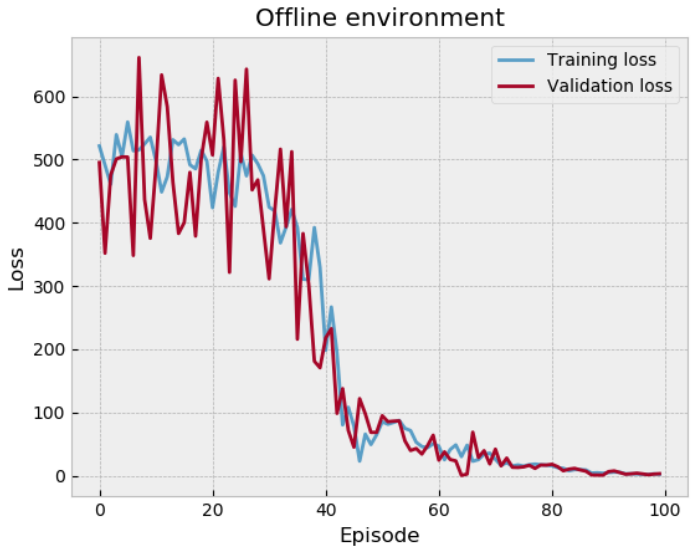

Figure A.2: Regression task performance of the Graph Neural Network designed in this thesis.

A.6 Training of the Agent

Figure A.3: Training progress of the agent while interacting with the Column Generation process of the Trainee and Training Device Optimization solver.

DEPARTMENT OF PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY