
# University of Padua

Department of Mathematics

*Master Thesis in Data Science*


# Bin Packing through Machine Learning


*Supervisor*

Professor Francesco Rinaldi

University of Padua

*Co-supervisor*

Gianluca Emireni

Senior Data Scientist, Horsa Group

*Master Candidate*

Pierpaolo D'Odorico



*Academic Year*

2021-2022

Dedicated to my family, friends, and all the people who push me to be better everyday.

# Abstract

In this thesis project we propose a wide range of Machine Learning techniques for dealing with the Bin Packing problem. The business domain is transportation optimization, a popular application field of Operational Research methods. The work is inspired by a real project by the consulting firm Horsa Group. The aim is to inspect the business problem from a mathematical point of view and to focus on different state-of-the-art techniques involving Machine Learning.

The objective is to give an overview of the different possible approaches for further developments and compare the pros and cons of possible solutions. We will also compare the performances of those techniques on generated example data and real-world data.

The final goal is to reduce the costs of the shipping process by increasing efficiency. The focus will be on how the shipping pallets are composed, packing the items with an efficient and scalable framework.

The road map consists in defining in a formal way the Operational Research problem and the business problem, to compare classical approaches with some of the methods that nowadays are more and more popular and involve Machine Learning techniques. Some of those approaches involve Deep Reinforcement Learning and Graph Neural Networks.

Finally, we will inspect a wide range of possibilities for making the bin packing process more efficient, simulating different real case scenarios. The aim is to give a clear overview of future developments in Bin Packing Optimization algorithms. Those developments can make the company's shipping software scalable and well-performing, with more efficient use of resources.

# Contents

# Listing of figures

# Listing of acronyms

**MILP** . . . . . Mixed Integer Linear Programming

**NP-hard** . . . . . Non-deterministic Polynomial-time hard

**RD** . . . . . Research and Development

**LP** . . . . . Linear Program

**OR** . . . . . Operation Research

**ML** . . . . . Machine Learning

**DL** . . . . . Deep Learning

**RL** . . . . . Reinforcement Learning

**CO** . . . . . Constrained Optimization

**GCNN** . . . . . Graph Convolutional Neural Network

**PCT** . . . . . Packing Configuration Tree

**MLP** . . . . . Multi Layer Perceptron

**GAT** . . . . . Graph Attention Layer

# 1 Introduction

The problem analyzed in this master thesis belongs to the context of Transportation Optimization. Different size items need to be stored in shipping pallets of a fixed dimension. For this reason, we consider the Bin Packing problem, which is a classical Operation Research problem suitable for this type of decision.

Bin Packing problem is an optimization problem, in which items of different sizes must be packed into a finite number of bins or containers, each of a fixed given capacity, in a way that minimizes the number of bins used.

The problem has many applications, such as filling up containers, loading trucks with weight capacity constraints, and creating file backups in media and technology mapping. There are many variations of this problem, such as 2D packing, linear packing, packing by weight, packing by cost, and a lot of other applications.

Before diving into the mathematical formalization we need to focus on some basic concepts about computational complexity theory and the NP-hardness of algorithms. Then we need to understand the basics of general Mixed Integer Linear Programming problems, which from now on we will shorten to MILP.

After the formal problem definition, we need to describe the Operational Research field from a decision-making point of view and its intersection with Machine Learning techniques. Those intersections are becoming so popular in academic research and corporate research nowadays.

## 1.1 Business Problem

Optimizing the shipping process between constraints and costs is a fundamental aspect of sustainable supply chain optimization. And it is a crucial element in preserving the competitiveness of a company. In this optimization process, one aspect is often overlooked: the management of the shipping or transportation supply chain.

The costs of the shipping activity represent a non-negligible part of the costs of the entire supply chain. To exclude this phase from the optimization process can strongly affect good margins.

The management of the shipping activity is very complex because it must take into account multiple constraints while keeping costs under control. However, in many companies, the management of shipping transportation is:

- It is based on formal and fixed rules;
- It relies on a single person experience;
- It requires a strong manual work.

Those critical issues do not allow complete optimization of the shipping process.

Starting from the list of items to be shipped, their characteristics, and the shipping constraints the goal is to optimize in an efficient and scalable way the optimal number of shipping units and their composition. And also the optimal composition of the loaded trucks in terms of saturation, combined with the optimization of the shipping routes. The result can be more shipping units with the

same number of trucks loaded or the same number of shipping units with fewer trucks loaded.

From a practical point of view, we need to split the process into two different steps, as reported in Figure 1.1. The first one takes into account a list of items with some physical characteristics and the final destination labels. The goal of the first step is to create optimized shipping units containing items all directed to a unique final destination. With this constraint, we can consider every shipping unit as a unified group of items that can be moved conveniently. The second step takes into account the shipping units and loads trucks in an optimized way, taking into account space constraints and routes.

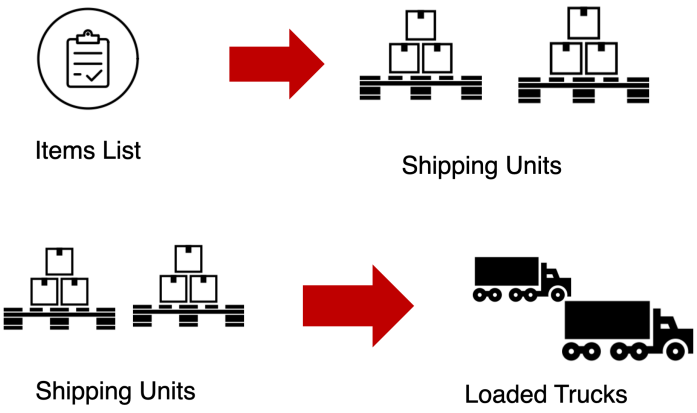


**Figure 1.1:** Description of the business problem.

Some of the key constraints of this business problem are related to practical constraints in the real world scenario. We need to take them into account for creating a custom-made optimized process. We enumerate some of the most important ones:

- The shipping unit includes all constraints related to product characteristics;
- Warehouse picking rules (if any) need also to be included;

- Includes all the dimensional constraints of the means of transport and deliver;
- Includes logistical constraints related to the different types of vehicles and shipping areas;
- Includes the optimization of routes according to the customers to be reached.

As a result of this whole optimization process, we will reduce planning and preparation time for shipment. It's also important to ensure greater security and accuracy in adhering to shipping standards.

In this thesis project the main focus is on the bin packing process in the first step of the optimization, as described in Figure 1.2. The goal is to inspect several methods for creating the shipping units in an efficient and scalable way, taking into account all the given constraints.

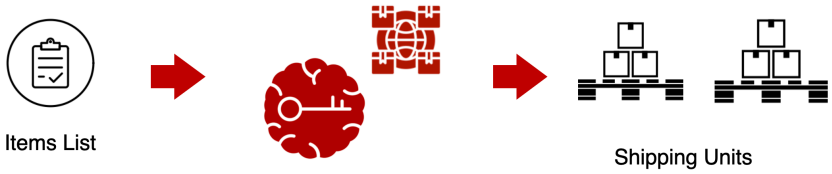


**Figure 1.2:** Business problem in this thesis.

## 1.2 Complexity Problems

The bin packing business problem has some computational complexity issues that need to be introduced. The bin packing optimization is a computational problem solvable by applying mathematical steps. Computational complexity theory classifies computational problems in classes, according to the resources used for finding the solution, that can be measured in time and storage. The goal

of computational complexity theory is to quantify the algorithms' computational complexity.

The complexity of an algorithm is often expressed using Big-O notation. It is a standard mathematical notation that shows the efficiency of an algorithm in the worst-case scenario relative to its input size. Big-O notation captures the time or space complexity upper bound to show how much an algorithm would require in the worst-case scenario as the input size grows.

$$f(n) = O(inputSize)$$

Given an input size and a c constant, we can give some examples of time complexity. Given a list of $n$ elements, we want to extract the first element of the list the time would be constant and the notation would be $O(1)$.

If we want to extract a specific item in a list of $n$ elements, we can not know the number of items to check for finding the right one. The worst-case scenario is that the item we are looking for is in the final position of the list. In this case, the notation would be $O(n)$, because the worst case is to check every element in the $n$ elements list.

The bin packing problem belongs to the class of combinatorial optimization problems, and this specific class of problems belongs to a particular complexity class called NP-hard. But for understanding what it means we need to define the context.

We call P the class of problems with polynomial time complexity $O(n^c)$, and we can see the comparison with other complexities in Figure 1.3. The class P consists of all those decision problems that can be solved on a deterministic sequential machine in an amount of time that is polynomial in the size of the input. The class NP consists of all those decision problems whose solutions can be verified in polynomial time. This means that we do not know if finding the solution requires a

polynomial time. But given a possible solution, we can verify if it is a solution in polynomial time.

NP-hard (non-deterministic polynomial-time hard) problems are the ones that can be informally defined as ”at least as hard as the hardest problems in NP problems space” as shown in Figure 1.4.

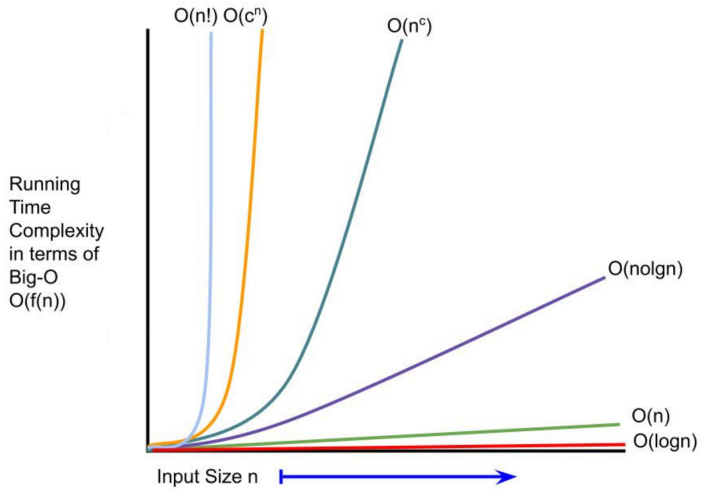


**Figure 1.3:** Big-O complexity comparison.

Bin packing problem belongs to the NP-hard class because the problem complexity grows exponentially with the problem size, since the number of possible partitions is higher than $\left(\frac{n}{2}\right)^{\frac{2}{n}}$ . This implies there is no efficient algorithm to find an optimal solution for every instance of bin packing problem.

Throughout the search for the best possible solutions for NP-hard problems, a wide variety of solution procedures have been proposed, however, there is no unique efficient algorithm capable of finding the best solution for all possible instances of a problem.

Bin packing problems has a significant number of applications in the industry, so it has been widely studied, and multiple algorithms have been developed to

solve it. The solutions range varies from approximate and exact algorithms to metaheuristics, but nowadays there is so much more work to be done.

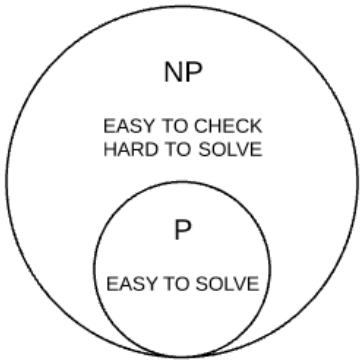


**Figure 1.4:** P vs NP class diagram.

## 1.3 Machine Learning contribution

How is it possible to solve NP-hard problems in practical time? Every classic NP-hard problem has a rich set of techniques researchers have developed for more efficient solving. The techniques also depend on the structure of the problem.

In addition to the problem structure, an expert will know how to refine algorithm parameters to different behaviors of the optimization process in the specific application domain. This extends the knowledge with unwritten intuition.

The focus of this thesis work is on combinatorial optimization algorithms that automatically perform learning on a chosen distribution of problems, with a deeper dive into learning bin packing problems. Incorporating machine learning components [1] in the algorithm can help in making processes more efficient and scalable in a real-world scenario.

Machine learning in an optimization context focuses on performing a task based on learning some finite data. Without the need for an explicit mathemat-

ical formulation. This is useful especially when the true data distribution is not known analytically.

There are two main reasons for inspecting machine learning techniques on combinatorial optimization problems. In the first place, the researcher assumes expert knowledge about the optimization algorithm and replaces some heavy exact computations with a fast approximation. Learning can be used to build such approximations without the need to derive new explicit algorithms, but to use data to train a model in a supervised learning way. The second reason is that a single expert knowledge may not be sufficient because he can make only a few selected decisions related to personal experience. With machine learning, we can expand the space of these decisions and learn the best-performing policy.

Using machine learning to tackle the bin packing problem, we need to decompose the problem into smaller and simpler learning tasks. For this reason, even if the result is an approximation of the exact solution, we would have a lot of time and scalability advantages.

## 1.4 Thesis Objective

In this thesis we have a set of objectives to achieve, that range from academic research topics to business application scenarios. The goal is to give a complete overview of the different techniques for solving MILP problems and in particular Bin Packing problem. This is important because the academic research about those topics runs on a parallel track if compared to Research and Development (R&D) in tech companies. The majority of enterprise solutions are not open source because they are part of proprietary software. For this reason, it is hard to inspect which are the state-of-the-art solutions and how fast the technology is growing in the operational research using the machine learning techniques.

Operational research (which from now on we will shorten to OR) and machine learning (ML) can be seen as completely different fields. OR is a field of

problem-solving and decision-making that is useful in the management of organizations. The objective is to break down problems into basic components and then solve them in defined steps by mathematical analysis. And is more focused on solving a specific and not generalized problem. The basic components that drive the solution are related to a specific task and custom created to solve it. ML is also tightly related to optimization because most ML problems are formulated as the minimization of a certain loss function. During the training of the model, an optimization algorithm minimizes the loss on the training set.

But there is an important difference: the ultimate goal of ML is to minimize the loss on unseen data. And not optimizing a specific task. We can simplify by saying that ML is an optimization problem with the goal of generalization. The objective of this work is to find a balance for exploiting the strengths of both OR and ML fields, and to create a unique solution for optimizing the bin packing problem.

This thesis research wants to give a clear overview of the open-source materials and applications that are available nowadays for dealing with MILP optimization using machine learning techniques. Before starting this research work the sources of information and materials about this topic were not unified in a complete work, but they were scattered across papers and works about specific applications with a not clear context. This work will be a complete survey on the state of the art academic techniques and can be useful for future improvements and business experiments. This thesis can be seen in the first place as a starting point for a better understanding of the topic.

The objective from a business point of view is to define the business problem and the increase of performances using joint OR and ML methods, comparing the performances given by the classical OR exact solution frameworks. For making this possible we need in the first place to formalize the problem from a mathematical point of view. Then we need to use sample data for testing different

solution performances.

The final experiments have not the objective of creating a full pipeline for solving the business problem from end to end, but they can compare different possible approaches instead. The final objective is to measure the performances of the different bin packing techniques based on three metrics:

- Time required for solution;
- Mean percentage of the bins filled;
- Number of bins used for packing all the items required.

The objective is to find more scalable options that can be suitable for future improvements of the Horsa Group algorithmic framework.

# 2
# Background

In this section we will introduce the background of the thesis project, defining from a formal point of view all the key points that define the basis for the complete work. The roadmap is to define in the first place the business problem, and then operational research, machine learning, Markov Chain concepts, and the bin packing formalization.

We start by describing business problems and the real-world data. Then we will focus on some specific features and solutions for trying methods for optimizing the shipping process. The data consists of a table where single item orders are contained in rows, and the features, for each order, are:

- Client code;
- Final client destination;
- Item;
- Item weight;
- Item volume;
- Item x,y,h dimensions.

We have also a pallet dimension, that can be chosen by the user. We consider the pallet as a bin. A pallet can contain only items that will be delivered to the same final client destination, for making the routing easier. Most of the clients are just intermediaries that will deliver again the items to the final clients, and they prefer to not have a pallet that contains boxes for multiple clients inside.

We take into account some items and bins. We want to pack items into bins in an efficient way. Then we will take the loaded pallets to store them in trucks, but this is the next step. The interesting thing is that we can consider both pallets and trucks as bins, in this way the pallet optimization part can be seen as a base version of the truck optimization algorithm. Using this trick we can consider the truck optimization as a generalization of the pallet optimization problem that considers spatial distances in delivery. But this part will not be covered in this master thesis work.

## 2.1 Operational Research definition

From now on we will use the terms Operations Research and Operational Research to refer to OR. Operations research is concerned with finding optimal solutions to decision-making problems. In the late nineteenth century, researchers began to explore the application of mathematical and scientific analysis to the production of goods and services [2]. The field was accelerated during the industrial revolution when companies began to subdivide their management into departments responsible for distinct aspects of overall decisions. The main idea is that, if management, organization, planning, or decision-making is a logical process, it can be expressed in terms of mathematical symbols and relationships. This desire to be able to better model and understand business decisions sparked the development of several concepts used today, such as linear programming, dynamic programming, and queuing theory.

### 2.1.1 Optimization Problems

An optimization problem is a problem of finding the best solution from all feasible solutions. It's a popular problem in mathematics, computer science, and economics. Optimization problems can be divided into two different categories.

- Continuous optimization consists in finding the minimum or maximum value of a function of one or many real variables, subject to constraints. The constraints usually take the form of equations or inequalities. An example could be:

  $\min_{x \in \mathbb{R}^n} c^T x$ , subject to $Ax = b$ and $x \geq 0$

- Discrete optimization consists in finding the minimum or maximum discrete value of a function of one or many variables, subject to constraints. An example could be a optimization problem with integer value condition:

  $\min_{x \in \mathbb{R}^n} c^T x$ , subject to $Ax = b$, $x \geq 0$ and $x \in \mathbb{Z}^n$

Constraint optimization is the name given to identifying feasible solutions out of a set of candidates, where the problem can be modeled in terms of arbitrary constraints. It's also known as constraint programming.

The subtle difference is that constraint programming is based on constraints and variables, which means finding feasible solutions. Optimization instead focuses on the objective function, of finding an optimal solution among all the feasible ones.

Constraint programming problems may not even have an objective function, and the goal may simply be to find a very large set of possible solutions, to find a more manageable subset by adding constraints to the problem. The combination of constraint programming and optimization has been successfully applied in planning, scheduling, and numerous other domains with heterogeneous constraints.

### 2.1.2 Optimization Problems formalizations

We can simply define a linear programming optimization problem with the following configuration: decision variables, function to minimize or maximize, and constraints related to variables.

An example is finding integer values for the variables $x$, $y$, $z$ and $k$ with $k > 2$ such that $xk + yk - zk$ is minimal.

From a formal point of view we can write:

**Maximize** $c_1x_1 + \cdots + c_nx_n$

**Subject to**

$$a_{1,1}x_1 + \cdots + a_{1,n}x_n \leq b_1$$
$$\vdots$$
$$a_{m,1}x_1 + \cdots + a_{m,n}x_n \leq b_m$$
$$x_1 \geq 0, \ldots, x_n \geq 0$$

Where $m, n \in N$, $c_j, b_i$ and $a_i, j$ are constants, $x_j$ are the decision variables and $i = 1, ..., m\,; j = 1, ..., n$.

Such problems are called linear optimization problems or linear programs (LP) since all functions involved are linear. Special interest in this problem class stems from the fact that the linear programming paradigm can serve as the formal model for several economic resource allocation problems. And for this reason, they are so well studied in an industry context.

## 2.2 Machine Learning key ideas

Nowadays, for dealing with complex integer linear programming problems researchers are looking for approximate Machine Learning solutions to the problem.

We need to define the basics of the ML approach[1], which consists in learning to operate on data, without defining a specific function for mapping inputs to outputs. This operation can be done by learning the policies on a subset of data, called train set, and then testing the process on unseen data, called test data.

In this way, we can quantify the generalization capacity using some defined metrics calculated on the unseen set of data. We can divide ML into three subclasses: supervised learning, unsupervised learning, and reinforcement learning. In this thesis, we will focus on two of them, supervised learning and reinforcement learning (RL).

### 2.2.1 Supervised Learning

In supervised learning setting a set of input, features-target pairs are provided and the task is to find a function that for every input has a predicted output as close as possible to the provided target. Finding such a function is called learning and is solved through an optimization problem over a family of functions. The loss function measures the discrepancy between the output and the target. The loss function can be chosen depending on the task (regression, classification, etc.) and on the optimization methods.

From a mathematical point of view let $X$ and $Y$, following a joint probability distribution $P$, be random variables representing the input features and the target. Let $l$ be the per sample loss function to minimize, and let $\{f_\theta | \theta \in R_p\}$ be the family of machine learning models, parametric in this case, to optimize over.

The supervised learning problem is framed as:

$$\min_{\theta \in \mathbb{R}^p} \mathbb{E}_{X,Y \sim P} \ell \left(Y, f_\theta(X)\right) \tag{2.1}$$

For instance, $f_\theta$ could be a linear model with weights $\theta$ that we wish to learn, and loss function l is task-dependent. The probability distribution is unknown and inaccessible, for this reason, it is approximated by the empirical probability

distribution over a finite dataset $D_{train} = (x_i, y_i)_i$ solving the following optimization problem:

$$\min_{\theta \in \mathbb{R}^p} \sum_{(x,y) \in D_{\text{train}}} \frac{1}{|D_{\text{train}}|} \ell\left(y, f_\theta(x)\right) \tag{2.2}$$

Because the true probability distribution remains unknown, we estimate the generalization error by evaluating the trained model on a separate test dataset $D_{test}$ with the following formula:

$$\sum_{(x,y) \in D_{\text{test}}} \frac{1}{|D_{\text{test}}|} \ell\left(y, f_\theta(x)\right) \tag{2.3}$$

Selecting the best among various trained models cannot be done on the test set. Selection is a form of optimization, and doing so on the test set would bias the estimator in minimizing the loss function. This is a common mistake to be avoided. To perform model selection, a validation dataset $D_{valid}$ is used to estimate the generalization error of different ML models is necessary.

Model selection can be done based on the estimates, and the final unbiased generalization error of the selected model can be computed on the test set. The validation set is therefore often used to select the effective capacity of the model.

For example, we can increase performances by changing the amount of training, the number of parameters $\theta$, and the amount of regularization imposed on the model. Regularization refers to the techniques that are used in ML models to minimize the loss function and prevent overfitting (learning well but not generalizing) or underfitting (generalizing well but with low learning).

### 2.2.2 Markov Decision Process

Before introducing RL we need to define Markov Decision Processes (Fig. 2.1). A Markov Decision Process (MDP) is described by a tuple $(S, A, P, r)$ where:

- $S$: a set of states related to environment and agent
- $A$: a set of actions the agent can take
- $P$: a state-transition probability which is the probability of moving from state $s_t \in S$ to state $s_{t+1} \in S$ under the action $a \in A(s_t)$, where $A(s)$ is the set of possible actions to be taken in state s
- $r$: the reward function corresponding to a certain action feedback. This function maps the tuple of consequential states $(s_t; s_{t+1})$ to a correspondent reward as a feedback for agent $a$ for the state transition.

More formally we can describe the environment as a transition probability for state-transition (2.4) and for transition-reward (2.5).

$$P_a\left(s_{t+1}, s_t\right) = P\left(s_{t+1} \mid s_t, a\right) \tag{2.4}$$

$$P\left(s_{t+1}, r_{t+1} \mid s_t, a_t\right) \tag{2.5}$$

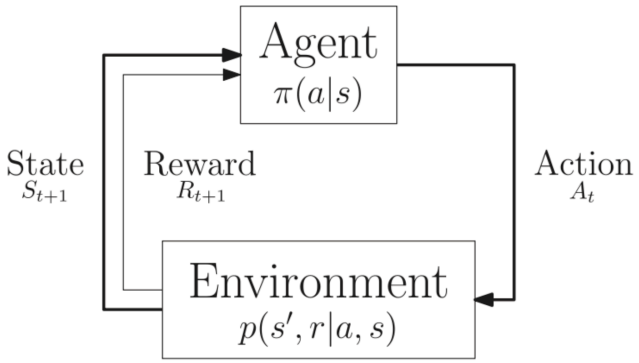


**Figure 2.1:** Markov Decision Process scheme: the agent takes action $A_t$ on state $S_t$ and receives a feedback. The following state is $S_{t+1}$ and a reward $r$ is given. And this process can be iterated $t$ times.

With MDP we are able to model the steps needed to reach the final state, and to compose an episode in which the pairs action-states compose a trajectory in

a dynamic environment. This learning from experience through trial and error, maximizing a reward function, is the key idea of Reinforcement Learning.

### 2.2.3 Reinforcement Learning

The goal of RL is to train the agent to maximize the expected sum of future rewards. The dynamics of the environment, based on MDP, need not be known by the agent and are learned through exploration vs exploitation dilemma. This means choosing between exploring new states for refining the knowledge searching for possible long-term improvements, or exploiting the best-known scenario learned so far.

For finding the optimal policy we first need to define what a policy is. A policy $\pi$ is a density function (2.6) mapping for each state the probabilities of taking different actions $a$. A policy can be stochastic as shown in this Equation (2.6), or it can be deterministic with a binary output (probability 1 just for one of the possible actions).

$$\pi : S \times A \to [0, 1], \pi(s, a) = P(a \mid s) \tag{2.6}$$

A policy is optimal when it maximizes the expected cumulative reward. The cumulative reward is called return and it is defined with a weighted sum of rewards over states (2.7).

$$G_t = \sum_{i=0}^{\infty} \lambda^i R_{t+1+i} \tag{2.7}$$

The return is weighted by $\lambda \in [0, 1]$, the so called discount rate. The presence of discounting is essential for two reasons. The first it is because with $\lambda$ the sequence $R_i$ is bounded, in order to get the sum in Equation (2.7) convergent. The second is the intuitive meaning of penalizing future rewards for the benefit of present reward. We need to find a $\lambda$ value representing the trade-off between

assuring convergence ($\lambda < 1$) and caring about future rewards ($\lambda > 0$).

### 2.2.4 Reinforcement Learning for CO

Reinforcement learning algorithms depend on the functions that take as input the states of MDP and outputs the actions' values or actions. States represent some information about the problem. For this reason, encoding problem states into numbers, they can be suitable for solving CO problems including techniques such as recurrent neural networks, graph neural networks, attention-based networks, and multi-layer perceptrons [3]. We can show (Fig. 2.2) an overview of a possible pipeline for solving CO problem with RL.

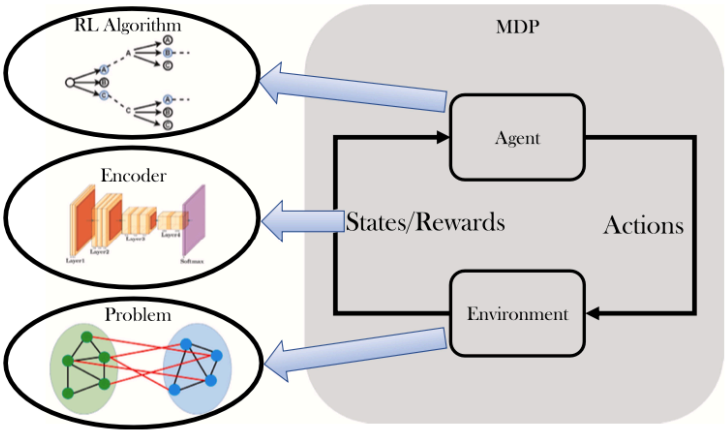


**Figure 2.2:** Using RL and formulating MDP for solving CO overview example. States are encoded with a neural network model. The agent is driven by an RL algorithm (a Tree Search in this example) and makes decisions that move the environment to the next state.

A CO problem is first formulated in terms of MDP defining the states, the actions, and the rewards. Then is defined an encoder of the states represented by a parametric function that encodes the input states and outputs a numerical vector. The vector contains Q-values or probabilities of each action.

After the formulation of the problem we can display the key steps for solving the CO problem through RL:

1. Run RL algorithm that determines how the agent learns the parameters of the encoder and makes the decisions for a given MDP.
2. After the agent has selected an action, the environment moves to a new state and the agent receives a reward for the action it has made.
3. Once the parameters of the model have been trained, the agent is capable of searching the solutions for unseen instances of the problem.

## 2.3 Bin packing formalization

In this chapter we start describing in a formal way the problem of storing items into pallets. So we need to describe variables, constraints and the function to minimize. For making the formalization suitable for generalization, we will call *items* the objects to store, and bins the pallets.

**Problem formalization:**

$i$ = item, $I$ = total sum of items
$j$ = bin, $J$ = total sum of bins
$W$ = max weight of bin, $w_i$ = weight of item $i$
$V$ = max volume of bin, $v_i$ = volume of item $i$

**Variables:**

$$x[i,j] = \begin{cases} 1 & : \text{ item i is packed in bin j} \\ 0 & : \text{ otherwise} \end{cases} \tag{2.8}$$

$$y[j] = \begin{cases} 1 & : \text{ bin j contains at least 1 item} \\ 0 & : \text{ otherwise} \end{cases} \tag{2.9}$$

**Constraints:**
Every item is stored in exactly one bin:

$$\sum_{j=0}^{J-1} x_{ij} = 1 \text{ for } i = 0, \ldots, I-1 \tag{2.10}$$

Total weight in each bin can't exceed bin max weight capacity, only if bin is used:

$$\sum_{i=0}^{I-1} w_i x_{ij} \leq W y_j \text{ for } j = 0, \ldots, J-1 \tag{2.11}$$

Total volume in each bin can't exceed bin max volume capacity, only if bin is used:

$$\sum_{i=0}^{I-1} v_i x_{ij} \leq V y_j \text{ for } j = 0, \ldots, J-1 \tag{2.12}$$

**Objective function:**
Total number of bins used:

$$\min \sum_{j=0}^{J-1} y_j \tag{2.13}$$

In the following chapters we will describe different techniques for dealing with Bin Packing problem, and MILP problems in general, using Machine Learning tools. The goal is to inspect open source state of the art techniques searching for possible scalable solutions to the business problem. We will also compare pros and cons from a theoretical point of view based on industry needs.

# 3

# Approaches for solving bin packing

In this chapter, we want to highlight how bin packing problem, and MILP problems in general, are solved nowadays in industry and academy. Comparing the classical approach for finding the exact solution to the minimization problem, with ML-based ones. The chapter will focus on the theoretical side of approaches. The final goal is to give an overview on possible paths for solving bin packing problem, from simpler algorithms to more complicated ones.

## 3.1 Classic approaches for solving BPP

Solving bin packing problem seems easy thinking about a small number of items and bins. In this section we will focus on some classical algorithms for solving bin packing problem, without using ML. So we will start form basics and then we will describe branch and bound algorithm, that is a general technique for solving combinatorial optimization problems widely used for bin packing problems.

### 3.1.1 Classic algorithms

Before starting with classic algorithms for bin packing we define the **lower bound** concept, that is simply the minimum number of bins required for storing all the items. We can define it as the sum of weights of all items divided by the maximum weight that a bin can hold, rounded up to a whole number.

$$L = \left\lceil \sum_{i=1}^{n} w_i / W \right\rceil \tag{3.1}$$

The lower bound represents the best packing possible, if all other constraints were respected (3.1). It can be also seen as a benchmark for evaluating packing performances of different algorithms. From now on we will inspect different packing algorithms.

**First-fit** is a simple algorithm for bin packing, that can be used also in a context where items are available one at a time. That is a common situation and it's called online bin packing. First-fit bin packing takes as input a list of items of different sizes. Its output is a packing a partition of the items into bins of fixed capacity. Ideally, we would like to use as few bins as possible, but minimizing the number of bins would became an NP-hard problem. The first-fit algorithm idea is the following:

**Algorithm 3.1** First-fit pseudocode

```
Take a bin, which is initially empty
Take first item, find the first bin into which the item can fit
if such a bin is found:
    The item is placed inside it
else
    New bin is opened and the item is placed inside it
end if All items are inside bins
```

The **First-fit Decrease** works in a similar way if compared to the previous algorithm. The main difference is that we need all the items list before starting with the algorithm. The idea is to order items from largest to smallest, and start packing the largest item first. With this procedure is easier to saturate bins, because the smaller ones are placed in small spaces when bins are almost completed. For this reason the empty spaces in final bin configuration are smaller. It's important to recall that we are not yet minimizing the number of bins.

**Algorithm 3.2** First-fit Decrease pseudocode

```
Take a bin, which is initially empty
Order the items from largest to smallest
Take largest item, find the first bin into which the item can fit
if such a bin is found:
   The item is placed inside it
else
   New bin is opened and the item is placed inside it
end if All items are inside bins
```

The **Full Bin Packing** is a different technique to produce an optimal solution, using the least possible number of bins. It works by matching object trying to reach bin dimensions. With the goal to fill as many bins as possible after the matching is completed. We will inspect a powerful Full Bin Packing technique in detail in Chapter 3.1.2, called Bin Completion Algorithm [4] by Richard E. Korf.

### 3.1.2 Bin Completion algorithm

Richard E. Korf presented a new algorithm for optimal bin packing, called Bin Completion. Rather than considering the different bins that each number can be placed into, he considered the different ways in which each bin can be packed. The most famous existing algorithm for full bin packing is due to Martello and Toth [5], presented in 1990. Bin Completion, published in 2002, searches a dif-

ferent problem space instead. Rather than considering each element in turn, and deciding which bin to place it in, we consider each bin in turn, and consider the feasible sets of elements that could be used to complete that bin. In this way the algorithm runs faster and has a simpler structure.

We sort the elements in decreasing order of size, and consider the bins containing each element in turn, enumerating all the undominated completions of that bin, and branching if there are more than one. In other words, we first complete the bin containing the largest element, then complete the bin containing the second largest element, etc.

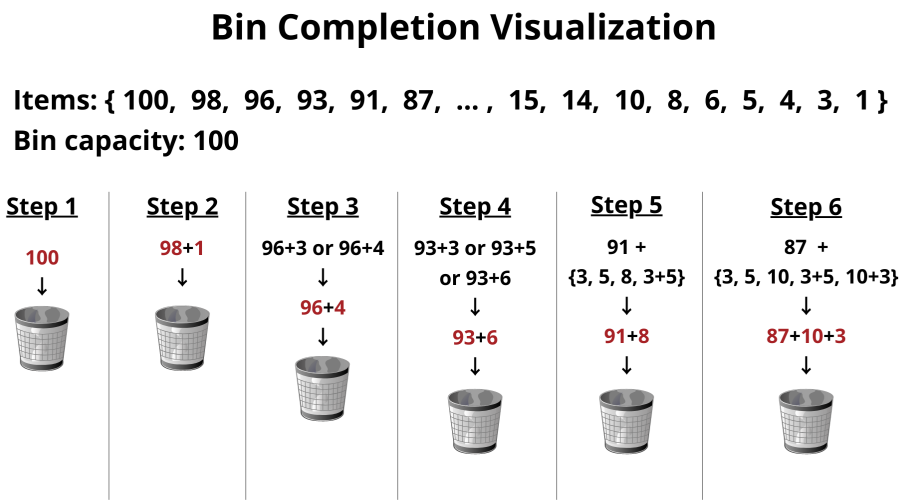


**Figure 3.1:** Bin Completion visualization.

We need also to estimate the wasted space for being able to verify optimality of the solution. And we need to calculate it with a slightly better approach than the obvious one in Equation 3.1. This estimated wasted-space calculation proceeds considering the elements in decreasing order of size. Given an element $x$, the residual capacity $r$ of the bin containing $x$ is $r = c - x$ where $c$ is the bin capacity. Then we consider the sum $s$ of all elements less than or equal to $r$, which have not already been assigned to a previous bin.

There are three possible cases:

- The first is that $r = s$: no waste and no carry over to the next bin;
- If $s < r$, then $rs$ is added to the estimated waste, a no carry over to the next bin;
- If $r < s$, then there is no waste added, and $s - r$ is carried over to the next bin.

Once the estimated waste is computed, it is added to the sum of the elements, which is divided by the bin capacity, and then rounded up.

In the first place we compute the best-fit decreasing BFD solution. Next, we compute a lower bound on the entire problem using the wasted-space bound described above. If the lower bound equals the number of bins in the BFD solution, it is returned as the optimal solution.

Otherwise we initialize the best solution so far to the BFD solution, and start a branch-and-bound search for strictly better solutions. Branch and Bound procedure is described in Chapter 3.1.3. Once a partial solution uses as many bins as the best complete solution found so far, we prune that branch of the search, making the search more efficient.

Now we have an overview of the possible bin packing solutions. We summarize pros and cons of different approaches:

- First-fit is quick and easy to perform but does not usually lead to an optimal solution;
- First-fit decrease is gives usually better solution than First-fit, but the problem remains that it not always get an optimal solution;
- Full Bin Packing usually gets a better solution compared to previous ones, but it can be difficult to perform, if numbers awkward.

**Algorithm 3.3** Bin Completion pseudocode

```
Consider the elements in decreasing order of size
while One or more items are outside of bins
  Take largest item
  Generate the undominated completions of the bin containing the chosen
  item
  if There are no completions or only one completion:
    Complete the bin
  else
    There is more than one undominated completion
    Order them in decreasing order of total sum
    Consider largest, with less elements in case of a tie
  end if
end while All items are inside bins
```

For solving the problems described in previous algorithms nowadays more powerful and general ways for solving BPP, and MILP problems in general, are developed. We will now inspect some classic algorithms for MILP problems such as Branch and Bound and Branch and Cut.

### 3.1.3 Branch and Bound algorithm

Branch and Bound algorithm [6] is used to find the optimal solution for given an NP-Hard optimization problems. B&B algorithm explores the entire search space of possible solutions and provides an optimal solution. Branch and bound algorithm consist of step wise enumeration of possible candidate solutions by exploring the entire search space, building a rooted decision tree. The root node represents the entire search space, and each child node is a partial solution and part of the solution set. Consider a general combinatorial optimization problem like the following:

$$z = \max\{f(x) : x \in S\}$$

B&B is a divide and conquer strategy, which decomposes the problem into sub-problems over a tree structure, which is referred to as branch-and-bound tree [7]. The decomposition works based on a simple idea: If $S$ is decomposed into $S^1$and $S^2$ such that $S = S^1 \cup S^2$, and we define sub-problems:

$$z^k = \max \left\{ f(x) : x \in S^k \right\}, \quad \text{for} \quad k = 1, 2$$

$$\text{then} \quad z = \max_k z^k.$$

Each sub-problem represents a node on the tree. The process of dividing a node sub-problem into smaller sub-problems is called branching and sub-problems $S^1$ and $S^2$ are called branches created at node $S$. Sub-problem $S^1$ is further branched into smaller sub-problems and so on. In this way, we can find the best and optimal solution fast. It is crucial to define an upper and lower bound for bins used. We can find an upper bound by using any local optimization method or by picking any point in the search space. We can easily obtain a lower bound for bin packing as described in Eq 3.1.

The general rule is that we want to partition the solution set into smaller subsets of solution, as shown in Figure 3.2. Then we construct a rooted decision tree and finally we choose the best possible subset. The subset is represented by a node and at each level we want to to find the best possible solution set.

There are two important phases of B&B algorithm: the first is the search phase, in which the algorithm has not yet found an optimal solution $x^*$. The second is the verification phase, in which the incumbent solution is optimal, but there are still unexplored sub-problems in the tree that cannot be pruned. Note that the incumbent solution cannot be proven optimal until no unexplored sub-problems remain. It's important to note that the delineation between the search phase and the verification phase is not known until the algorithm terminates. In

**Algorithm 3.4** Branch-and-Bound(X,f) pseudocode

Set $L = X$ and initialize $\hat{x}$
**while** $L \neq \varnothing$:
  Select a sub-problem $S$ from $L$ to explore
  **if** A solution $\hat{x}' \in \{x \in S \mid f(x) < f(\hat{x})\}$ can be found:
    Set $\hat{x} = \hat{x}'$
  **end if**
  **if** $S$ cannot be pruned:
    Partition $S$ into $S_1, S_2, ..., S_r$
    Insert $S_1, S_2, ..., S_r$ into $L$
  **end if**
  Remove S from L
**end while**
Return $\hat{x}$

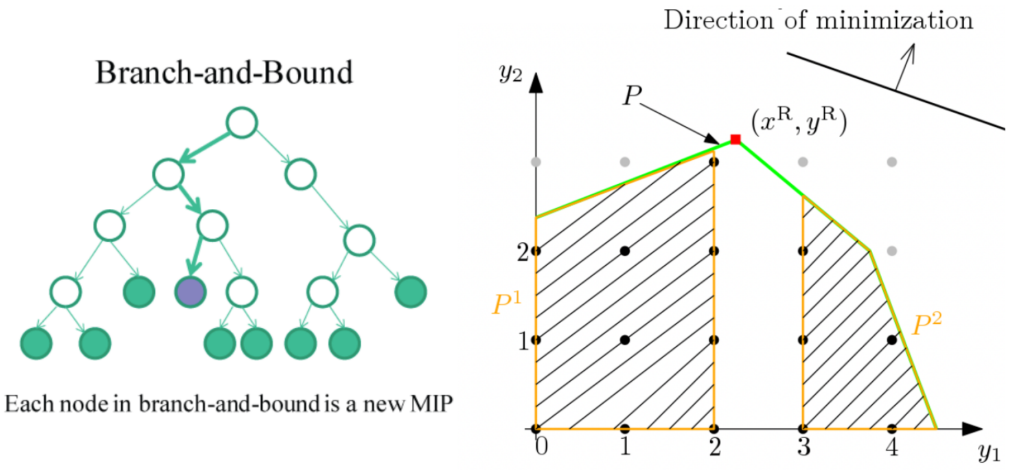


**Figure 3.2:** Branch and Bound visualization.

a slight abuse of terminology, a problem $P$ is said to be solved if the B&B algorithm has completed the verification phase. In this case, the algorithm is said to have produced a certificate of optimality.

Branch and bound is a very useful technique for searching a solution but the problem is that in worst case, we need to fully calculate the entire tree. The problem, as we said in previous chapters, are typically exponential in terms of time complexity and this is a problem especially in the worst case scenario. At best, we only need to fully calculate one path through the tree and prune the rest of it. For this reason in Chapter 4.1 we will inspect methods for using ML to choose the best branching based on past data.

### 3.1.4 Branch and Cut algorithm

The Branch and Cut algorithm is an optimization algorithm used to optimize integer linear programming. It combines two optimization algorithms: Branch and Bound and cutting planes. With this intuition we are able to utilize the results from each method in order to create the most optimal solution. In order to understand the idea behind this method we need to explain what cutting planes method is.

The idea behind the cutting plane method is to solve a sequence of linear relaxations that approximate better and better the convex hull of the feasible region around the optimal solution. The cutting plane method is commonly used for solving MILP problems and finding integer solutions. Suppose that we want to solve the integer linear programming problem $P_I$:

$$\max c^T x \quad (P_I)$$
$$x \in X$$

$$X = \{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0, x_i \in \mathbb{Z} \text{ for every } i \in I\}$$

We start with the linear relaxation $\max \left\{c^T x \mid Ax \leq b, x \geq 0\right\}$, and we solve it letting $x^*$ be a basic optimal solution. If $x^* \in X$, then $x^*$ is an optimal solution for $(P_I)$. Otherwise find an inequality that is valid for $X$ and cuts off the $x^*$

solution and add this inequality to the current linear relaxation. The process can be repeated with the new linear relaxation. This cutting process is repeated until the optimal solution found is also an integer solution.

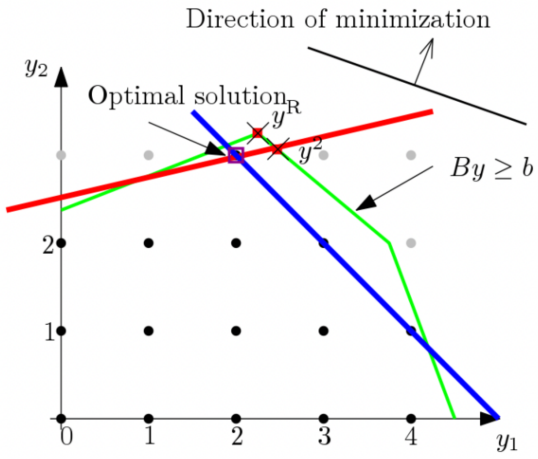


**Figure 3.3:** Branch and Cut visualization.

The Branch and Cut procedure (Fig. 3.3) consists of performing branches and applying cuts at the nodes of the tree. First, the tree is initialized to contain the root node as the only active node that represents the entire problem, ignoring all of the explicit integrality requirements. The interest is to keep the problem size reasonable, so not all cuts are applied to the model immediately. If possible, an incumbent solution is established at this point for later use in the algorithm.

When processing a node, the algorithms starts by solving the continuous relaxation of its subproblem, it means the subproblem without integrality constraints. If the solution violates any cuts, those are added to the node problem. This procedure is iterated until no more violated cuts are detected by the algorithm. If at any point in the addition of cuts the node becomes infeasible, the node is removed from the tree. Otherwise, we check if the solution of the node-problem satisfies the integrality constraints. If the objective value is better than that of the current incumbent, the solution of the node-problem is used as the new incum-

bent. The branch, when it occurs, is performed on a variable where the value of the present solution violates its integrality requirement. This practice results in two new nodes being added to the tree for later processing.

In commercial sovers it's possible to set to terminate the Branch and Cut procedure sooner than a completed proof of optimality. For example, a user can set a time limit or a limit on the number of nodes to be processed. This time limit is convinient because, in some cases, investing a lot of resources for a slightly better result is not worth it.

### 3.1.5 Or-Tools suite

SCIP solver is currently one of the fastest non-commercial solvers for mixed integer programming (MIP) and mixed integer nonlinear programming (MINLP). It is also a framework for constraint integer programming and branching algorithms. It allows for total control of the solution process and the access of detailed information about the solver. SCIP is implemented as C callable library and provides C++ wrapper classes for user plugins. For this reason it's widely used by companies as a solver for building fast and scalable optimization packages.

OR-Tools, developed by Google AI, is the leading open source software suite for optimization, tuned for dealing with problems in vehicle routing, flows, integer and linear programming, and constraint programming. With Or-Tools an optimization problem properly defined by mathematical equations, can be solved using different programming languages. We will use this solver as a baseline for dealing with optimization problems in Chapter 5.2, and comparing results with techniques involving ML approaches.

There are also different non open source solutions made by commercial competitors, for example Gurobi or CPLEX by IBM. But we will not focus on those solutions because of monetary barriers for using those solvers. And we will focus mainly on open source solutions.

## 3.2 ML approach current state

It is common to repeatedly solve similar NP-hard Combinatorial Optimization problems in real-world scenarios. Classical mathematical solvers processes each new problem independently and retains no memory of the past. We know that there exist strong statistical similarities between each of those sequentially solved problems, which could potentially be exploited to solve future problems more efficiently. This is the idea behind solving MILP problems using machine learning. Currently there are two growing lines of machine learning research on this field:

- ML approaches, where CO solvers are entirely replaced by an ML model trained to produce approximated solutions;
- Joint approaches, where hand designed decision criteria are replaced by machine learning models trained to optimize a particular metric of the solver. This second approach is attractive because of the exact solving.

On the other hand there are also several technical obstacles in studying and applying those approaches to MILP problems. Those problems are the following:

1. *Reproducibility*, that is currently a major issue. The benchmarks are different in majority of different research papers. A solution can be to adopt standard feature sets, problem benchmarks and evaluation metrics for being able to compare different approaches to the same problems.
2. *Barrier to entry to the field*. This happens because the modern exact solvers are complex and not specifically designed for customization through ML. So implementing a new research idea requires low-level solver code, that is hard even for OR experts. And, on the other hand, abstracting away a proper MDP formulation using is no trivial task for OR experts, because it requires a clear understanding of statistical learning concepts.
3. *Experts in both OR and ML are rare*. ML experts typically employ very simplified CO solvers, while OR experts typically employ basic ML models and algorithms missing potential improvements.

The solutions that we will inspect apply approaches used for solving general MILP problems, that are useful for dealing with the classical bin packing problem. The major issue is that the majority of those solutions are based on very complicated Deep Reinforcement Learning policies, and are custom made for big players in industry and big tech companies. The result is that the majority for real world solutions for those reason are not open source.

In addition to this issue, research papers are focused on benchmarks and metrics calculated on randomly generated or custom created datasets. And we can not find real world examples of this type of applications in literature. This happens because research papers methods need a custom made dataset with a specific structure, depending on the different approach chosen for learning to solve MILP problems.

In next subsections we will explore different ideas from a theoretical point of view, and in chapter 4 we'll inspect deeply the theoretical details of implementations.

### 3.2.1 Imitation Learning Approach

The idea behind the imitation learning approach is to find statistical rules and patterns during optimization process and train a model for learning the rules and speed up the process. The state of the art methods involve Graph Convolutional Neural Networks for learning B&B variable selection policies.

The whole work is described in a paper by Gasse, Ferroni and Lodi called Exact Combinatorial Optimization with Graph Convolutional Neural Networks [8]. The practical pros of this technique are the following:

- Learn directly from an expert branching rule;
- Small B&B trees, it means efficiency;
- Generalize on set of instances bigger than train set;

- Fast training compared to Reinforcement Learning techniques.

We will explain later the why those pros are possible. We will look the so called "Learning to Branch" in detail in Chapter 4.1. The main idea remains to learn which B&B splits are the best ones for reaching the optimal solution quickly.

Now we summarize the cons of this technique:

- We need a new training for every MILP problem;
- Hard to deal learn from unstructured data;
- We need training samples for learning.

Despite of those cons, for industry applications Learning to Branch with imitation learning is the best industry approach for optimizing MILP problems in a scalable way. Training on instances it's not a problem because we have historical data and the packing procedure for the client maintain the same constraints and rules in time. And in industry it's easy to collect data for repeated tasks such as daily bin packing into pallets.

From a practical point of view, running strong branching at every node is prohibitive, and modern B&B solvers instead rely on hybrid branching which computes strong branching scores only at the beginning of the solving process and gradually switches to simpler heuristics. But learning how to branch from historical data is and approach that can improve a lot the branching strategy creating a whole new set of possibilities.

### 3.2.2 Reinforcement Learning Approach

Reinforcement Learning approach is based on the idea of automating the search of the optimization heuristics by training an agent in a supervised or self-supervised

manner. In order to apply RL to CO, we need to think at the problem as a sequential decision-making process, where the agent interacts with the environment by performing a sequence of actions in order to find a solution.

One of the first attempts to solve a variant of Bin Packing Problem with modern reinforcement learning was described in the paper "Solving a New 3D BPP with Deep RL Method" by Hu and Zhang, published in 2017 [9]. The authors have proposed a new, more realistic formulation of the problem. In the Chapter 4.2 we will inspect deeply the use of RL for Bin Packing. We can summarize the pros of this approach, that are the following:

- Natural Markov Decision Process formulation;
- No need of expert examples for training;
- A deeper research in the field of RL for optimization.

Despite this technique has a lot of interesting advantages, there are also some cons due to real world perspective of the problem. In fact the issues are related to the training of RL models, that it is very costly in terms of computational power and training time. Here are the cons:

- Randomly initialized policies perform poorly;
- Slow early in training;
- Large Markow Decision Process involved.

We presented the two possible approaches that nowadays are available for optimizing MILP problems through ML. In the next chapters we will focus deeper on them and finally perform some experiments on data for comparison.

# 4

# Machine learning for MILP problems

In this chapter we will focus on the two main approaches for Bin Packing, and MILP problems through Machine Learning in general:

- Imitation Learning through Graph Convolutional Neural Networks
- Reinforcement Learning in Bin Packing Problem

Now in the following sections we start focusing on those topics.

## 4.1 Imitation Learning through GCNN

The paper "Exact Combinatorial Optimization with Graph Convolutional Neural Networks" by Gasse et al. [8] propose a interesting new approach for learning branching rules in MILP problems optimization. The reason is that optimization problems are typically solved by the branch-and-bound algorithm. The work proposes a new Graph Convolutional Neural Network (GCNN) model for learning branch-and-bound variable selection policies. This is possible creating a custom designed graph representation for encoding the variables and constraints. Training the model via imitation learning from the strong branching expert rule, it

is able to generalize to new instances. The result will be to improve the performances of the classic B&B solver.

### 4.1.1 Background

We start with some background on the main idea behind the approach. Most combinatorial optimization problems can be formulated as MILPs, and the majority of them is solved through branch-and-bound algorithm. As we explained in Chapter 3.1.3, branch-and-bound recursively partitions the solution space into a search tree, and computes relaxation bounds along the way to prune sub-trees that do not contain an optimal solution. This is a sequential decision-making process. Other decisions to be made are the following:

- Node Selection;
- Variable selection.

In many contexts it is common to repeatedly solve combinatorial optimization problems with a similar problem and solution structure. For example day-to-day production planning and sizing problems, or also packing items into pallets referring at our business case. This happens because every day new items are ready to be packed into pallets for shipping, and the day-to-day loading is similar form one day to the other.

Since solutions for similar problems have not so different behaviour, it is appealing to use statistical learning for inspecting similarities among them. This is possible when we deal with a specific class of problems. However, this line of work has two challenges to deal with. First, it is not obvious how to encode the state of a MILP B&B decision process, due to the fact both search trees and integer linear programs can have a variable structure and size. Second, it is not clear how to create a model architecture that leads to rules which can generalize. We

need to be able to generalize at least to a similar number of instances, but also ideally to instances larger than seen during training.

For addressing both the above challenges we can use Graph Convolutional Neural Networks, that we will explain later on. We can focus on variable selection, also known as the branching problem, which lies at the core of the B&B paradigm. The idea is to adopt an imitation learning strategy to learn a fast approximation of strong branching. Strong branching is a high-quality and expensive branching rule. The following keypoints are the two reasons why GCNN are very useful when dealing with this learning approach:

1. First, we propose to encode the branching policies into a GCNN. This specific structure allows us to exploit the natural bipartite graph representation of MILP problems composed by variables and constraints.
2. Secondly we will approximate strong branching decisions by using behavioral cloning with a cross-entropy loss, a less difficult task than predicting strong branching scores.

We will also evaluate this approach on different NP-hard problems, for inspecting if this approach can offer a substantial improvement over traditional branching rules, and if it is able to generalize well outside training instances.

### 4.1.2 Graph Convolutional Neural Networks

For understanding Graph Convolutional Neural Network we need to get familiar with Graph Neural Network idea. The basic graph theory is developed over the idea of encoding data in nodes and edges, that connect nodes based on a specific and defined characteristic defined in advance. Nowadays, a lot of information are represented in graphs. An example are document citation networks: document A has cited document B, so we have a direct edge form A to B. Another example is social media networks. For example friendship on Facebook connects two accounts, the nodes, through an undirected edge representing the friendship (Fig. 4.1).

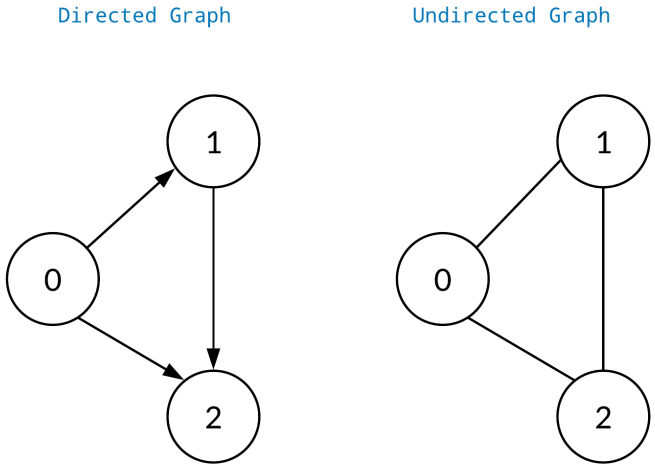


**Figure 4.1:** Directed and undirected graphs.

But how can graphs be used for training a Neural Network? And which are the advantages of this application?

Early variants of Neural Networks can only be implemented using regular or Euclidean data, while a lot of data in the real world have underlying graph structures which are non-Euclidean. We can deal with non-regularity of data structures using Graph Neural Networks. For this specific reason the past few years, different variants of Graph Neural Networks are being developed.

How can those graphs be shaped into features to be fed into the Neural Networks? We can have different options for feeding a neural network with Graph data:

1. Adjacency matrix $A$ of a $N$ node graph: create a $N \times N$ matrix filled with 1 if the nodes in the intersection are connected, and 0 otherwise.

2. Node attributes matrix $X$: it represents the features or attributes of each node. If there are $N$ nodes and the size of node attributes is $F$, then the shape of this matrix is $N \times F$.

3. Edge attributes matrix $E$: useful when edges have its own attributes. If the

size of edge attributes is $S$ and the number of edges available is $n_e$, the shape of this matrix is $n_e \times S$.

The classic method to perform image classification is using Convolutional Neural Networks. An example are the images of digits that are represented in pixels and the Convolutional Neural Network would run sliding kernels across the images for learning the behaviour of the adjacent pixels. Images can also be seen as a graph, where each node represents a pixel and the node feature represents the pixel color value. Edge feature represents the Euclidean distance between each pixel, and the closer 2 pixels are to each other, the larger the edge values. We can see the example in Figure 4.2. This idea makes it possible a non-uniform connection among pixels, because the node connections are dynamic.

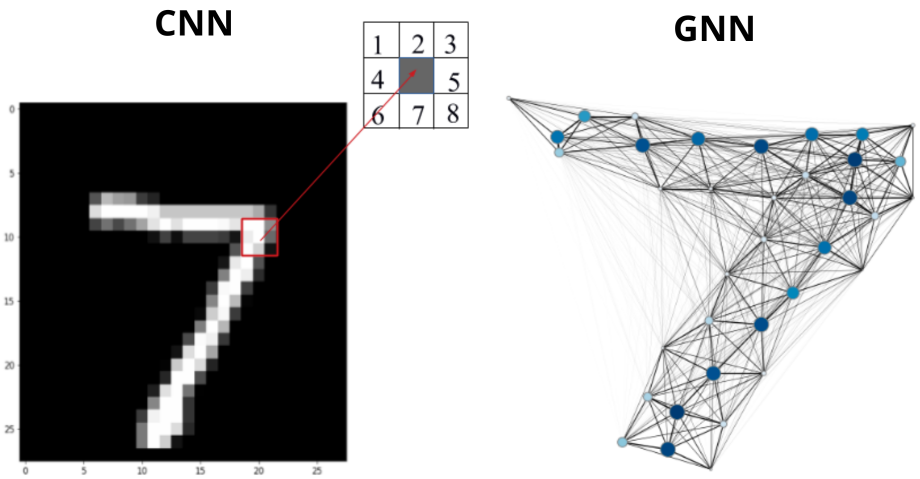


**Figure 4.2:** CNN vs GNN in images.

Summarizing, Convolutional Neural Network work only on data with regular structure, like images (2-dimensional) and text (1-dimensional). Graph Neural Networks are a generalization for working on data in non-Euclidean domain. Graph Neural Networks for this reason are becoming the solution that enables us to capture rich features from the complex relationships among the data.

Now that Graph Neural Networks main ideas are defined, we need to add the concept of convolution. In this way we can better understand how the learning of B&B decisions is performed and why branching policies are efficiently encoded in a GCNN.

The idea behind convolutions is to have filters that act as a sliding window across the whole image. Those filters enable CNNs to learn features from neighboring cells, using weight sharing through the filter. GCCNs perform a similar operations but in this scenario the model learns the features by inspecting neighboring nodes instead of pixels, as shown in Figure 4.4. We can see GCNNs as the generalized version of CNNs where the numbers of nodes connections vary and the nodes are not ordered.

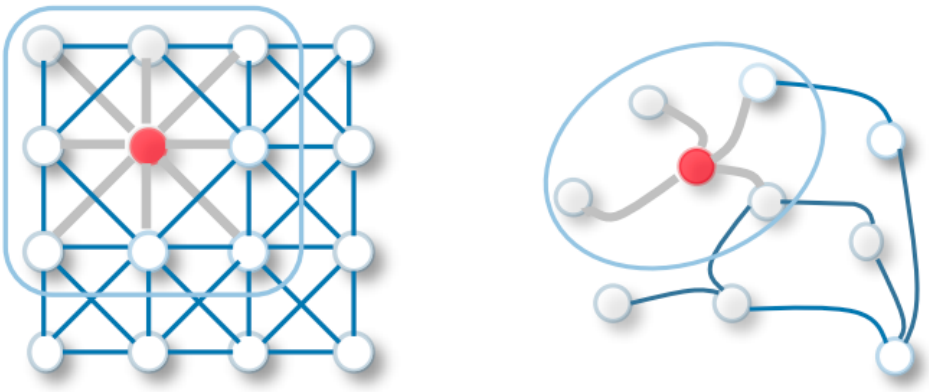

**Figure 4.3:** CNN vs GCNN.

But how a Graph Convolution works from a mathematical point of view? The original idea was inspired by signal propagation along nodes. Spectral GCNs make use of the Eigen-decomposition of graph Laplacian matrix to implement this method of information propagation: this is useful for understanding the graph structure. In Fast Approximation method, used for propagating signal along the nodes, we are not going to use matrix Eigen-decomposition explicitly.

In this forward propagation approach we will take into account the Adjacency Matrix $A$ in addition to the node features. The insertion of A in the for-

ward pass equation enables the model to learn the feature representations based on nodes connectivity. For simplicity we are omitting the bias b that is present in every Neural Network forward propagation. The resulting GCNN can be seen as the first-order approximation of Spectral Graph Convolution in the form of a message passing network where the information is propagated along the neighboring nodes within the graph. In the Equation 4.1, $A^*$ is the normalized version of $A$ that does take in account the node self-loop for including the node features itself. The graph forward pass equation will then be:

$$H^{i+1} = \sigma(W^i H^i A^*) \tag{4.1}$$

The result is that for every GCNN layer a node takes information from the neighbours. And in the case of a two layers the node information is spread at two nodes distance along the graph.

### 4.1.3 Methodology

We need to inspect how those concepts are useful when dealing with learning to branch in B&B algorithm. A mixed-integer linear program is an optimization problem of the form:

$$\arg\min_{\mathbf{x}} \left\{ \mathbf{c}^\top \mathbf{x} \mid \mathbf{A}\mathbf{x} \leq \mathbf{b}, \quad \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \quad \mathbf{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \right\} \tag{4.2}$$

Where the size of a MILP is typically measured by the number of $m$ rows and $n$ columns of the constraint matrix. Now we will describe the terms of the equation 4.2 for a better understanding:

- $c \in R_n$ is called the objective coefficient vector;
- $A \in R^{m \times n}$ the constraint coefficient matrix;
- $b \in R^m$ the constraint right-hand-side vector;

- $l, u \in R^n$ are respectively the lower and upper variable bound vectors;
- $p \leq n$ is the number of integer variables.

The simplest formulation of B&B algorithm repeatedly performs this binary decomposition, giving rise to a search tree. The solving process stops whenever both the upper and lower bounds are equal or when the feasible regions do not decompose anymore, thereby providing a certificate of optimality or infeasibility, respectively. A key step in the B&B algorithm is selecting a fractional variable to branch on in, which can have a very significant impact on the size of the resulting search tree, as shown in Figure 4.4. The branching strategy consistently resulting in the smallest BB trees is strong branching that consist in computing the expected bound improvement for each candidate variable before branching. It requires the solution of two LPs for every candidate, and for this reason is very expensive. Modern B&B solvers, for dealing with this complexity problem, rely on hybrid branching instead: it computes strong branching scores only at the beginning of the solving process and gradually switches to simpler heuristics. We can notice how the variable selection in B&B tree can be seen as a Markov Decision Process (Fig. 4.4). We can consider the solver to be the environment, and the brancher the agent.

At the decision number $t$ the solver is in a state $s_t$, which comprises the B&B tree with information about:

- All past branching decisions;
- The best integer solution found so far;
- The LP solution of each node;
- Currently focused leaf node;
- Any other useful solver statistics.

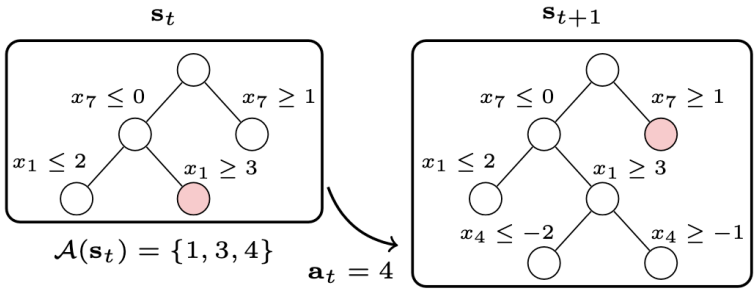


**Figure 4.4:** B&B variable selection as a Markov decision process. On the left, a state $s_t$ comprised of the branch-and-bound tree, with a leaf node chosen by the solver to be expanded next (in pink). On the right, a new state $s_{t+1}$ resulting from branching on the variable $a_t = x_4$.

The brancher selects a variable at among all the possible ones $A(s_t) \subseteq \{1, ..., p\}$ at the currently focused node, according to a policy $\pi(a_t \mid s_t)$ (Eq. 4.3). The solver in turn extends the B&B tree, solves the two child LP relaxations, runs any internal heuristic, prunes the tree if warranted, and finally selects the next leaf node to split. Then in a new state $s_{t+1}$, and the brancher is called again to take the next branching decision. This process continues until there are no leaf node left for branching. As a Markov decision process, B&B is episodic, where each episode amounts to solving a single instance. The probability of a trajectory $\tau = (s_0, ..., s_T) \in T$ then depends on both the branching policy $\pi$ and the remaining components of the solver (Eq. 4.3). A natural approach to find good branching policies is RL with a carefully designed reward function. But we will adopt an imitation learning scheme, for the reasons shown in Chapter 3.1.2.

$$p_\pi(\tau) = p\left(\mathbf{s}_0\right) \prod_{t=0}^{T-1} \sum_{\mathbf{a} \in \mathcal{A}(\mathbf{s}_t)} \pi\left(\mathbf{a} \mid \mathbf{s}_t\right) p\left(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}\right) \tag{4.3}$$

### 4.1.4 Imitation learning with GCNN

We train by behavioral cloning using the strong branching rule, which suffers a high computational cost but usually produces the smallest B&B trees. The main difference comparing this approach to RL is that we need to first run the expert on a collection of training instances of interest. Than we need to collect the data in a record a dataset of expert state-action pairs $\mathcal{D} = \{(s_i, a_i^*)\}_{i=1}^N$. Finally we learn our policy by minimizing the cross-entropy loss (Eq. 4.4).

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{(\mathbf{s},\mathbf{a}^*)\in\mathcal{D}} \log \pi_\theta \left(\mathbf{a}^* \mid \mathbf{s}\right) \tag{4.4}$$

Here comes the state encoding part. The right side of Figure 4.5 provides an overview of our architecture. The idea is to represent the MILP problem as a bipartite graph made by constraints $c$ and varibles $v$, as shown on the left side of the Figure 4.5. An edge $(i, j) \in E$ connects a constraint node $i$ and a variable node $j$ if the latter is involved in the former, that is if $A_{ij} \neq 0$.

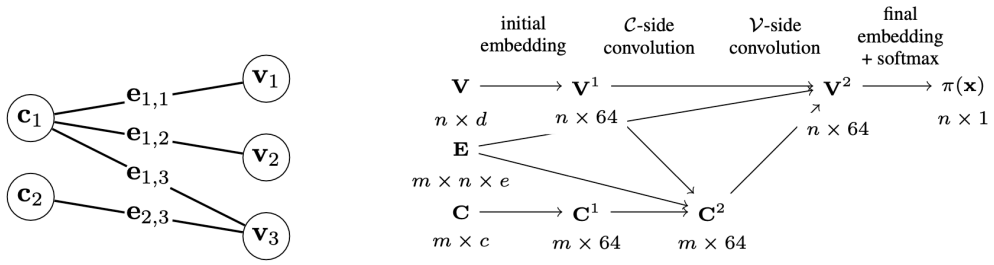


**Figure 4.5:** Encoding B&B in GCNN and training.

We encode the state $s_t$ of the B&B process at time $t$ as a bipartite graph with node and edge features $(G, C, E, V)$ we described previously. On one side of the graph are nodes corresponding to the constraints in the MILP, one per row in the current node's LP relaxation, with $C \in R^{m\times c}$ their feature matrix. On the other

side are nodes corresponding to the variables in the MILP with $V \in R^{n \times d}$ their feature matrix. $E \in R^{m \times n \times e}$ represents the sparse tensor of edge features.

The intuition in this case is that the graph structure is the same for all LPs in the B&B tree, which reduces the cost of feature extraction. We note that this is really only a subset of the solver state, which technically turns the process into a partially-observable Markov decision process. From the learning point on view we share the general idea behind the Paper [8], because inspecting it deeply would require an entire chapter. For more information we can see the supplemantal materials of the paper itself.

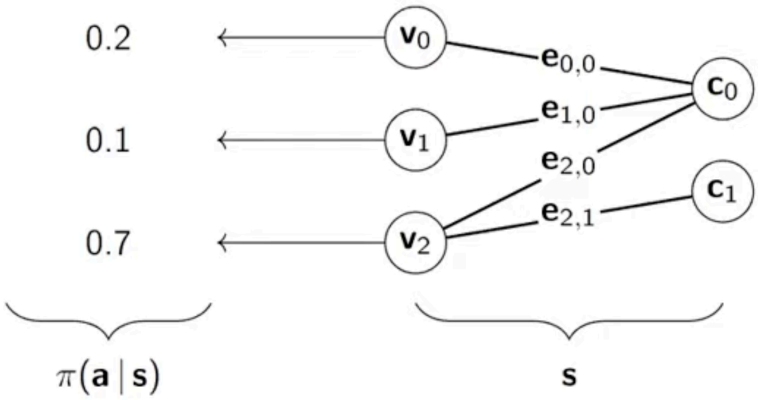


**Figure 4.6:** Probability distribution over candidate nodes.

The model takes as input our bipartite state representation $s_t = (G, C, E, V)$ and performs a single graph convolution, in the form of two interleaved half-convolutions. Following the graph-convolution layer (Fig. 4.5), we obtain a bipartite graph with the same topology as the input, but with potentially different node features, so that each node now contains information from its neighbors. We obtain our policy by discarding the constraint nodes and applying a final 2-layer perceptron on variable nodes, combined with a softmax activation to produce a probability distribution over the candidate branching variables (Fig. 4.6).

Those are the non-fixed LP variables, that are represented by the leaf nodes of the tree.

In the literature of GCNN, it is common to normalize each convolution operation by the number of neighbours. This might result in a loss of expressiveness, as the model then becomes unable to perform a simple counting operation (e.g., in how many constraints does a variable appears). This learning to branch technique uses un-normalized convolutions instead. However, this introduces a weight initialization issue. Indeed, weight initialization in standard CNNs relies on the number of input units to normalize the initial weights, which in a GCNN is unknown beforehand and depends on the dataset and we don't know it in advance. To overcome this issue, a simple affine transformation is adopted. $x \leftarrow (x - \beta)/\sigma$, which we call a pre-norm layer, applied right after the sum operation. $\beta$ is the empirical mean and $\sigma$ is the empirical standard deviation of $x$ on the training dataset, and fixed once and for all before the actual training. This is possible and useful because bipartite MILP graphs that we are dealing with have a limited number of nodes, if we compare them to other type of graphs.

## 4.2 BPP with Reinforcement Learning

Before describing the actual method, we give an overview about the scientific research on the topic "Reinforcement Learning for BPP". One of the first attempts to solve a variant of Bin Packing Problem with modern reinforcement learning was by Hu et al., in 2017 [9]. The authors have proposed a new, more realistic formulation of the problem, where the bin with the least surface area that could pack all 3D items is determined. The state space, $S$, is denoted by a set of sizes (height, width, and length) of the items that need to be packed. The second approach is the one proposed by Bello et al. in 2017, which utilizes the Pointer Network, is used to output the sequence of actions, $A$, that represent the sequence of items to pack. In this approach the Reward $R$, is calculated as the value of the

surface area of packed items. The more surface is used, the better the algorithm is packing items.

In all RL approaches, the baseline is provided by the known heuristic. The improvement over the heuristic and no learned item selection was the valuation metric. Most of Reinforcement Learning 3D Bin Packing methods usually solve the problem with limited resolution spatial discretization. The other problem is that they cannot deal with complex practical constraints well. Those can for example be packing stability or specific packing preferences. The paper "Learning efficient 3D BPP on Packing Configuration Trees" by Zhao & Yu [10] propose a new hierarchical representation called Packing Configuration Tree (PCT).

PCT is a description of the state and action supported by bin packing, and the packing policy can be learned using Reinforcement Learning. The idea is to have a size of the packing action space that is proportional to the number of leaf nodes, that are the possible candidate placements.

### 4.2.1 Background

Learning based approaches for MILP usually perform better than heuristic methods. However, the learning is hard to converge with a large action space, which has greatly limited the applicability of those methods in real world scenario.

PCT can be imagined as a dynamically growing tree where the internal nodes describe the space configurations of packed items. Leaf nodes instead are the packable placements of the current item that needs to be packed. The idea is to extract state features from PCT using graph attention networks, called GAN, which encodes the spatial relationships between nodes. The state feature is the input into the actor and critic networks of the DRL model (Chapter 2.2.4). The trained actor network is able to weight the leaf nodes and to give as output the bin placement.

RL model learns a discriminant function for the candidate placements, result-

ing in an effective and robust packing policy if compared to the heuristic methods.

Practical constraints as in the majority of literature for 3D-BPP are the ones by Martello et al. [11] that only considers the basic non-overlapping following two constraints:

$$p_i^d + s_i^d \leq p_j^d + S^d \left(1 - e_{ij}^d\right) \quad i \neq j, i, j \in \mathcal{I}, d \in \{x, y, z\} \tag{4.5}$$

$$0 \leq p_i^d \leq S^d - s_i^d \quad i \in \mathcal{I}, d \in \{x, y, z\} \tag{4.6}$$

Where $p_i$ means the front-left-bottom coordinate of item $i$ and $d$ the coordinate axis, $e_{ij}$ takes value 1 otherwise 0 if item $i$ precedes item $j$ along $d$. The idea is to start packing from the front-left-bottom corner of the bin, and than expand the tree according to the new corners created in the bin. With this simple trick we are able to grow a decision tree, the so called PCT (Fig. 4.9). This structure encodes the position of the bins and is very useful for packing, especially since this set of decisions can be learnt by a RL algorithm.

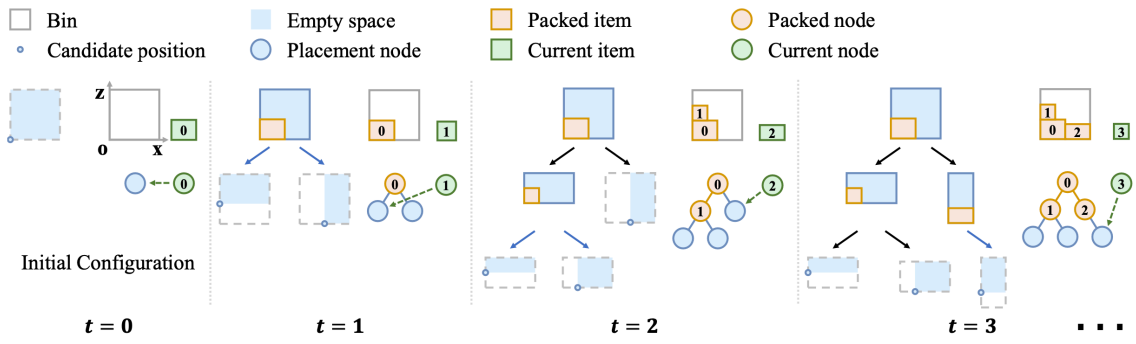


**Figure 4.7:** PCT expansion illustrated using a 2D example. A new item introduces a series of empty spaces and new possible placements: the left-bottom corner of the empty space.

### 4.2.2 Packing configuration tree

In this chapter we start describing the features of the Packing Configuration Tree. Every time a rectangular item $n_t$ is added to a given packing, has position $(p_n^x, p_n^y, p_n^z)$ at time step $t$.

The single packing introduces a series of new candidate positions where future items can be accommodated (Fig. 4.9). We can combine the position with the item orientation $o \in O$ for $n_t$ based on existing positions, so we can get the candidate placements with different position and orientations for each single object. This is a very useful feature with objects that have constraints on orientation packing, such as fragile boxes. We can see the packing process can be seen as a placement node being replaced by a packed item node. Every time it happens, new candidate placement nodes are generated as children. We can think about this process in $2D$, and expand later the same way of thinking in $3D$ real world scenario. As the packing time step $t$ goes on, these nodes are updated iteration after iteration. Finally a dynamic packing configuration tree is formed, and we call it $T$.

We need in the first place to define the main vectors for performing the PCT expansion:

- $n_t$ is the new rectangular item to pack, has a defined position at time step $t$;
- Internal node set $B_t \in T_t$ is a vector that represents the space configurations of packed items;
- Leaf node set $L_t \in T_t$ represents the packable candidate placements.

During the procedure, leaf nodes that are no longer feasible due to other packed items that cover the spot, will be removed from the vector $L_t$. When there is no packable leaf node that makes $n_t$ satisfy the constraints of placement, tree growth stops.

The main idea is to promote this problem for practical demands in real-world scenario, because other similar techniques are much arder to use in a company context. 3D-BPP is able to satisfy more complex practical constraints compared to other RL based techniques.

Talking about the packing policy over $L_t$, it is defined as $\pi(L_t \mid T_t, n_t)$. Those are the probabilities of selecting leaf nodes from $L_t$ given $T_t$ and $n_t$ as a state. The aim is to find the best leaf node selection policy through RL. In this way we can expand the PCT with the best constraints and be able to append the higher number of items possible.

The performance of online 3D-BPP policies has a strong relationship with the choice of leaf node expansion schemes. We need to incrementally calculate new candidate placements, that are iteratively introduced by the just placed item $n_t$. The problem is that designing an expansion scheme from scratch is not simple.

So the idea is to take in account some placement rules from classic packing problems that have been proposed in scientific literature. Some of them are "Corner Point" concept by Martello et al. in 2000 [12] and "Extreme Point" by Crainic et al. [13]. The hard task is encoding three different descriptor vectors. The raw space configuration nodes $B_t$,$L_t$ and $n_t$ are presented by descriptors in different formats. The idea is to use three independent node-wise Multi-Layer Perceptron (MLP) blocks for projecting these indipendent descriptors into homogeneous features in the nodes of the tree 4.7.

$$\hat{\mathbf{h}} = \{\phi_{\theta_B}(\mathbf{B}_t), \phi_{\theta_L}(\mathbf{L}_t), \phi_{\theta_n}(n_t)\} \in \mathbb{R}^{d_h \times N} \tag{4.7}$$

$d_h$ is the dimension of each node feature and $\phi_\theta$ is a Multi Layer Perceptron block with its parameters $\theta$.

The feature number $N$ should be $(\mid B_t \mid + \mid L_t \mid +1)$, which is also a variable. The GAT layer is used to transform $\hat{h}$ into high-level node features.

### 4.2.3 GAT attention layer

For making features comparable we use the very famous Scaled Dot-Product Attention by Vaswani et al. [14]. This strong idea makes possible to apply an attention framework to each node for calculating the relative weight of one node respect to another. These relation weights are normalized and used to compute the linear combination of features $\hat{h}$. The feature of node $i$ embedded by the GAT layer that can be represented by the following Formula 4.8, that is not so easy to understand.

$$\mathrm{GAT}\left(\hat{h}_i\right) = W^O \sum_{j=1}^{N} \mathrm{softmax}\left(\frac{\left(W^Q \hat{h}_i\right)^T W^K \hat{h}_j}{\sqrt{d_k}}\right) W^V \hat{h}_j \qquad (4.8)$$

Where $W^Q \in \mathbb{R}^{d_k \times d_h}$, $W^K \in \mathbb{R}^{d_k \times d_h}$,$\mathbb{W}^V \in \mathbb{R}^{d_v \times d_h}$ and $W^O \in \mathbb{R}^{d_h \times d_v}$ are projection matrices. The $d_k$ and $d_v$ dimensions are the ones of projected features. The softmax operation normalizes the relation weight between node $i$ and node $j$.

Given the node features $h$, we need to decide the leaf node indices for accommodating the current item $n_t$. Since the leaf nodes vary as the PCT keeps growing, we use a pointer mechanism (Vinyals et al. [15]). This method can be seen as a context-based attention over variable inputs to select a leaf node from $L_t$ for packing the $n_t$ item.

We still adopt Scaled Dot-Product Attention [14] for calculating pointers. An important part is that the global context feature $\bar{h}$ is aggregated by a simple mean operation on $h$: $\bar{h} = \frac{1}{N}\sum_{i=1}^{N} h_i$

The global feature $\bar{h}$ is projected to a query $q$ by matrix $W^Q$ and the leaf node features $h_L$ are utilized to calculate a set of keys $k_L$ by $W^k$.

### 4.2.4 Markov Decision Process Formulation

For understanding how the agent learns in this RL framework we need to focus on the MDP formulation. As we saw in Chapter 2.2.2, the MDP consists in a framework composed by states, actions, transitions probabilities and a reward function that the agent tries to maximize.

The online 3D-BPP decision at time step $t$ depends on the tuple $(T_t, n_t)$. It means the decision is based on the tree structure, containing all the information about nodes, and the added item to pack. The decision can be formulated as Markov Decision Process, which is constructed with state $S$, action $A$, transition $P$, and reward $R$. We solve this MDP with Deep Reinforcement Learning agent.

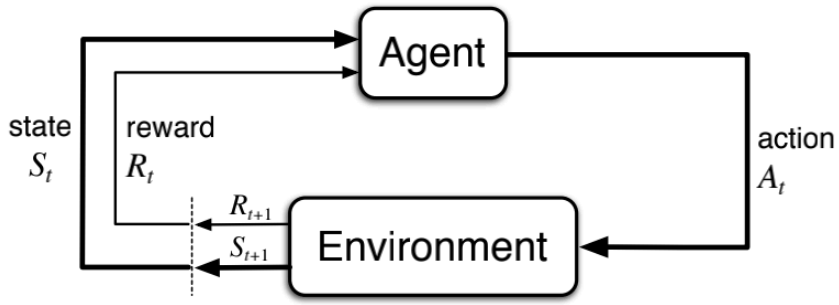


**Figure 4.8:** The agent–environment interaction in a Markov decision process.

The model is formulated as follows:

- **State**: The state $s_t$ at time step $t$ is represented as $s_t = (T_t, n_t)$, where $T_t$ consists of the internal nodes $B_t$ and the leaf nodes $L_t$. It's important to notice that each internal node $b \in B_t$ is composed by a spatial configuration of sizes $(s_b^x, s_b^y, s_b^z)$ and coordinates $(p_b^x, p_b^y, p_b^z)$ that describe the specific packed item. The current item $n_t$ has a size defined by a tuple $(s_n^x, s_n^y, s_n^z)$. As we shown before, extra properties will be appended to $b$ and $n_t$ for specific packing preferences, some examples can be density or item category. The descriptor for leaf node $l \in L_t$ is a placement vector of sizes $(s_o^x, s_o^y, s_o^z)$ and position coordinates $(p^x, p^y, p^z)$. Those coordinates indicate the sizes

of $n_t$ along each dimension after packing the item starting from $o \in O$, that is the previously defined corner. Only the packable leaf nodes which satisfy placement constraints are provided, the other are removed from $L_t$ vector.

- **Action**: The action $a_t \in A$ is the index of the selected leaf node $l$, denoted as $a_t = index(l)$. It means that basically the action to perform is the leaf where the item will be packed. For this reason the action space $A$ has the same size as $Lt$. This choice is different from existing works, because our action space depends only on the leaf node expansion scheme and the already packed items $B_t$. An interesting outcome of this packing trick is that the method can be used to solve online 3D-BPP with continuous solution space, that is not a trivial task.

- **Transition**: The transition $P(s_{t+1} \mid s_t)$ is jointly determined by the learned policy $\pi$ and the probability distribution of sampling items. The data sequences that we will describe in the Chapter 3.1 are generated from an item set $I$ in a uniform distribution, for testing the method.

- **Reward**: Our reward function $R$ is defined as $r_t = c_r w_t$ once $n_t$ is inserted into PCT as an internal node successfully. Otherwise $r_t = 0$ and the packing episode ends. The value $c_r$ is a constant and $w_t$ corresponds to the weight of item $n_t$. The choice of $w_t$ depends on the customized needs, and can be decided depending on the task needs. For simplicity and clarity we will set $w_t$ as the volume occupancy $v_t$ of the item $n_t$, defined by the multiplication of the dimensions $v_t = s_n^x s_n^y s_n^z$. This point is useful because we can add to the problem different weights based on the needs of the specific business task.

Now that we have a clear view on the MDP, we can focus on the training method. A DRL agent seeks for a policy $\pi(a_t \mid s_t)$ to maximize the accumulated discounted reward. The actual training, as we mentioned before, is a mix of techniques and state-of-the-art methods that now we try to describe in a simple and understandable way.

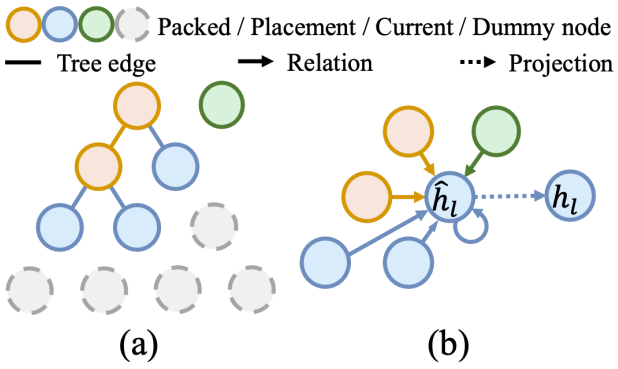


**Figure 4.9:** Visualization of batch calculation for PCT, in this way the process can be speeded up.

Our DRL agent is trained with the ACKTR method (Wu et al. 2017 [16]). According to this method the actor weighs the leaf nodes $L_t$, and outputs the policy distribution $\pi_\theta(L_t|T_t, n_t)$ we need for choosing the PCT leaf where the item will be packed. The critic maps the global context $\bar{h}$ into a state value prediction. In this way we try to increase the predicted value. This is possible predicting how much accumulated discount reward the agent can get from $t$, and this estimation helps the training of the actor. The action $a_t$ is sampled from the distribution $\pi_\theta(L_t \mid T_t, n_t)$ for training, and we take the argmax of the policy for the test.

From a computational point of view there are some interesting tricks performed by ACKTR. The technique consists in running multiple parallel processes for gathering training samples. The node number $N$ of each sample varies with the time step $t$ and the packing sequence of each process. For performing a batch calcuation, the trick is to full-fill the PCT to a fixed length with dummy nodes, as illustrated by Figure 4.9 (a). These redundant nodes are eliminated by the Masked Attention described before, during the feature calculation of GAT. But they help because during the calculation we have a fixed length PCT. The aggregation of $h$ only happens on the nodes that are eligible. For preserving node spatial relations among the nodes, the single state $s_t$ is embedded by GAT as a fully connected graph. An example it's shown in Figure 4.9 (b).

In conclusion, with this parallel process trick and using fixed length PCT we can speed up the training process. This is important because, as we explained in previous chapters, with RL techniques the training phase is very expensive and time consuming.

# 5 Experiments

In this chapter we will run some experiments on MILP problems, taking into account the different challenges we highlighted in this thesis work. Since we described two main method for learning to solve MILP problems, we will show some results applying those techniques in a real world scenario. The main focus will be the time spent for finding a solution, that is the initial business problem we are trying to solve, due to the exponential growth in complexity in MILP classic algorithms.

As mentioned before, one of the major issues in the learning Operational Research field is that the majority of developments are made by industry and we have no access to open source code. This happens because those techniques are highly valuable in a business scenario, so the companies use this technology as an competitive advantage against competitors. Some examples are famous solvers as AWS Amazon Bin Packing solver through Reinforcement Learning, IBM solvers or Gurobi. The last mentioned one is a company with a vertical business model focusing on mathematical optimization. This is a limitation for experiments in the final part of this thesis work, because we want to focus on open source tools.

On the other hand, it means that the topics covered in this thesis are highly valuable from a business point of view. We can consider this work as a survey on the state-of-the-art MILP and BPP solvers through Machine Learning.


**Figure 5.1:** Companies selling OR through Machine Learning products.

In the next subsections we will see how the different ML techniques give better results than classic optimization algorithms in a MILP context. We will execute GCNN Imitation Learning (Section 5.1) on a randomly generated classical MILP problem. We will compare the solution to classical Branch and Bound instances. Then we will try to compare a classical Bin Packing problem solved with SCIP open source library to a Reinforcement Learning approach (Section 5.2).

In this way we will test the best open source libraries available from a practical point of view. As we said those are not libraries designed for business cases, because they are developed for academic testing purposes. But, as we will see, the results are anyway able to outperform classical MILP solvers in terms of time required for solution.

## 5.1 GCNN experiments on MILP

In this section we will describe how we tested the Imitation Learning through GCNN algorithm on a MILP problem. We will describe in the first place Ecole [17], the library that made possible the experiments. And we will inspect how it

works according to the theoretical part. Sadly, after contacting the main contributor to the open source project, I discovered that the library is no longer maintained. But, anyway, we can start from the code for developing our experiments on a MILP problem. Than we will give some considerations on the results.

### 5.1.1 Ecole project

Ecole is a library to simplify Machine Learning research for Combinatorial Optimization. Ecole exposes several key decision tasks in general purpose combinatorial optimization solvers. The mission is making the library a standardized platform that will lower the bar of entry and accelerate innovation in the field. Because as we said the problem of this Machine Learning applications on MILP are the high entry barriers in terms of coding very complex networks and data structures for encoding data. The goal of Ecole project is to gain interest through a unified ML-compatible API for MILP problem solving, that will help attract interest from both the traditional ML and OR communities.

For describing Ecole characteristics we need to start form the modularity concept. For example a branching environment is defined with a node bipartite graph observation and the negative number of new nodes created as a reward. But we can also create new strategies according to new future improvements. From a scalability point of view, Ecole is designed to add as little overhead as possible on top of the solver. And it allows straightforward parallelism in Python with multithreading.

The Ecole core is also written in C++ for interacting directly with the low-level solver API. This makes the library extremely efficient and comparable to SCIP in terms of low-level coding. Coding in C++ is an example of high barrier in developing those systems from scratch. The library provides a thin Python API returning Numpy arrays for being able to interface directly with ML libraries such as torch.

Ecole supports also the already mentioned state-of-the-art open-source solver SCIP as a back-end for performing all the non-learned tasks. In addition, the library also provides out-of-the box instance generators for classical CO problems. The generated instances can be saved to disk, or passed directly to Ecole environments from memory. Since the generation of BPP problem is not implemented, we will choose a generic MILP problem for testing the performances in this section of the thesis. Then we will focus on BPP in the Section 5.2 with the specific 3D-BPP algorithm.

### 5.1.2 Training with GCNN Imitation Learning

The library supports two control tasks. The first is related to hyperparameter tuning for selecting the best solver hyperparameters. The second is variable selection, related to deciding sequentially on the next variable to branch on during the construction of the branch-and-bound tree. We will compare the results to an empty baseline environment that can be used to benchmark against the solver, that is basically an optimized SCIP solver with no learning.

In our experiments we will reproduce a simplified version of the paper of Gasse et al. 2019 [8] on learning to branch with pytorch using Ecole library in addition. We collect strong branching examples on randomly generated maximum set covering instances, then we will train a bipartite GCNN with state encodings to imitate the expert by classification The idea is to predict the probabilities of the next B&B node to branch on. And than use the higher probability node for the next branching step.

The biggest difference with the original paper experiment is that we collected only $n = 1.000$ training examples of expert decisions for training, to keep the time needed reasonable according to our laptop pc hardware. Even with free cloud GPUs working with more instances were hard to manage. As a consequence, the resulting policy is a bit undertrained if compared to the $n = 100.000$ training

samples considered for training in the paper experiment. Despite of this computational problem, as we will see, the results are able to show some improvements to default SCIP solver.

The explore-then-strong-branch described in the paper is not implemented by default in Ecole. For being able to diversify the states in which we collect examples of strong branching behavior, we follow a weak but cheap expert that is the pseudocost branching. And only occasionally we call the strong branching. This can be implemented in Ecole by creating a custom observation function, which will randomly compute and return the cheap pseudocost scores or the expensive strong branching scores. This operation can be written directly in Python language.

We can now create the environment with the chosen correct parameters. We used 1h time limit for computing and a sampling of 5% from expert. Our environment will return the node bipartite graph representation of branch-and-bound states used in used in the paper with the custom Ecole function. As we described in Chapter 4.1, on one side of that bipartite graph nodes represent the variables of the problem. On the other side of the bipartite graph, the nodes represent the constraints of the problem, with a vector encoding features of that constraint. An edge links a variable and a constraint node if the variable participates in that constraint.

Now we loop over the instances, following the strong branching expert 5% of the time and saving its decision, until enough samples are collected. We select only the decision of strong branching expert among the other ones, for capturing samples at different stages of the B&B tree. This is the reason of mixing cheap pseudocost and expensive strong branching scores.

The next step is to train a GNN classifier on these collected samples to predict similar choices to strong branching. We will first define pytorch geometric data

classes to handle the bipartite graph data. We can then prepare the data loaders, for looping through data and train the GCNN. Next we need to define our graph neural network architecture. We start with linear embedding layers, with a common dimension feature output of $lenght = 64$. Then we perform the two half convolutions described in the Paper [8]. Then we create a final Multi Layer Perceptron on the variable features for predicting probabilities over candidate nodes. With this model we are finally able to predict a probability distribution.

With randomly initialized weights, we notice that before training the model, the initial distributions tend to be close to uniform. Now we need to define two functions: one for training the model on a whole epoch and compute metrics, and one for padding tensors when doing predictions on a batch of graphs of potentially different number of variables. After this operations, we can actually train the model.

We trained the model twice. In a first place with $n = 100$ expert samples and than with a $n = 1000$ expert samples. The entire process of generating samples and training with 1000 samples took several hours of computing.

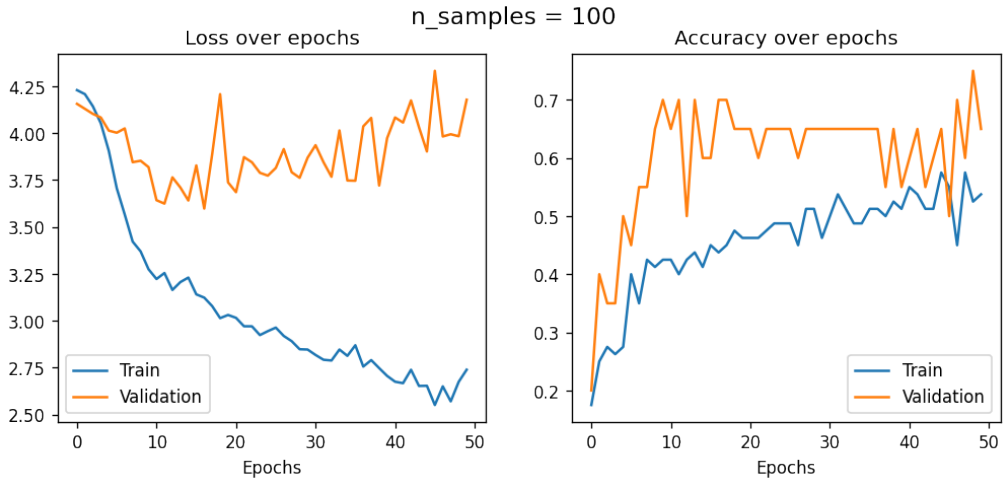


**Figure 5.2:** Training curve with 100 instances.

In a first place we trained for 50 epochs on generated data with a test set composed of $n = 100$ sampled items. As we can see from the learning curves in Figure 5.2 the training doesn't perform so good in terms of loss function. The main issue is that the validation loss starts to grow instead of decreasing. This means that the model is not able to generalize to validation set on the basis of the data in training set. Anyway the accuracy seems to grow over time, especially on validation test. We need to be careful about this results because we are training with a very little number of samples and the results can be biased. For this reason the next experiment takes into account 1000 samples.

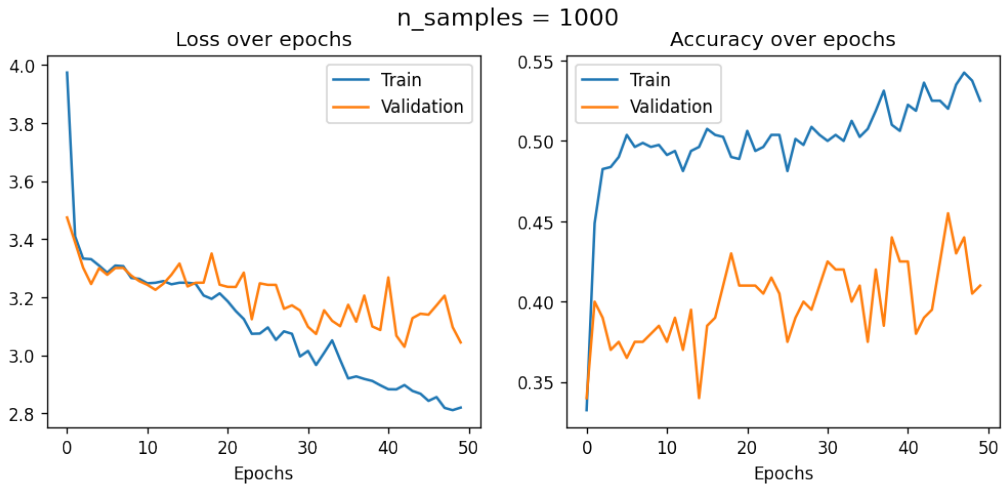


**Figure 5.3:** Training curve with 1000 instances.

The result in Figure 5.3, despite of a better loss function behaviour on the validation set, seems to have a bad accuracy. If compared to the training with 100 samples. But we are sure that, with more and more epochs, the results of the bigger number of samples would be so much better than the smaller one. In the next subsection we will see how, increasing the number of samples, we outperform the non learned policy.

### 5.1.3 Results

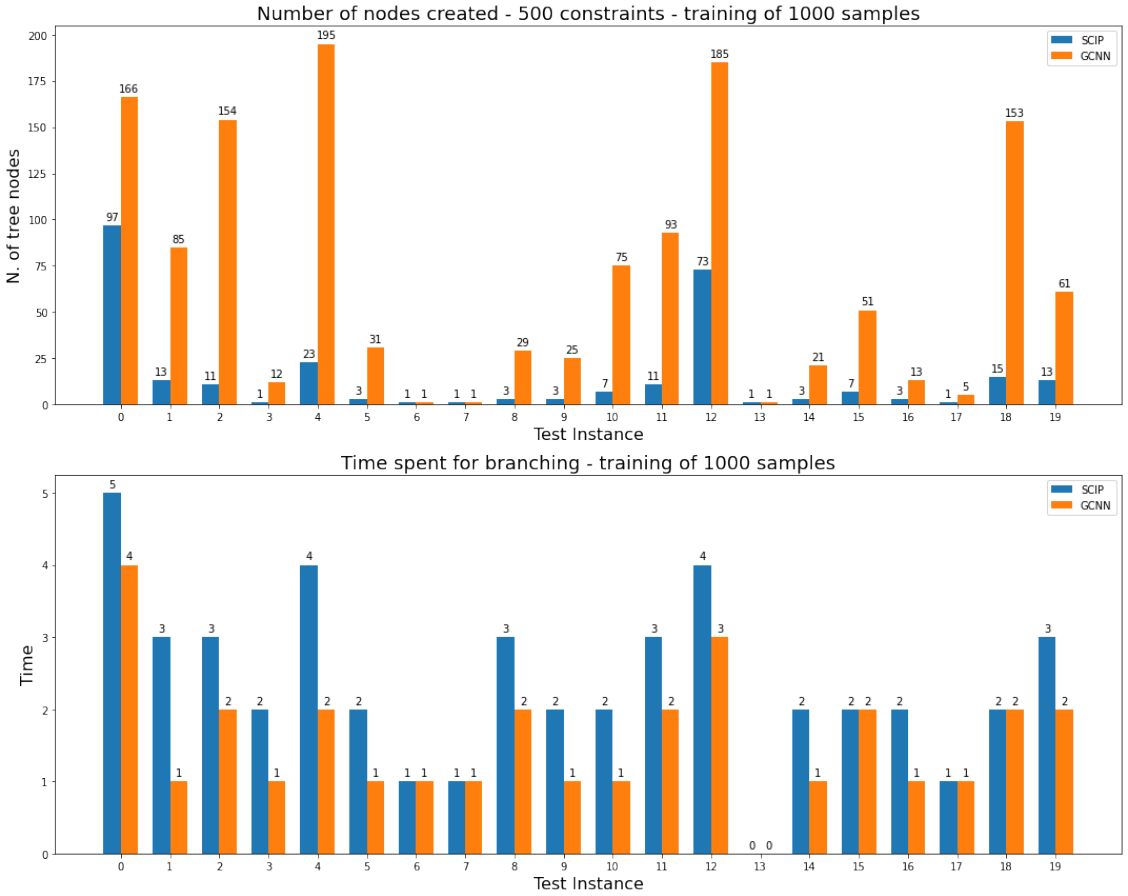


**Figure 5.4:** Results with 1000 sample training and the same number of constraints in test set, on 20 different instances.

Finally, we can evaluate the performance of the model. We first define appropriate environments. For benchmarking purposes, we include a trivial environment that merely runs SCIP. In Figure 5.4 and Figure 5.5 we have the results. We decided to encode a MILP problem with in a matrix with 1000 samples and 500 constraints. In this way we can encode the problem in a bipartite graph and run two branching experiments. The first one takes into account a train set and a test set with a equal number of constraints. But, as we reported, this imitation learning method is able to generalize on a higher number of constraints and samples

on the test set. For this reason the second experiment is performed with a higher number of constraints with respect to the train set ones.

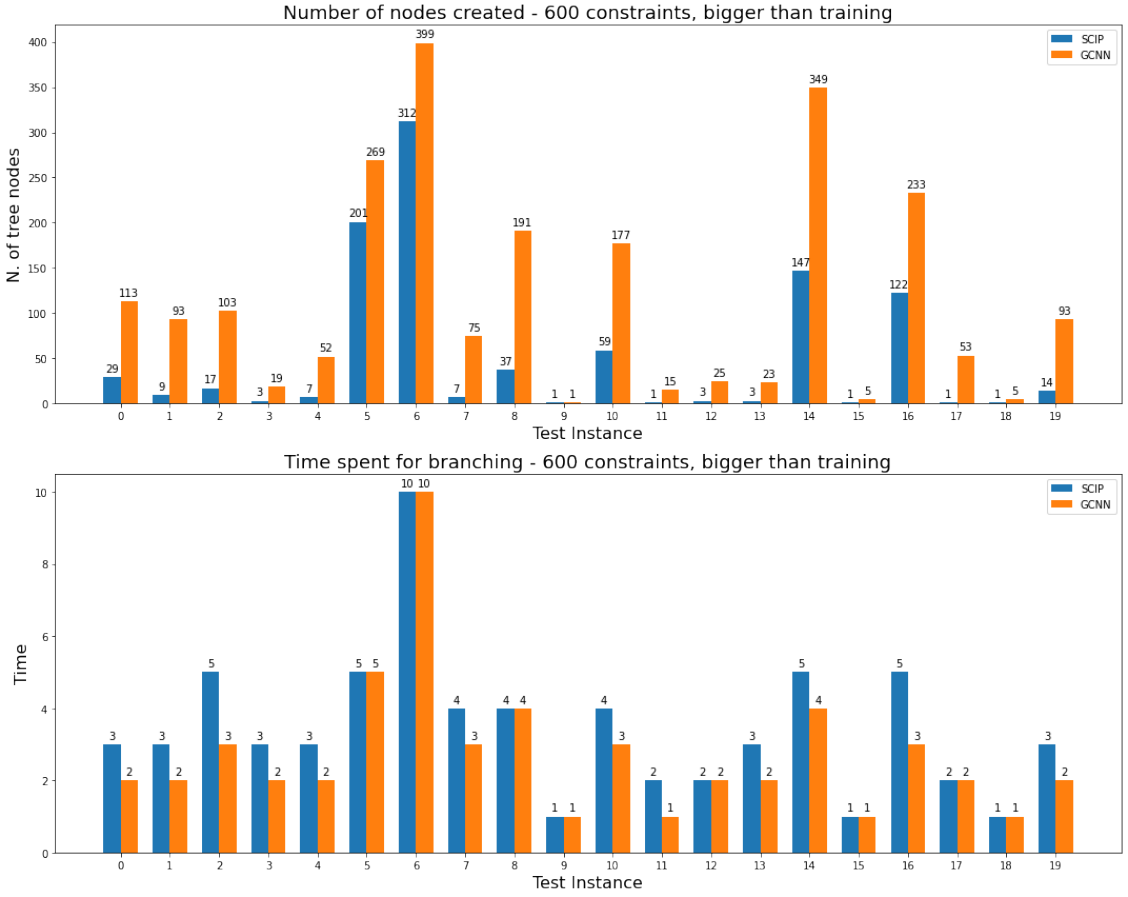


**Figure 5.5:** Results with 1000 sample training and increased number of constraints in test set w.r.t. train, on 20 different instances.

Looking at the results of the first experiments in Figure 5.4, we see that the trees size created for finding the best solution are bigger than the classical no learning SCIP environment. This seems strange because this learning method would be able to learn to create smaller trees and speed up the process. But happens that with such a small training the trees generated are much bigger. But looking at Figure 5.6, reported in the Paper [17], we can see that B&B trees with 100.000 training samples and a 10.000 epochs training are getting smaller and smaller. So

in our case the problem is the computational power for running the experiments. The good news in that, despite of bigger trees, the time graph over test instances gives pretty good results. In fact the GCNN learned policy gives a quicker result in test instances. This means that with this technique we can speed up the process even with a relatively small number of training samples.

The second experiment is interesting because, even if the number of test constraints became bigger than the train ones, the results are stable in terms of time spent for branching. That is what matters in a real case business scenario. We can see the results in Figure 5.5. We are sure that with a longer training we would reach results that can be better and better. The reason is also that in learning curves in Figure 5.3 there is so much space for improvement with a higher number of epochs.

In next section we will also take a look to some experiments on RL policy called 3D-BPP. Before giving general conclusions in the Conclusions [6].

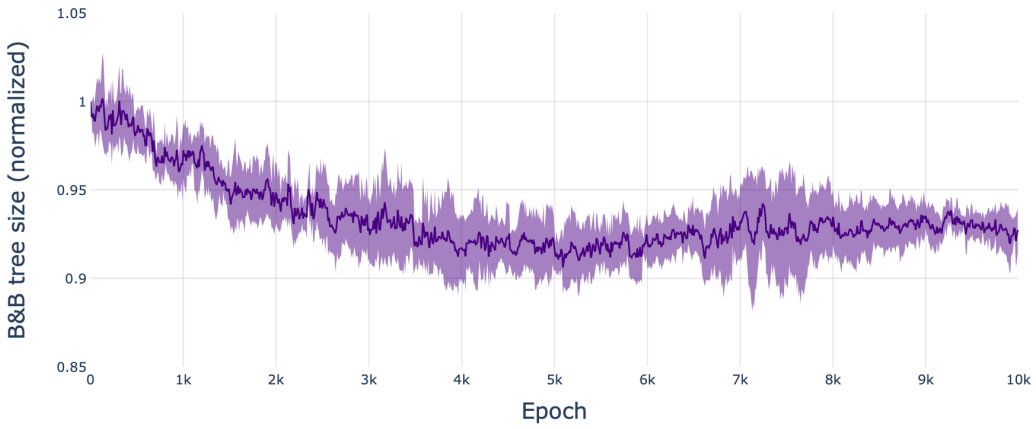


**Figure 5.6:** Training curve of a branching environment on randomly generated instances reported in the paper, with 100.000 training samples. This is the normalized performance on validation instances, the it is lower the more the GCNN is learning to branch. .

## 5.2 RL 3D-BPP experiments

The 3D-BPP Reinforcement Learning framework has no libraries to support experiments on data. For this reason we inspected the source code of the paper experiments to make possible recreate the framework used for training the model. And we will try to compare the packing behaviour to the OR-Tools Suite by Google, that rely on SCIP.

The idea, as described in Chapter 4.2, is to enhance the practical applicability of 3D bin packing problem via learning on a packing configuration tree. The training is performed through a complex DRL model containing GAT mechanisms. But, despite the complexity of the training, the result in a business context would be to easily deal with practical constraints and well-performing in terms of time.

From the GitHub repository we inspected we found some keyponts that are useful for describing the experiments we will perform:

- Bin size and item sizes can be set arbitrarily;
- It's provided an environment for running 3D-BPP scripts on data;
- We can have a stable RL training that is very time and computational cost consuming ;
- We can start from pre-trained network weights for better performances on similar data.

The original dataset consists of 3000 trajectories, each with 150 items. A trajectory is a sequence of packed items in a bin that represents the packing tree we want to learn to build with Reinforcement Learning. Every item is a tuple of length 3 or 4, the first three numbers of the item represent the size of the item, the fourth number, if any, represents the density or the weight of the item. The pretrained models are trained on $(10, 10, 10)$ dimension bins.

### 5.2.1 OR Tools results

For solving classical Bin Packing with classical MILP solvers, as we said in chapter 3.1.5, we used OR Tools library by Google. We collected some real world data in the same structure of the 3D-BPP data experiments. We have a tuple composed by the x,y,z dimensions of the item and the item weight.

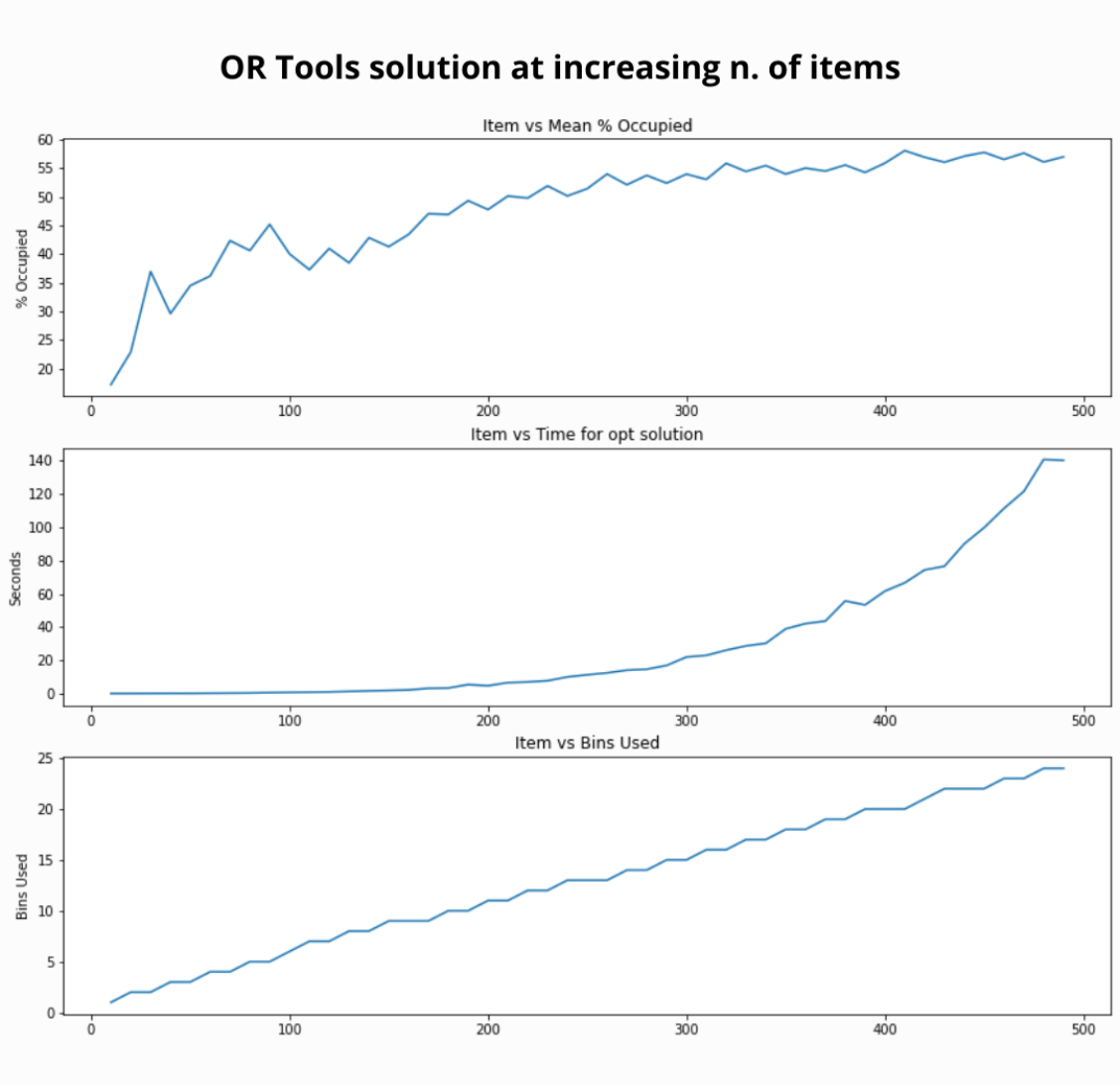


**Figure 5.7:** OR Tools solution at increasing number of instances. We took into account % of bin occupied, time for computing the solution and number of total bins used.

We encoded the constraints and instances in OR Tools library grammar. Than we created some custom functions for looping over different sets of items for inspecting the following metrics:

- Mean percentage of the single bin occupied;
- Time for finding the solution;
- Number of bins used.

It's interesting, as we can see in Figure 5.7, that those metrics have a completely different behaviour increasing the number of bins to pack. The mean percentage of bin occupied follows a curve that tends to stabilize at 60% of bin occupation. The number of bins used is linear with respect to the number of items to pack instead. The most important thing remains the time. In fact the whole point of this thesis work is the exponential time growth in solving BPP with classical MILP solvers.

As we can see in the image, in a real world scenario with thousands of items, it's hard to find solution with classical SCIP solvers. For packing 500 items it took 140 seconds. If we increase the number of items we can spend several hours for getting a solution. This is the reason why other solutions as the ones mentioned in this thesis are fundamental if we want to scale the dimension of items we want to pack. In a business context having to wait hours for the solution is something that is not a possibility. For this reason the idea of the PCT, that takes one item at time, gives better results.

In the next subsection we will finally inspect some practical solutions to the exponential time complexity. We will show some results from our experiments, and also from literature ones. Including some literature results is necessary because some results are not reachable without custom code and expensive hardware.

### 5.2.2 3D-BPP results

In this subsection we will show some results of 3D-BPP packing. In the first place we show some results from our simple experiments on data. Those are the experiments we were able to perform with the scripts provided by the paper repository. Than we will discuss some results from the paper, for having a more wide view on how those 3D-BPP perform if compared to classical MILP solvers.

Before describing the graphs we need to make a distinction between two terms that seems similar, but they are not. In fact, when training RL algorithms we have to refer to episodes, instead of classical learning epochs. One epoch, in neural network terms, is one forward pass and one backward pass of all the training examples. One episode is a sequence of RL states, actions and rewards which ends with terminal state. For example, playing an entire card game can be considered as one episode, the terminal state being reached when one player loses/wins/draws. This is important because the RL model learns only once the game is finished, ad a reward is given.

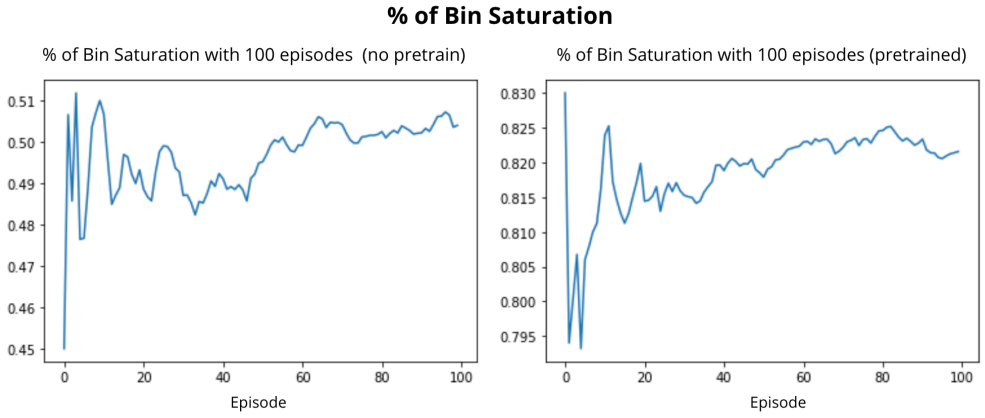


**Figure 5.8:** % of bins saturated over episodes, with pre-training and without pre training.

As we can see from the Figure 5.8 we performed the experiment twice. The first time, on the left, without using pre-trained weights. For this reason the re-

sults in terms of % of bin saturation are not so good, arriving at $50\%$ at episode number $100$. The things change when starting from pre-trained weights. In fact we reach a higher percentage of bin saturation of $83\%$. The most interesting thing of this method is that, after the RL training, the time spent for packing the items with PCT technique is linear.

Talking about paper result, the 3D-BPP method surpasses all other 3D-BPP algorithms. The surprising thing is that, due to the PCT implementation, is the first learning-based method that solves 3D-BPP with continuous solution space. They tried also to incorporate various practical constraints.

In the paper experiments $3$ different settings are used, for both discrete and continuous solution space:

- Setting 1: The stability of the $B_t$ is checked when $n_t$ is placed. Only two horizontal orientations (|O| = 2);
- Setting 2: The arbitrary orientation (|O| = 6) is allowed here. This is the most common setting in 3D-BPP literature;
- Setting 3: Each item $n_t$ has an additional density property sampled from $(0, 1]$ uniformly. This information is appended into the descriptors of $B_t$ and $n_t$.

The three existing schemes are the ones which have proven to be both efficient and effective: Corner Point (CP), Extreme Point (EP), and Empty Maximal Space (EMS). They extended these classical schemes to the PCT model.

We can see in Figure 5.9 that although PCT grows under the guidance of heuristics, the combinations of PCT with EMS and EV learn effective policies and outperform all baselines by a large margin regarding all settings. It is interesting to notice that policies guided by EMS and EV even exceed the performance of the full coordinate space FC which is expected to be the optimal one. This result makes us notice that a good leaf node expansion scheme reduces the complexity

of the problem and helps DRL agents learn better performance, even if simplifies some aspects reducing variability.

| | Method | *Setting* 1 Uti. | Var. | Num. | Gap | *Setting* 2 Uti. | Var. | Num. | Gap | *Setting* 3 Uti. | Var. | Num. | Gap |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Heuristic | *Random* | 36.7% | 10.3 | 14.9 | 51.7% | 38.6% | 8.3 | 15.7 | 55.1% | 36.8% | 10.6 | 14.9 | 51.4% |
| | BR | 49.0% | 10.8 | 19.6 | 35.5% | 56.7% | 6.6 | 22.6 | 34.1% | 48.9% | 10.7 | 19.5 | 35.4% |
| | Ha et al. | 52.1% | 20.1 | 20.6 | 31.4% | 59.9% | 10.4 | 23.8 | 30.3% | 51.9% | 20.2 | 20.6 | 31.4% |
| | LSAH | 52.5% | 12.2 | 20.8 | 30.9% | 65.0% | 6.1 | 25.6 | 24.4% | 52.4% | 12.2 | 20.7 | 30.8% |
| | Wang & Hauser | 57.6% | 11.5 | 24.1 | 24.2% | 66.1% | 8.4 | 25.9 | 23.1% | 56.5% | 11.2 | 22.3 | 25.4% |
| | MACS | 57.7% | 10.5 | 22.6 | 24.1% | 50.8% | 8.8 | 20.1 | 40.9% | 57.7% | 10.6 | 22.6 | 23.8% |
| | DBL | 60.5% | 8.8 | 23.8 | 20.4% | 70.6% | 7.9 | 27.8 | 17.9% | 60.5% | 8.9 | 23.8 | 20.1% |
| Learning-based | Zhao et al. | 70.9% | 6.2 | 27.5 | 6.7% | 70.3% | 4.3 | 27.4 | 18.3% | 59.6% | 5.4 | 23.1 | 21.3% |
| | PCT & CP | 69.4% | 5.4 | 26.7 | 8.7% | 81.8% | 2.0 | 31.3 | 4.9% | 69.5% | 5.4 | 26.7 | 8.2% |
| | PCT & EP | 71.9% | 6.6 | 27.8 | 5.4% | 78.1% | 3.8 | 30.3 | 9.2% | 72.2% | 5.8 | 27.9 | 4.6% |
| | PCT & FC | 72.4% | 4.7 | 28.0 | 4.7% | 76.9% | 3.3 | 29.7 | 10.6% | 69.8% | 5.3 | 27.1 | 7.8% |
| | PCT & EMS | 75.8% | 4.4 | 29.3 | 0.3% | **86.0%** | **1.9** | **33.0** | **0.0%** | 75.5% | 4.7 | 29.2 | 0.3% |
| | PCT & EV | **76.0%** | **4.2** | **29.4** | **0.0%** | 85.3% | 2.1 | 32.8 | 0.8% | **75.7%** | **4.6** | **29.2** | **0.0%** |
| | PCT & EVF | 75.7% | 4.8 | 29.2 | 0.4% | 80.5% | 2.9 | 31.0 | 6.4% | 73.5% | 4.6 | 28.4 | 2.9% |
| | PCT & EV/GS | 75.8% | 4.7 | 29.2 | 0.3% | 84.8% | 2.1 | 32.6 | 1.4% | 75.5% | 4.8 | 29.1 | 0.3% |
| | *Random* & EV | 45.7% | 13.5 | 18.4 | 39.9% | 51.0% | 8.3 | 20.4 | 40.7% | 45.1% | 12.5 | 18.1 | 40.4% |

[1] https://github.com/alexfrom0815/Online-3D-BPP-PCT

**Figure 5.9:** Performance comparisons in a discrete solutions space.

They also performed a continuous domain experiment and this is very interesting from a future development point of view. They found that some heuristic methods also have the potential to work in the continuous domain. They set these heuristic methods as baselines for banchmarking. As we can see form the Figure 5.10, the 3D-BPP with PCT problem and outperforms the performance of all other methods again. This specific PCT work is very interesting because it seems to be the first that deploys the learning-based method on solving 3D-BPP with continuous solution space successfully.

In the final Figure 5.11 we can see how this 3D-BPP solution can be automated in a industry domain. Obviously this is a next level automation, but it's possible to implement in a real case scenario. After commenting those results, we can finally try to get some conclusions from this thesis work.

| | Method | *Setting* 1 Uti. | Var. | Num. | Gap | *Setting* 2 Uti. | Var. | Num. | Gap | *Setting* 3 Uti. | Var. | Num. | Gap |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Heu. | BR | 40.9% | 7.4 | 16.1 | 37.5% | 45.3% | 5.2 | 17.8 | 31.7% | 40.9% | 7.3 | 16.1 | 38.6% |
| | Ha et al. | 43.9% | 14.2 | 17.2 | 32.9% | 46.1% | 6.8 | 18.1 | 30.5% | 43.9% | 14.2 | 17.2 | 34.1% |
| | LSAH | 48.3% | 12.1 | 18.7 | 26.1% | 58.7% | 4.6 | 22.8 | 11.5% | 48.4% | 12.2 | 18.8 | 27.3% |
| DRL | GD | 5.6% | – | 2.2 | 91.4% | 7.5% | – | 2.9 | 88.7% | 5.2% | – | 2.1 | 92.2% |
| | PCT & EMS | 65.3% | 4.4 | 24.9 | 0.2% | **66.3%** | **2.3** | **27.0** | **0.0%** | **66.6%** | **3.3** | **25.3** | **0.0%** |
| | PCT & EV | **65.4%** | **3.3** | **25.0** | **0.0%** | 65.0% | 2.6 | 26.4 | 2.0% | 65.8% | 3.6 | 25.1 | 2.7% |

**Figure 5.10:** Performance comparisons in a continuous domain.

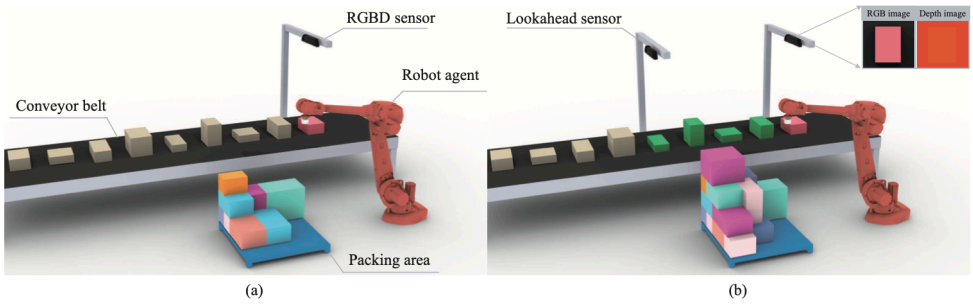


**Figure 5.11:** Online 3D-BPP has widely practical applications in logistics, manufacture, warehousing and other fields. This is a visualization from the additional materials of original paper.

# 6 Conclusion

In this master thesis I focused on some cutting edge research topics. Those can be also useful in a real business scenario, if engineered with a proper R&D team or using third-part solutions that are not free to access. We proceed by giving some final thoughts about the value this thesis work can provide.

Using mathematical techniques and the modern hardware tools, a lot of time and inefficiency costs can be cut in industry. This will be one of the challenges of the next decades, even if it's not as mind blowing as neural networks painting pictures or applications like that.

We can see this research work as a survey for starting to inspect different approaches for tackling BPP or MILP problems in general through ML techniques. With the aim that OR and ML experts will start making more and more cooperation for solving the issues about practical applications and benchmarking of algorithms. For having a fast growing development on the field, especially in open source projects. We hope that will be created open source libraries for making experiments more accessible in the future.

The topics covered, as GCNN or RL framework, are very valuable from a scientific point of view. I hope this work is able to simplify those concepts and give some practical ideas of application, which are very fascinating.

The experiments give better performances in terms of time spent if compared to classical solvers. Those inspected developments can improve the companies shipping software making it more scalable and well-performing. A practical idea for the business side is to invest in some non-free software, making the improvements possible in the short term.

In conclusion, leaving aside the business purposes and focusing on the research questions, I hope some future students or researchers will be able to inspect deeply those topics. This thesis work can be seen as a starting point for making a further step in the Operational Research with Machine Learning field.

# References


[1] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," *European Journal of Operational Research*, vol. 290, no. 2, pp. 405–421, 2021.

[2] H. Larnder, "Or forum—the origin of operational research," *Operations Research*, vol. 32, no. 2, pp. 465–476, 1984.

[3] N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev, "Reinforcement learning for combinatorial optimization: A survey," *Computers & Operations Research*, vol. 134, p. 105400, 2021.

[4] R. E. Korf, "A new algorithm for optimal bin packing," in *Aaai/Iaai*, 2002, pp. 731–736.

[5] S. Martello and P. Toth, "Bin-packing problem," *Knapsack problems: Algorithms and computer implementations*, pp. 221–245, 1990.

[6] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell, "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning," *Discrete Optimization*, vol. 19, pp. 79–102, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1572528616000062

[7] S. I. Gass and C. M. Harris, "Encyclopedia of operations research and management science," *Journal of the Operational Research Society*, vol. 48, no. 7, pp. 759–760, 1997.

[8] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi, "Exact combinatorial optimization with graph convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[9] H. Hu, X. Zhang, X. Yan, L. Wang, and Y. Xu, "Solving a new 3d bin packing problem with deep reinforcement learning method," *arXiv preprint arXiv:1708.05930*, 2017.

[10] H. Zhao, Y. Yu, and K. Xu, "Learning efficient online 3d bin packing on packing configuration trees," in *International Conference on Learning Representations*, 2021.

[11] S. Martello, D. Pisinger, and D. Vigo, "The three-dimensional bin packing problem," *Operations research*, vol. 48, no. 2, pp. 256–267, 2000.

[12] S. Martello, M. Monaci, and D. Vigo, "An exact approach to the strip-packing problem," *INFORMS journal on Computing*, vol. 15, no. 3, pp. 310–319, 2003.

[13] T. G. Crainic, G. Perboli, and R. Tadei, "Extreme point-based heuristics for three-dimensional bin packing," *Informs Journal on computing*, vol. 20, no. 3, pp. 368–384, 2008.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[15] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," *Advances in neural information processing systems*, vol. 28, 2015.

[16] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," *Advances in neural information processing systems*, vol. 30, 2017.

[17] A. Prouvost, J. Dumouchelle, L. Scavuzzo, M. Gasse, D. Chételat, and A. Lodi, "Ecole: A gym-like library for machine learning in combinatorial optimization solvers," *arXiv preprint arXiv:2011.06069*, 2020.