Spring 2024

BIN PACKING PROBLEM - AN EFFICIENCY ANALYSIS BETWEEN COMMERCIAL CONTAINER LOADING AND OPEN-SOURCE OPTIMIZATION ALGORITHMS

Gabriel Mihu
University of New Hampshire

BIN PACKING PROBLEM - AN EFFICIENCY ANALYSIS BETWEEN COMMERCIAL CONTAINER LOADING AND OPEN-SOURCE OPTIMIZATION ALGORITHMS

by

Gabriel Mihu

MFA, University of Art and Design, Cluj-Napoca, Romania, 2006

BS, Technical University of Cluj-Napoca, Romania, 2004

THESIS

Submitted to the University of New Hampshire

In Partial Fulfillment of

The Requirements for the Degree of

Master of Science

In

Information Technology

May, 2024

This thesis has been examined and approved in partial fulfillment of the requirements for the degree of Master of Science in Information Technology by:

Thesis Director, Michael Jonas
Associate Professor, Applied Engineering & Sciences

Ken Gitlitz, Senior Lecturer, Computer Science

Timothy Finan, Lecturer, Applied Engineering & Sciences

On May 5, 2024

Original approval signatures are on file with the University of New Hampshire Graduate School.

ABSTRACT

BIN PACKING PROBLEM - AN EFFICIENCY ANALYSIS BETWEEN COMMERCIAL CONTAINER LOADING AND OPEN-SOURCE OPTIMIZATION ALGORITHMS

by

Gabriel Mihu

University of New Hampshire, May 2024

Bin packing problem literally refers to bin packing: optimize fitting of a number of smaller containers into bins of equal volume, in order to occupy the smallest number of bins. It has various real-world applications, from stock cutting, container shipping to 3D transistor microprocessor design and computer files packing. Finding an optimal solution can easily become a very challenging task as the number of elements increases.

Single algorithm application can be very effective in certain cases, but usually multiple algorithms are used in conjunction could be more effective for increased efficiency and for what is considered a highly optimized result. Developing a high-quality algorithm combination can be a challenging task. Companies keep their successful solutions protected as proprietary information, but there are open-source algorithms that can be used. This thesis presents a comparison and analysis between available commercial and open-source algorithms.

ACKNOWLEDGEMENTS

I hereby acknowledge and thank my thesis advisor, Dr. Michael Jonas, who patiently supported and guided this research and my efforts during the process. I would like to thank my committee members: Prof. Ken Gitlitz and Prof. Timothy Finan for their interest and expertise. I want to thank to Dr. Mihaela Sabin and Assistant Director for Student Affairs Candice Morey, as from different perspectives they helped me find my bearing, encouraged me throughout my academic journey and helped achieve my goals.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

According to the Merriam-Webster dictionary, optimization represents the process that leads to making a system/design "as fully perfect, functional, or effective as possible" [1]. While maintaining its original meaning, optimization can have different connotations based on association to a different domain (marketing optimization, website optimization, business optimization, etc.). Not every optimization process can be simply reduced to solving mathematical equations – different domains require different approaches, methods and principles in order to achieve the desired results. Still, there are a multitude of problems, from various domains, that can be represented by similar mathematical models, solved by a common set of methods, as Stephen J. Wright points out in his Encyclopedia Britannica article about optimization [2].

Everyday life challenges like stock cutting, bin/container packing/loading or the knapsack problem can be solved to a more or less optimum solution by applying a set of algorithms to a mathematical representation of the initial problem. The solving process requires abstraction of one or more item dimensions, such as weight, length, surface, volume, orientation, etc. to create a mathematical model for successful resolution.

1.1 Drawbacks of Brute Force Approach

Applying a brute force algorithm could provide a complete set of solutions to any problem, but it is not a viable problem-solving technique the eight queens puzzle problem illustrates. On a chessboard, 8 queens need to be positioned so they don't attack each other. In

this case, there are 92 possible solutions, with 4,426,165,368 possible placements. With the computing power of today's processors, a brute force algorithm would be able to find a solution in a fraction of a second. Things get complicated if, for example, we increase the number of queens to 64, and the board size to 64 x 64 = 4096 squares. In this case, the possible number of placements is approx. 7,46 x $10^{\wedge 141}$, which would render practically impossible generating and testing every possible combination in a timely manner.

This is the reason why one of the most important steps towards solving such a problem is represented by finding and applying sets of constraints that drastically decrease the number of possible combinations – example: place a queen on each row reduces the number of possible placements from 4,426,165,368 to 16,777,216. Based on the research done by a group from University of St Andrews, UK, "the counting version of the problem, i.e., to determine how many solutions to n-Queens there are, is sequence A000170 of the Online Encyclopedia of Integer Sequences (Sloane, 2016). The sequence is currently known only to n = 27, for which the number of solutions is more than $2.34 \times 10^{\wedge 17}$" [3] [4] [5]. According to N. J. A. Sloane, in the same article referenced in the above citation, the computation for the 25 queen project, the "has been confirmed by the NTU 25Queen Project at National Taiwan University and Ming Chuan University, led by Yuh-Pyng (Arping) Shieh, Jul 26 2005. This computation took 26613 days CPU time." [4]. Even if "days CPU time" is not an agreed upon measurement unit, and it can highly depend on the testing CPU processing power, the approximately 73 years puts the resource intensiveness of a brute force approach into perspective.

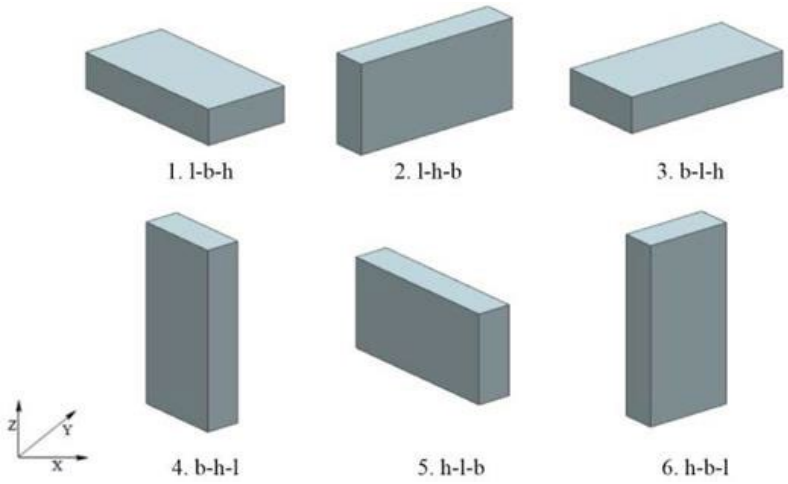

Figure 1.1 – Possible orientations for a loaded rectangular item [6]

The complexity of a brute force approach in relation to container loading can be exemplified by a 1000 items container load. Considering each item can have 6 possible orientations (Figure 1.1), the total number of possible combinations is approximately $1.42x10^{\wedge 778}$, which is approximately $10^{\wedge 5}$ the amount of possible combinations for a n=64 queens puzzle problem, or approximately $10^{\wedge 45}$ the amount of combinations for n=27 of the same problem. Brute force algorithm applications, while offering a complete set of solutions, often suffer from inefficiency and high computational resource consumption, resulting in slower performance compared to optimized methods. Facing scalability issues and becoming increasingly resource-consuming, as problem complexity increases, they fail to deliver the most efficient solutions.

1.2 Bin Packing

The bin packing problem has its origins in everyday life, with one of its variations trying to determine a minimal size container that would completely fit a specific number of objects. The variant this thesis is focused on is represented by the attempt to find the best arrangement for a

set of objects to maximize the volume occupancy of a fixed size container. From an economical perspective the most obvious benefit is reduced shipping cos by reducing container cost per item, and by reducing the number of total containers, inherently loading/unloading time and total cost for ground container transportation. Consider the scenario of shipping 100,000 items. Without optimization, a shipping container can fit 1000 identical items, at 85% occupancy. Just by changing orientation on some items, the container can be filled up to 95% occupancy, with 1117 identical items. Assuming a container shipping cost of $4000 (revised as of 2024), this adjustment translates to a reduction of shipping cost per item of $0.42, corresponding to a reduction $42,000 in total shipping costs by sea. Considering shipping each container port to location for 500 miles, at a rate of $2.5 per mile [7] [8], the shipping cost savings are $13,100, adding to a total of $55,100.

This cumulates with the reduced wait time incurred by the reduced number of containers and the benefits presented above can be achieved by applying or removing constraints during the mathematical analysis of the container load. Such constraints can be: allow item orientation, repositioning, grouping, packing one on top of the other in multiple layers, etc. – increasing the container occupancy rate. Imposing container related constraints can also address packaging requirements aligned with safety standards and regulations. This includes even distribution of items within the container to achieve a balanced load and maintain a low center of gravity, according to the container packing recommendation brochure of Hapag-Lloyd, one of the global container shipping leaders [9].

1.3 Problem Statements

The problem statements explored by this thesis are aimed at analyzing the comparative efficiency between commercial container loading and open-source optimization algorithms in the

context of the bin packing problem. Focusing on key factors such as volume occupancy, result consistency, adaptability to diverse datasets, computational speed, and the number of iterations required, the following problem statements can be formulated:

1. **Problem 1 – Volume Occupancy:** how do commercial container loading and open-source optimization algorithms compare in relation to maximizing the volume occupancy in container loading scenarios?
2. **Problem 2 – Algorithm Efficiency:** which algorithmic approach achieves faster results in reaching a volume occupancy threshold, and how does the number of algorithm iterations impact this achievement?
3. **Problem 3 – Consistency**: what variations in results and performance consistency are observed between commercial and open-source algorithms when applied to a limited input dataset?

1.4 Goal

The core objective of this thesis is a comparative analysis that allows achieving results and drawing meaningful conclusions towards determining algorithm(s) effectiveness for specific scenarios. By doing so, this study aims to address the formulated problem statements, achieve advancements in the understanding and application of bin packing algorithms, to enhance their practical utility in container loading applications. Currently there is a lack of conducted research that compares the efficiency of various container loading applications, while articles such as "Intelligent 3D packing using a grouping algorithm for automotive container engineering" by Youn-Kyoung Joung and Sang Do No [6], or "Smart Packing Simulator for 3D Packing Problem Using Genetic Algorithm" by U. Khairuddin, N. A. Z. M. Razi, M. S. Z. Abidin and R. Yusof [10] touch on certain practical aspects related to 3D bin packing. The challenge of this study is

developing an appropriate framework that supports and makes such an analysis and comparison possible. This involves several key aspects: selecting an appropriate data set, implementing wrappers for open-source solvers, analyzing and interpreting the results to respond to the problem statements mentioned above.

1.5 Implementation

The necessity of this study resulted as a consequence of a quick evaluation request of various container loading optimization software applications. While addressing the comparison request on an assumed unchallenging data set, the tested applications returned varied results, with volume occupancies ranging from 80% to 95%. For some applications the variance in occupancy results between attempts was notable within the same software under identical conditions, with one application in particular returning 95% for the first attempt, and only 85% during more than 30 subsequent attempts (as shown in Table 2.2). The initial findings, which are detailed further in the background section of this thesis, triggered the exploration of possible open-source applications relevant to the bin packing problem and the comparison of the results to their commercial counterparts. Section 2.2 reveals that this effort returned mixed outcomes, emphasizing the limitations of the open-source ready-to-use applications created for the purpose of this analysis.

Further, the lack of suitable open-source applications triggered the open-source algorithm investigation in relation to container loading, with a focus on the selected OR-Tools package. Although multiple tests have been conducted based on both knapsack and linear solver form the Or-Tools package, the resulting script during the analysis phase was developed in Python and utilizes OR-Tools linear solver. The testing has been conducted under similar conditions and environment as the commercial software applications. As part of the implementation effort,

specific data sets have been selected to challenge the algorithms in specific scenarios. Given the significant time gap between the initial application assessment and the open-source algorithm testing phase of the research, the latest releases of commercial applications have been retested with the aforementioned selected specific data sets. This approach ensured that the comparison between recently tested open-source algorithms and commercial applications was conducted under current conditions, with the latest developments and updates, providing an accurate reflection of the capabilities and improvements in both types of solutions. Ultimately, the analysis of the test results led to a set of conclusions and further development opportunities.

1.6 Structure

The thesis structure is roughly mapped to the section 1.5 above, and begins with the chapter Preliminary Analysis And Algorithm Selection – an exploration of the original evaluation of both types of ready-to-use software applications (container loading commercial and bin packing open-source). The core of the research is represented by the chapter Evaluation Of Applications And Open-Source Algorithms, that encompasses most of the implementation strategy for this study. From data sets and open-source algorithm selection to scripts implementation and partial results, most of the testing phase content is included in this chapter. The final representation of the overall significant results, together with their interpretation and analysis can be found in the chapter Analysis. The final chapter, Conclusions And Future Developments, presents a set of summarizing conclusions based on the analysis findings, together with recommendations for further development.

2 PRELIMINARY ANALYSIS AND ALGORITHM SELECTION

The presence of open source algorithms, that could constitute the base for an open source load optimization application, can be acknowledged after few minutes of investigation. Algorithms such as VPSolver, or packages such as OptaPlanner or OR-Tools are viable options for container loading applications, unfortunately research did not reveal any open-source applications implementing them successfully for the scope of this study. As a consequence, for the preliminary phase of this investigation, in terms of usable software, the initial focus was towards commercial applications.

2.1 Commercial applications

Commercial applications attempt to solve packing or related problems in a more or less successful way. This section of the research focuses on preliminary algorithm results. Close to 10 applications have been tested during this initial phase of the investigation, some of them being discarded during the process and from the final comparison, due to lack of features that would allow obtaining pertinent results, or due to extremely inefficient results based on the sample data set. As a result, the following list of application was left as suitable candidates: Cargowiz, CubeMaster, EasyCargo, PackVol, CubeIQ, MaxloadPro.

2.1.1 Preliminary testing and results

The test data was represented by a real world scenario, named Data Set Preliminary 1 (DSP1), of a 40-foot shipping container as the “bin”, and different types of rectangular shape

boxes for occupancy (Table 2.1). The total volume of the boxes to be fit in the container is slightly larger than the volume of the container, thus allowing flexibility for the testing algorithms to assess and determine various counts of the two box types as an optimal volumetric fit solution.

Item	Quantity	Dimensions	Orientations allowed
Container	1	1180 x 230 x 265	x
Box1	550	37 x 37 x 52.5	6
Box2	290	41 x 41 x 59	6

Table 2.1 – Data Set Preliminary 1 (DSP1) – preliminary commercial applications test data

The applications employed a series of options or stopping criteria that support customizing the optimization process to a certain extent, with the intent of improving the final results. Some of the available stop criteria are: maximum processing time, maximum number of total iterations, maximum number of non-improving iterations, different algorithm combinations, etc. Multiple tests have been conducted, with different options selected, aiming to maximize container load occupancy. Table 2.2 displays in summary the best results that could be reproduced multiple times with the available applications selected at the date of the testing.

Software package	**Container load occupancy (best repeatable result)**
Cargowiz	92%
CubeMaster	94%
EasyCargo	80-85%
PackVol	95%
CubeIQ	95%
MaxloadPro	95% / 85%

Table 2.2 – Commercial applications test results

Based on the test results presented above and other test parameters and observations (not presented in detail in this thesis) the following preliminary observations can be noted:

a. **results consistency** – the best fill factor for this test data set is 95%, with two applications consistently returning this result, two others following closely (94%, 92%); two applications were not able to get anywhere close to the 95% occupancy factor, with consistently variable results.
b. **computation time** – whether we use time or number of tries as a stop criterion, some applications spent the same amount of time to conclude the optimization process. In contrast, if using the number of passes as limiting factor, some applications spent a variable amount of time to run the same number of passes, thus implying the use of different combination of computational paths to achieve the results.
c. **number of passes** – if the number of passes is not set as a stopping criterion before running the application on a data set, the software will decide based on different other criteria before ceasing to search for a better result. Based on observations, we can have simple stopping criteria such as time, set by the user, or more complex stopping criteria, such as result improvement (for example: if the last 5 results are all within 0.1%, then the best of them will be selected and returned as best overall result).

Figure 2.1 represents a sample from the preliminary analysis results, based on the data set DSP1. This example illustrates how the algorithm addressed box positioning and grouping inside the container, maintaining the two types of boxes in two separate groups. It is also obvious that certain boxes have different orientations compared to others, the algorithm displaying the use of box rotation to optimize the results.

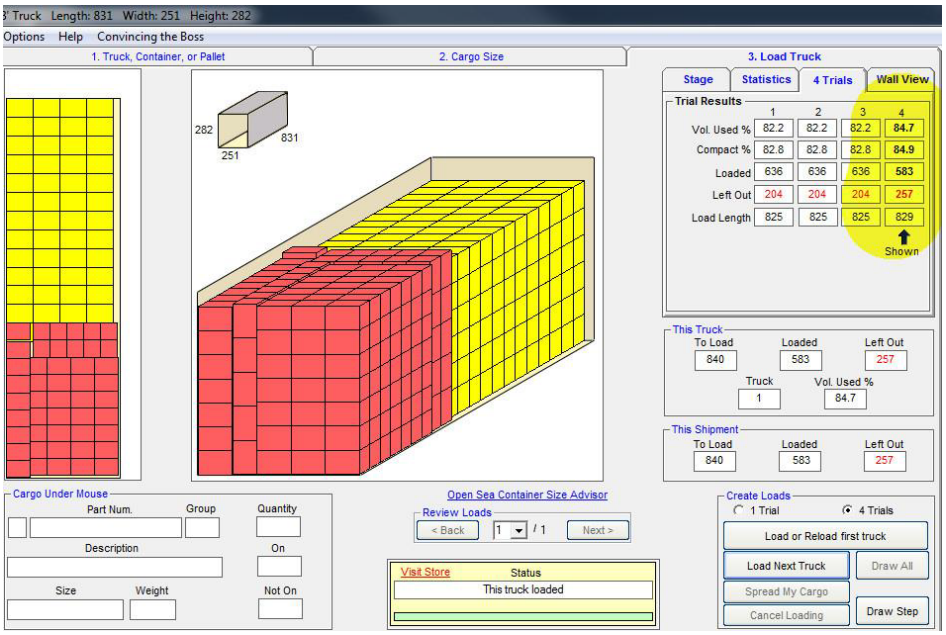

Figure 2.1 – Example of Data Set 0 container 3D layout – Cargowiz

The same data set has been used for the representative image in Figure 2.2, the 3D visualization section displaying the difference between the two applications in positioning the boxes inside the container: observe different grouping and orientation. From the two images, the red and green boxes represent Box 1, while yellow and purple represent Box 2 from the data set.

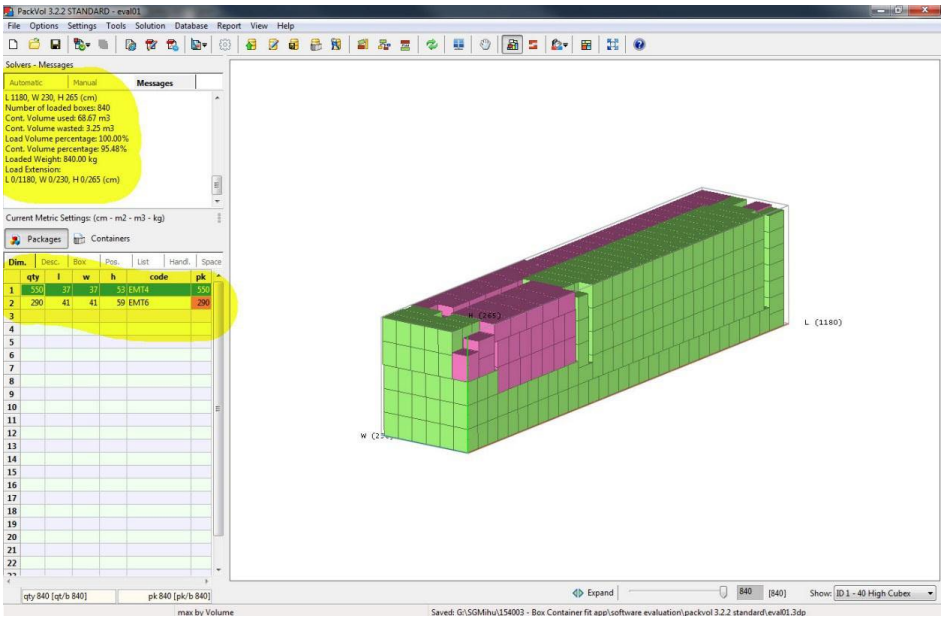

Figure 2.2 – Example of Data Set 0 container 3D layout - Packvol

2.1.2 Preliminary considerations

Based on the test results observations from section 2.1.1 a set of preliminary considerations and hypothesis can be determined. They are validated or invalidated by further testing conducted in chapter 3. Repeated testing and results support the hypothesis that some applications use a “static” determination sequence towards applying different algorithms. Based on the same set of data, with the same imposed constraints, they adopt the same sequence of algorithms, in the same order, in the attempt to solve the optimization problem. Other applications apply a certain level of randomness in selecting the appropriate algorithms or starting point box. Although there might be a certain degree of basic algorithm selection randomization, there is no evidence to support algorithm selection based on a random number generator and a set of pre-determining algorithms, rather a simple attempt to determine the best algorithm to be used or using the only algorithm available. After the first pass, the result is assessed, the algorithm selection reviewed, and the next result computation follows. For each

pass, it is possible a new algorithm or configuration is selected from a internally predetermined list of available algorithms. It is also plausible to hypothesize that the applications use a single solving algorithm, and the variance in test results is dictated by random starting point (box) or other variables impossible to determine from a user perspective. The following paragraphs have been constructed based on the assumption that the applications do use multiple algorithms.

It is plausible that applications that obtain the best results are the ones that apply a certain degree of randomness, this allowing the test of different algorithms on a data set, leading to the selection of the best algorithm. These applications returned the most inconsistent results. The applications that seem more static in algorithm sequence selection actually returned the best results in terms of consistency. This might be also due to the fact that the latter set of applications have a very good decision path implemented, and they don't need to try to adapt the used algorithm in order to achieve the best results. At the moment of the preliminary research, above-mentioned conclusions were all plausible. Later in this study, some of them were confirmed, others denied by the more in-depth analysis.

2.2 Open-source algorithms and preliminary selection

An open-source algorithm search led to Wikipedia's "bin packing problem" page [11] and from that page to a "list of optimization software" [12], which presented a set of possible open source and/or free applications/algorithms to be used for retrieving an optimal solution to the problem. After screening the available options presented, together with investigating other resources, the following list has been constructed:

- **BPP Spreadsheet Solver** – two-dimensional bin packing problem solver (Microsoft Excel and VBA)

- **OpenOpt** – free language framework (Python)
- **OptaPlanner/Timefold** – open source Java constraint solver (Java)
- **OR-tools** – open source optimization software package (libraries or source code native in C++, available for Python, C#, Java)
- **VPSolver** – open source – solver for one-dimensional cutting and packing problems (libraries for both Python/C++)

Implementing an application for all the open source algorithms is beyond the scope of this project, a preliminary algorithm selection was warranted at this point, based on existing information related to parameters such as: efficiency, adaptability, availability, speed. The lack of specific documentation pertinent to this study provided to be a challenge for some of the algorithms. Descriptions about their best application, their best fit for a certain set of data and use case proved difficult to find. Depending on the situation, the selection process was based on prior existing documented assessments or studies, documentation details and/or empirical preliminary testing.

2.2.1 BPP Spreadsheet Solver

Bin Packing Problem Spreadsheet Solver (BPP Spreadsheet Solver) is an open source Microsoft Excel worksheet that can solve two-dimensional bin packing problems, allowing also multiple item types, bin types and partial constraints or stop criteria. BPP Spreadsheet Solver has been developed by Dr. Gunes Erdogan, at University of Bath, UK, in addition to two other spreadsheet optimization problem solvers developed by the same author [13]. This solver is presented as an Excel spreadsheet with an Add-in menu, backed up by a set of VBA scripts. The

spreadsheet allows the user to customize bins (up to 10) and items (up to 100), presenting certain limitations [13]:

- two-dimensional
- no safety distance between items in a bin
- all items assumed to be rectangles
- no prepositioning of items in a bin
- no constraints other than no-overlap

Although upon initial assessment the constraints don't seem to be limiting in relation to the purpose of this thesis, the lack of 3D optimization, the existence of hardcoded sheet names and absolute cell references make this algorithm inflexible for implementation and future scalability. First data set tested with BPP Spreadsheet Solver is presented below, in Table 2.3. The first result was that there are not enough bins to fit all the items, and after further analysis it was determined that the number of bins must be entered prior to applying the algorithm. The number of bins is not automatically adjusted to fit all the objects.

Item	**Data**
Objects (dimensions)	(3, 3), (4, 5), (3, 7), (8, 7), (7, 2), (2, 3)
Bin dimensions	(7, 14), (7, 14)
Test duration (Seconds)	60
Iterations	1157395
Fit count/occupancy	All/100%

Table 2.3 – Data Set Preliminary 2 (DSP2) – 2D data set and results

Increasing the complexity of the test data set as presented in Table 2.4 resulted in a significant performance decrease, with the algorithm performing only 2200 iterations during 60

seconds of computation. The occupancy rate also decreased to 84.5% area of container occupancy. As a reminder, this is a two-dimensional bin packing solver application.

Item	**Data**
Objects (qty, (dimensions))	(1000, (7, 2)), (1000, (2, 3))
Bin dimensions	(100, 100)
Test duration (seconds)	60
Iterations	2200
Fit count/occupancy	98/84.5%

Table 2.4 – Data Set Preliminary 3 (DSP3) – 2D data set

BPP Spreadsheet Solver is a lightweight, ready to use algorithm, for 2D objects fitting in a 2D container. Its limitations, however (i.e. hard coded Excel cell indexes, low number of items/item types and bins/bin types), make it more suitable for example visualization in a teaching environment, not for inclusion in a full scale working application intended for large scale utilization. Due to its limitations and inflexibility, this solver application was not a suitable candidate as an open source algorithm for further evaluation.

2.2.2 OpenOpt

OpenOpt is another framework known for solving optimization problems. Developed as cross-platform Python language modules, it constitutes a framework of "universal numerical optimization [..] with several own solvers [..] and connections to tens of others, graphical output of convergence" [14] and other benefits. Based on these observations, at the starting time of this research (2015), the framework appeared as a viable option for further testing. However, at the time the final algorithm selection (2023/2024), the last repository update for this framework on GitHub was 12 years old. The package is also available for install from pypi.org, with the latest

release on December 23rd, 2018. Based on more recent review of implementation resources and usability assessment, the framework appears to be lacking meaningful documentation in comparison with other options such as OptaPlanner and OR-Tools. There is a possibility that the OpenOpt set of algorithms is so efficient and robust, that the necessity for updates was not needed over this period of time, and it would still be a good candidate for testing and analysis. However, this is uncertain, and comparing it overall with the other available options, this OpenOpt presents as less viable for implementation purposes, and therefore not selected for further use in this thesis.

2.2.3 OptaPlanner/Timefold

OptaPlanner is a 100% Java written embeddable planning engine, which combines multiple algorithms, supposedly leading to very satisfactory results in satisfying multiple constraints. It is backed up by a well written documentation page, with solid explanations related to possible use cases, configuration, integration opportunities, and detailed optimization algorithms description. Some of the applications include handling complex logistics, such as vehicle routing, employee rostering, task allocation, where multiple constraints need to be managed simultaneously. OptaPlanner "combines sophisticated Artificial Intelligence optimization algorithms (such as Tabu Search, Simulated Annealing, Late Acceptance and other metaheuristics)" [15]. Since May 2023, the OptaPlanner project is available also under the name of Timefold [16] as open-source and also as a commercial option. Throughout this thesis we referred to this set of algorithms, and will continue to do so, as OptaPlanner. According to the OptaPlanner User Guide, "OptaPlanner is a lightweight, embeddable constraint satisfaction engine which optimizes planning problems." [17]. Some of the use cases mentioned are: shift rostering, vehicle routing, agenda scheduling, financial optimization, bin packing. Although bin

packing appears to be an applicable use case for this framework, research did not find implementations or examples to support this statement.

2.2.4 OR-Tools

OR-Tools is primarily a collection of algorithms written in native C++, as detailed on its installation webpage. The page states: "Google created OR-Tools in C++, but you can also use it with Python, Java, or C# (on the .NET platform)" [18]. The toolkit includes support for several programming languages, including C++ and Python. Given the vast number of possible solutions associated with optimization problems, “OR-Tools uses state-of-the-art algorithms to narrow down the search set, in order to find an optimal (or close to optimal) solution.” [19]. The documentation page presents a direct example of the bin packing problem, using Glop, “Google's in-house linear programming solver”, according to the “Solving an LP Problem” [20] page, which together with “Advanced LP Solving” [21] discuss about using Glop as a linear solver and also present a list with other solvers available in OR-Tools (Clp, CPLEX, GLPK, Gurobi, PDLP, XpresS), with some of them requiring a commercial license. The advanced page dives into a slightly more detailed analysis of the available solvers, with a final recommendation to “Try Glop” as an open-source and trusted for Google’s production workloads, and if a license is available, suggests using a commercial solver (CPLEX, Gurobi or Xpress) [as] these solvers are industry standard and highly optimized. Since this thesis has a stated goal to compare the commercial container loading applications to open-source algorithms, the first recommendation is the pertinent one in this context. The apparent well-documented nature, existing examples, wide and active community support, suggest this set of algorithms as being a suitable candidate for further testing and development.

2.2.5 VPSolver

VPSolver is very well documented, with easily accessible and available resources. It appears to be very well developed, with strong capabilities towards solving vector packing or multiple choice vector packing problems. Vector packing represents n items of m different types packed in bins of the same size. Multiple choice vector bin packing problem allows several types of bins, with items capable of having one of multiple sizes, as described by F. Brandao, the author of the algorithm [22]. The algorithm is presented very well, as being compiled and used on Linux, Mac OS X and Windows, making it very well suited for implementation. This algorithm and its implementation was developed at University of Porto, Portugal, by Filipe Brandao and Joao Pedro Pedroso, and it is continuously updated and adapted, under different flavors/variations, to fit specific bin packing based problems. A detailed description can be found in the study "Bin Packing and Related Problems: General Arc-flow Formulation with Graph Compression" [23], with a summary in the poster "Arc-flow Formulation for Vector Packing" [24].

The p-dimensional vector packing problem, also called general assignment problem, is a generalization of bin packing with multiple constraints. In this problem, we are required to pack n items of m different types, represented by p-dimensional vectors, into as few bins as possible. In practice, this problem models, for example, static resource allocation problems where the minimum number of servers with known capacities is used to satisfy a set of services with known demands. "The multiple-choice vector bin packing problem is a variant of the vector packing problem in which bins have several types (i.e., sizes and costs) and items have several incarnations (i.e., will take one of several possible sizes). [23]

2.2.6 Preliminary selection

Based on the above analysis, OR-Tools, OptaPlanner and VPSolver appear to be the most suitable candidates for the research section of this thesis. With up-to-date repositories and detailed documentation, they are strong candidates for implementation. The focus of this study is a comparative analysis between commercial applications and open source optimization algorithms. In order to achieve this goal, time and dedication need to be focused on testing and results analysis. In a comparison between OR-Tools and OptaPlanner, Jeroen Colin presents a couple of industry cases on which two solving methods from the two frameworks are applied [27]. The results weigh in favor of OptaPlanner, however the solver used for OR-Tools is CP-SAT, a constraint programming solver, different than the linear solvers recommended for bin packing. Considering the above, together with the specific algorithms details presented in the previous sections of this chapter, the option selected for further development is the OR-Tools package. Although it did pose some challenges (see section 3.1.1), it still presented the best choice in terms of development for the purpose of this study.

3 EVALUATION OF APPLICATIONS AND OPEN-SOURCE ALGORITHMS

The objective and conclusions of this thesis are highly supported by test data and result comparison and analysis. Taking this aspect under consideration, the integrity and credibility of this study are significantly reinforced by the employment of specific test systems, rather exclusively one system over multiple alternatives. Employing a single system ensures a controlled environment that supports results repeatability and eliminates possible concerns for variability that could arise from test system variations. Employing multiple systems without standardization could compromise the uniformity of the collected data. Furthermore, the increased complexity and unpredictability of managing multiple systems can lead to discrepancies in data interpretation and result comparison. The specific details of the initial system used for preliminary testing of paid applications (section 2.1.1) are not documented in this study. However, note that all the initial tests related to these applications were uniformly conducted on the same system, and the results were used as an initial benchmark on the variance of the application capabilities. Subsequently, the majority of the testing was conducted on the system with specifications outlined in Table 3.1. The table details the technical specifications of the primary hardware and software used, thus ensuring a controlled environment.

Item	Value
Operating system	Microsoft Windows 10 Pro
System type	x64
Processor	AMD Ryzen 7 5800X, 3.8MHz
Memory	96GB DDR4, 2666
Motherboard	AsusTek TUF Gaming X570-Pro
System drive	Crucial P5 Plus CT2000P5PSSD8, PCIe 4.0 NVMe technology with up to 6600MB/s sequential reads, random read/write 720K/700K IOPS

Table 3.1 - Test system configuration

Some applications tested offered cloud-based computational solutions accessed via a web-based interface clients, applications such as EasyCargo and CubeDesigner. For consistency, the same hardware/software configuration as in Table 3.1 was used to access the web clients. Although additional testing was occasionally performed on other configurations, these instances were only for partial results and did not impact the final conclusions of the study. These alternative setups were employed to validate and compare partial or minor aspects of the testing, ensuring that the core results remained robust and unaffected by any hardware variances.

3.1 OR-Tools development environment

The process of installing development environments appears straightforward post-completion, yet the setup of the open-source OR-Tools presented challenges that led to delays during the project's development phase. The initial step was to install the release tailored for the Windows operating system, adhering to the designated instructions, by following the guidelines specified on the "Installing OR-Tools for C++ from Binary on Windows" [26]. However, despite closely following the steps mentioned in the documentation, the execution of the examples included with the package was only partially successful. On the same lines, attempts to create

new applications to implement OR-Tools algorithms was met with failure. In both cases, the majority of the problems were linked to undefined or incorrectly defined paths pointing to essential header files. Adjusting the solution and project path variables to align with those used in successful sample examples was unsuccessful, facing persistent errors regarding undefined paths. Various OR-Tools source files within the installed package persistently generated these errors. In response to the initial setbacks, a second attempt to configure OR-Tools by compiling directly from the source files was also unsuccessful. Following the provided documentation guided instructions on "Building from source OR-Tools C++ on Windows" [27] found within the comprehensive Google OR-Tools installation guide required an even deeper engagement with the configuration details and dependencies necessary for environment setup.

3.1.1 Environment setup

Given the setbacks during the second attempt to install the C++ version of OR-Tools from the source files, installing OR-Tools for Python appeared as a viable alternative. With Python already installed on the test system, the setup was straightforward, by simply following the instructions from the "Installing OR-Tools for Python from Binary on Windows" [28], executing the following command at the command prompt:

```
python -m pip install ortools
```

The explored open open-source algorithms for the purpose of this study are contained within the OR-Tools package, representing the most current stable release available at the time of the research. For the purpose of implementing and testing these algorithms, the development environment utilized included Visual Studio Code, enhanced with several pertinent extensions. This setup provided an all-in-one robust framework for coding, debugging, managing the project

files efficiently, while also ensuring a light and agile development environment. The extensions mentioned in the list below were integrated as part of the VSCode test environment, needed for Python support and streamlining the coding process. The specific configuration is listed below:

- Python version initially installed was 3.11.8, updated to 3.11.9 during testing
- OR-Tools version 9.9.3963
- Visual Studio Code (VSCode) version 1.88.1, which was subject to automatic updates throughout the development process; included the following extensions
    - Pylance (latest update v2024.4.1) - this extension, which was updated automatically during my project, supports Python language development
    - Python (latest update v2024.4.1) - another automatically updated extension during my project, which facilitates various functionalities including debugging and provides interfaces for other extensions like Pylance and the Python Debugger
    - Python (latest update v2024.4.0) - Also part of the environment to ensure a comprehensive support system for Python development within VSCode

3.2 Selecting the data sets

3.2.1 General considerations

To get the best results and to reveal strengths and weaknesses of different algorithms, it is necessary to use various representative sets of data. The initial expectations were for certain algorithms to be more efficient, take less time to return a satisfactory result, or return better space occupancy results given certain stop limits such as a limited number of passes or limited time on a specific type of data. Not all algorithms are able to handle the entire range of test data sets (including constraints), some being able to handle a fixed number of dimensions, constraints or

stop limits, while others would present a certain degree of flexibility/adaptability. Implementing or declaring a new dimension or constraint could be either impossible, or as simple as declaring a new variable, depending on the application: this could prove to be an important factor during the algorithm testing and evaluation process. It should also be mentioned that a higher algorithm adaptability to a variety of problems could lead to a higher risk of efficiency loss in terms of processing time. Carefully determined data sets can make the difference between a successful testing and evaluation process and a failed one, by the ability to emphasize and prove the strengths and weaknesses of certain algorithms.

3.2.2 Measurement units

For this study, from an algorithm perspective, a test environment agnostic in relation to measurement units should present a valid base for comparison. However, given the objective of this thesis, paid application algorithms are only available and accessible through their interfaces. This introduces an extra layer of obfuscation and complexity – the algorithms are not directly accessible, and the objects dimensions need to be defined in common measurement systems. From this perspective, each software package came with its limitations in regard to objects and bins dimensions definition. Some software packages do not allow the use of decimals for object or bin dimensions, limiting the data sets to integers. While this does not appear to have a high impact, care needs to be taken regarding the use of same type of input data (integer or float), since computational efficacy can be significantly impacted.

Regarding the accepted units, some packages came preset to US units (inches, feet, pounds), while others to International System of Units (SI) (meters/centimeters, kilograms). It is beyond the purpose of this thesis to study the impact of using various measuring systems towards the test results. At the same time, it is unknown if the user facing interface units are converted to

a unit agnostic system before algorithm processing. For the purpose of this testing and evaluation centimeters and kilograms were selected for object data. To be noted, the weight of the objects should not have an impact during algorithm selection or processing results. Based on test observations, the applications only verify if the weight capacity of the container is higher than the total weight of the objects. It is unknown at this point, out of scope, and later inconclusive based on the test results, if object weight variability has a significant impact regarding the objective of this thesis. Every object in the data sets was added with a weight of 1kg, the total weight of the objects being approximately one order of magnitude lower than the capacity of the container. Package and container dimensions have been defined in centimeters. This allows for sufficient granularity in order to not have the need for decimal places, while allowing, as needed, dimensional variability.

3.2.3 Considerations regarding object orientation and other constraints

For this thesis, every object has been allowed all possible orientations while loading the container. There are no other constraints that have been considered or implemented for the purpose of this study. Examples of such constraints are: center of gravity, stacking limitations in terms of layers or weight, stacking order – larger or smaller objects at the bottom, or in the back of the container. Some of these constraints can have a significant impact in regard to aspects such as safety, efficiency during items loading and unloading in the container, or might be imposed by product limitations such as maximum layer stacking. Given the scope of this thesis, the afore-enumerated constraints are considered irrelevant towards achieving the goal and satisfying the problem statements formulated in chapter 1.

3.2.4 Data set attributes

While deciding the input data sets, the following attributes and concepts mut be considered, assessing them as relevant to ensure testing of various algorithm capabilities:

- Variability/diversity – box diversity introduces the necessary variability to challenge an algorithm's ability to handle a more diverse range of 3D objects
- Quantity – a high or low number of items can be considered an edge case and could present a challenge for the occupancy or fitting algorithms
- Aspect ratio – depending on the algorithm, aspect ratio can play a critical role by allowing the algorithm to focus on the dimensions of the objects, or other factors such as fill time and iterations
- Edge cases – extreme object dimensions are critical in testing the algorithm robustness and consistency in terms of results delivery
- Diverse or uniform size – challenging for an algorithm to focus on optimal placement and space utilization
- High quantity small items – algorithm scalability with a large number of small items (distribution centers)
- Almost-fit – challenge with boxes too large to fit neatly
- Long items – test capabilities to handle items that might require specific orientation and positioning

3.2.5 Object data

A few data sets are readily available, from different sources, for testing different algorithms. Unfortunately, for the purpose stated in the previous sections, the data sets found as used for optimization algorithm testing are not suitable for the use case of this thesis related to

container loading, as they don't meet criteria or guidelines for testing specific capabilities, as mentioned in section 3.2.4. Two of these data sets are presented below, solely for exemplification purposes and format visualization.

```
Input:     weight []        = {4, 8, 1, 4, 2, 1}
Bin Capacity c   = 10
Input:     Items []         = {1, 2, 3, 4, 5, 6}
Item length []   = {3, 4, 3, 8, 7, 2}
Item width []    = {3, 5, 7, 7, 2, 3}
Bin dimensions []     = {7, 14}
```

Representative datasets for the purpose of this research are presented in the following paragraphs and tables. The first data set (DS1) is a collection of objects with a wider variety of sizes, from small to large, with the purpose to test the algorithms' ability to handle varying dimensions and optimize space with a non-uniform set of items. The algorithm will require complex fitting logic and may need to rotate the boxes for a best fit scenario. The DS1 data is represented in Table 3.2.

Box_ID	Length	Width	Height	ds001
b100	58	58	58	150
b101	92	50	30	100
b102	120	80	60	50
b103	30	30	45	200
b104	75	75	100	25

Table 3.2 - Data Set 1 (DS1) diverse size

Data Set 2 (DS2) is a uniform set of objects of the same dimensions. The purpose of this particular data set is to test the basic functionality and efficiency of the tested algorithms, without the challenges presented by the other data sets. In particular, this data set presents another type of challenge – based on the dimensions of the boxes, even if the total volume is less than a

container, it is impossible to fit all the items. The DS2 data is represented in Table 3.3 - Data Set 2 (DS2) uniform size.

Box_ID	Length	Width	Height	ds002
b110	60	60	60	300

Table 3.3 - Data Set 2 (DS2) uniform size

Data Set 3 (DS3) features a high variability set, encompassing a wide range of box sizes from very small to large. This set aims to challenge the algorithm's ability to handle diverse box dimensions in a single packing scenario. Algorithms will need to utilize complex fitting logic and possibly rotate boxes to achieve the best fit. This data set mimics a real-world packing environment where the variety of item sizes can significantly impact the packing efficiency. The DS3 data is illustrated in Table 3.4 - Data Set 3 (DS3) high variability. Most commercial applications attempt to place the box types in a certain order, based on the volume of the box, and this particular case contains two box types that have the same volume: bh002 and bh010.

Box_ID	Length	Width	Height	ds006	Box volume
bh001	20	20	15	900	6000
bh002	30	20	15	1000	9000
bh003	12	12	12	1200	1728
bh004	14	14	30	1200	5880
bh005	12	45	35	500	18900
bh006	70	50	12	250	42000
bh007	32	60	37	200	71040
bh008	85	18	50	200	76500
bh009	25	70	55	200	96250
bh010	10	15	60	200	9000

Table 3.4 - Data Set 3 (DS3) high variability

3.2.6 Container data

For this study, one of the most common shipping container sizes has been selected, the 40-foot shipping container, high cube. There are slight variations for the values of the internal dimensions of a 40 foot container, the dimensions selected for analysis can be found in Table 3.5 - Container dimensions. This aspect is important, as one application (see section 3.3, subsection related to Cargo-Planner) uses the external dimensions of the container to calculate and return the occupancy rate, leading to discrepancies. Due to this inaccuracy, occupancy rates for this thesis have been calculated or confirmed manually at least for one test/application set, based on the number of packed boxes and their volume.

Container ID	Length	Width	Height
c001	1180	230	265

Table 3.5 - Container dimensions

3.2.7 Relative volume observations

When analyzing the volume of boxes in each dataset relative to the volume of the container, represented by percentages over and under 100%, it's important to consider how these datasets can influence the development and testing of bin packing algorithms. The relative volume of each dataset relative to the container volume can be found in Table 3.6 - Data set relative available volume. DS1 and DS3 have a maximum available volume of 131%, allowing the algorithms to select various box distributions to fill the container in search for optimal container volume occupancy. The performed testing showed that most commercial applications start by placing the higher volume boxes first, therefore maximizing occupancy by using the smaller boxes to fill the smaller spaces left open. In contrast, DS2 only has a maximum available

volume of 90%, however due to the box dimensions, not all of them can be fit inside the container.

Data set	Relative maximum available volume (%)
DS1	131
DS2	90
DS3	131

Table 3.6 - Data set relative available volume

3.3 Testing existing commercial applications

The results summary of the tested applications are presented in the following tables. Each application has its own implementation particularities in terms of stop limits or options, therefore multiple results might be available from the same application for a specific data set. One of the basic tested aspects was algorithm consistency and results repeatability with the same input data. This aspect is relevant for the study in the following manner: algorithms could be consistent and predictable, generating the same results based on the same input data, on the other hand the starting point can be randomized at application level, or algorithms can determine different optimal computational paths during execution, resulting in variability of the final container occupancy volume.

The set of tables below follow with summary results for the other tested container loading applications. The usability of the interfaces and/or the results of a limited number of tests allowed initial observations in relation to various performance factors of the application. Table 3.7 displays a testing results summary of the container packing software PackVol, on the various data sets presented in section 3.2. It can be observed that while having the same initial constraints and stop limits applied to the same dataset, the results are the same in terms of

solution time and volume occupancy. Although a more detailed analysis of the results is be presented in chapter 4, it is obvious that the “random start” option has an impact in relation to both “best solution time” and “volume occupancy”.

No	Data set	Options/limits	Best solution time (s)	Occupancy (volume %)
1	DS1*	1s	1	93.8
2	DS1	1s, random start	1	94.07
3	DS1	1s, random start	1	93.95
4	DS1	1s, random start	1	92.9
5	DS1	1s, random start	1	93.9
6	DS1	1s, random start	1	94.31
7	DS1*	10s	4	93.03
8	DS1*	10s	7	93.18
9	DS1	10s, random start	4	93.91
10	DS1	10s, random start	3	92.94
11	DS1	10s, random start	6	94.05
12	DS1	100s	36	93.61
13	DS1	100s, random start	17	94.95
14	DS1	100s, random start	75	98.69
15	DS2	10s	1	68.48
16	DS3*	1s	1	97.72
17	DS3	1s, random start	1	97.78
18	DS3	10s	10	98.4
19	DS3*	10s	10	98.33
20	DS3	10s, random start	10	97.35
21	DS3	10s, random start	10	98.32
22	DS3	10s, random start	10	98.04
23	DS3	100s	46	98.5
24	DS4	100s, random start	46	98.5
*note: test has been executed a minimum of 2 times				

Table 3.7 – PackVol test results summary

Cargo-Planner is a web-only container loading application, with limited adjustability in terms of constraints and stop limits, as can be observed in Table 3.8. One note needs to be made regarding Cargo-Planner, in respect to the volume occupancy of the cargo, as it reports the result in relation to the outside dimensions of the container, not the internal dimensions. For the purpose of this thesis, all the measurements, dimensions and calculations have been done considering the internal dimensions of the container and the external dimension of all the items. As an example, for Cargo-Planner the occupancy rate returned by the application for data set DS1 is 86%. By manually calculating the rate by using the external dimensions of the container, the number of loaded boxes and their volumes, the result is 86.2%. The actual occupancy rate using the internal available space of the container considered for this thesis is 91.48%, with a difference of 5.48%, which could distort the end results if not considered.

No	Data set	Options/limits	Best solution time (s)	Occupancy (volume %)
1	DS1*	2s	2	91.48
2	DS1	1s	2	91.48
3	DS1	10s	2	91.48
4	DS5	2s	2	68.48
5	DS3	2s	2	98.63
6	DS3	10s	10	98.63
*note: test has been executed a minimum of 2 times				

Table 3.8 – Cargo-Planner test results summary

CargoWiz is wrapped as a Windows desktop application, as most of the tested packages. It presents very limited options in terms of adjusting or imposing stop limits for the optimization algorithms. The only obvious one is the number of “trials”, and based by the description in the

user guide it is the equivalent of what we refer to as “number of iterations”, which represents how many times the algorithm/application attempts to fill the container with the given set of boxes. The user guide indicates that “many loads can be improved, typically by a few percent less floor length taken, by having the program do 4 trials” [29]. In case of our data sets, test results showed no improvements in case of DS1, and 0.7% improvement in case of data set DS3.

No	Data set	Options/limits	Best solution time (s)	Occupancy (volume %)
1	DS1*		1.5	80.6
2	DS1	4 trials	1.5	80.6
3	DS2		1	68.48
4	DS3*		3	97.05
5	DS3*	4 trials	22	97.76
*note: test has been executed a minimum of 2 times				

Table 3.9 – CargoWiz test results summary

Cube-IQ also uses a Windows desktop application to interface with the user, while allowing a more extensive set of customizable stop limits and/or options, as can be observed in the summary results Table 3.10.

No	Data set	Iterations (non-improving)	Options/ limits	Iterations (total)	Best solution time (s)	Occupancy (volume %)
1	DS1*	1000			0.5	87
2	DS1	1000000			0.5	87
3	DS1*	100	100s	513	5	88.51
4	DS1*	1000	100s	3633	34	89.04
5	DS1	1000000	9999s, 95%		over 1h	89.16
6	DS1	1000000	5s, 95%		5	88.89
7	DS1	1000			0.5	86.5
8	DS1	1000000	5s		5	92.91
9	DS1	1000000	100s		72	92.87
10	DS1	1000000	9999s		over 1h	92.96
11	DS2				1	68.48
12	DS3	1000			5	97.22
13	DS3				11	98.27
14	DS3		1000		109	98.73
*note: test has been executed a minimum of 2 times						

Table 3.10 – Cube-IQ test results summary

Beyond the scope of this research, the application offers a set of "selection rules" for multiple containers, presented in Table 3.11 – Cube-IQ container fill options list for multiple containers. It would be interesting as future development to investigate what impact these options have towards container loading optimization and why are they only present for multiple containers.

Container fill option list
- Best fill - Minimum overall cost - One container only - Largest reaching minimum fill first - Sequenced reaching minimum fill first - Minimum overall cost + largest first - Cheapest first - Flexi fit - One container type only

Table 3.11 – Cube-IQ container fill options list for multiple containers

3.4 OR-Tools knapsack solver

The initial to solve the "bin packing problem - container loading" with OR-Tools involved using the knapsack algorithm. The OR-Tools documentation has a specific page with an example of implementation for this algorithm [30]. An example with a simplified data set, the data and results are outlined below. As demonstrated by the results, all items were successfully packed in the bin.

```
# Bin volume
bin_volume = 10 * 10 * 10  # Example bin with 1000 cubic units of
     volume

# Items (represented by their volumes)
items_volumes = [2 * 3 * 4, 4 * 3 * 2, 3 * 2 * 1, 2 * 2 * 2, 5 *
     3 * 2]
Total packed volume: 92
```

```
Packed items: [0, 1, 2, 3, 4]
```

Based on these results, the initial goal was to expand on the knapsack solver to achieve the goals of this thesis. However, by running various scenarios, it is obvious that the knapsack solver alone will not be suitable for the purpose of this study. The knapsack solver in OR-Tools is designated to solve problems resembling to the classic knapsack problem – maximizing the total value of items packed into a container without exceeding its capacity limit. However, the knapsack problem and the corresponding algorithm from OR-Tools take into account a singular dimension for the object, limiting the usability of the algorithm.

A simple object with a volume of 12 x 1 x 1 = 12 units would definitely fit in a bin of volume 10 x 10 x 10 = 1000. However physically the object would not be able to fit into the container due to the fact that one of the dimensions – 12 – will simply not fit if the object is positioned in an orthogonal position. Another plausible "non-fit" scenario would be packing two objects of 6 x 6 x 6 = 216 volume in the same size bin as the above. In terms of volume, the knapsack solver will assess the solution as being viable, since 216 + 216 < 1000. While attempting to physically place the object inside the bin, without overlapping, it is an impossible task since the sum of same side dimensions for two objects is 6 + 6 = 12, larger than the largest bin dimension, 10.

3.5 OR-Tools linear solver

The Bin Packing Problem does have a specific example page in the OR-Tools documentation [31]. This section exemplifies filling a number of bins with objects of certain weights. Although not a perfect fit to the container packing strategy, testing a Python script using the linear solver was justified. The original example solver from the Bin Packing Problem page

creates the following model and returns the results below. The code was updated to display execution time.

```
def create_data_model():
    """Create the data for the example."""
    data = {}
    weights = [48, 30, 19, 36, 36, 27, 42, 42, 36, 24, 30]
    data["weights"] = weights
    data["items"] = list(range(len(weights)))
    data["bins"] = data["items"]
    data["bin_capacity"] = 100
    return data
```

```
Bin number 0
  Items packed: [0, 1, 2]
  Total weight: 97

Bin number 1
  Items packed: [3, 4, 5]
  Total weight: 99

Bin number 2
  Items packed: [6, 7]
  Total weight: 84

Bin number 3
  Items packed: [8, 9, 10]
  Total weight: 90


Number of bins used: 4
Time =  10  milliseconds
```

The next step was to implement a suitable application based on the linear solver, one that would read data from a data set and attempt positioning of the data set objects in the prescribed bin.

3.5.1 Basic implementations

Script py102 is the initial implementation script written with data set DS1 using the container C01 from tables Table 3.2 - Data Set 1 (DS1) diverse size and Table 3.5 - Container dimensions. Test results are included in the Table 3.12, with a sample results output from the

script. The test appears to execute in a very short period of time, and result in occupancy rates of 100%. This occupancy rate is backed up by mathematically calculating it based on the number of items packed, their volume, and the known bin volume of 71921000 as the container from the dataset. The calculated volumetric rate is:

$$71920816 \div 71921000 \times 100 = 99.99974\%$$

No	Data set	Best solution time (s)	Occupancy (volume %)	box1	box2	box3	box4	box5
1	DS1	15-20ms	100	43	99	50	173	25
note: test has been executed 10 times								

Table 3.12 – Script py102 results and box loading count

Scrypt py104.py represents a different approach to solving the problem, with different results, as presented in the table Table 3.13. One of the improvements over the previous algorithm can be observed as execution time difference. In comparison with py102.py, this script returns consistently the best solution in one third of the time. This aspect is important, as it can represent a significant difference if the script is executed on a larger data set.

No	Data set	Best solution time (s)	Occupancy (volume %)	box1	box2	box3	box4	box5
1	DS1	6-7ms	100	150	100	0	200	25
note: test has been executed 10 times								

Table 3.13 – Script py104 results and box loading count

3.5.2 3D positioning implementations

Script py132 represents and improvement from the previous scripts in the following aspects: it reads the data from a text file, thus allowing for flexibility and agility in execution,

with the most important improvement in regards to the methods the script attempts to fill the container, in two steps:

- step 1 – uses the linear solver to calculate the maximum occupancy of the container in terms of volume
- step 2 – uses a 3D positioning algorithm for the bins inside the container; the second step aims to position the boxes to achieve the best occupancy as close as possible to the one calculated during step 1

The second step, however, runs only one iteration. The difference between the two computational steps is significant, between 99.5% volume occupancy calculated in step 1 compared to 71.14% occupancy calculated during the second step based on actually positioning the boxes inside the container, without overlapping. Table 3.14 and Table 3.15 display the results of running script py132 on the data sets selected for this thesis. To confirm results consistency and repeatability, every test was performed at least 3 times.

No	Data set	Section	Best solution time (s)	Occupancy (volume %)
1	DS1	step 1	0.002	99.51
2	DS1	step 2	1.25	71.14
3	DS2	step 1	0.002	90.1
4	DS2	step 2	0.21	68.84
5	DS3	step 1	0.065	99.94
6	DS3	step 2	1109.25	85.01

Table 3.14 – Script py132 results

Data set	Section	box1	box2	box3	box4	box5	box6	box7	box8	box9	box10
DS1	step 1	150	100	11	200	25					
DS1	step 2	150	100	0	200	0					
DS2	step 1	300									
DS2	step 2	228									
DS3	step 1	900	1000	1200	1200	500	250	200	162	0	200
DS3	step 2	900	1000	1200	1200	500	250	191	30	0	200

Table 3.15 – Script py132 box distribution

This is the final OR-Tools implementation evaluated for this thesis, with the mention that other packing techniques can be implemented for results improvement. Techniques such as multiple runs with randomized starting points, randomizing the sequence of box placement, implementing advanced placement algorithms – best fit, first fit decreasing, rotations – or incorporating a linear solver could further refine object placement within the container. All these techniques can be used for a more detailed object placement and achieve container actual fill results closer to the ones obtained through the linear solver, based on object volume.

4 ANALYSIS

The previous chapters outlined the background, the motivation for this thesis, the methodology and implementation, with their own challenges encountered during the process. The final sections of chapter 3 presented the results of utilizing the commercial applications and the scripts built for this study on the selected data sets. While the previous chapter presented a few immediate high level observations based on the raw test results, this chapter delves deeper by compiling and analyzing the data, compounding various sections of the results and presenting them in a comprehensive manner to create an analysis that will better support the conclusions of the thesis. To ensure the analysis and conclusions do not deviate from the study's original objectives, the analysis focuses to address as best as possible the problem statements formulated in section 1.3. Below is a succinct reiteration of the problem statements, presented in form of a simple list, for clarity:

1. Problem 1 – Volume Occupancy
2. Problem 2 – Algorithm efficiency
3. Problem 3 – Consistency

4.1 Volume occupancy

The first problem statement addressed is the one related to volume occupancy. As it can be observed in Table 4.1 and Figure 4.1, for the same data set commercial applications returned a spread of results, varying between 80% and almost 100%, with a variability over a total span of

18%. In comparison, the script py104, one of the initial OR-Tools linear solver [31] implementations returns 90.7% occupancy.

Test ID	PackVol	Cube-IQ	Cargo-wiz	Cargo-planner	EazyCargo	py104	py132
1	93.8%	88.51%	80.6%	86%	87.66%	90.7%	71.14%
2	93.8%	88.51%	80.6%	86%	87.66%	90.7%	71.14%
3	93.8%	88.51%	80.6%	86%	87.66%	90.7%	71.14%
4	92.9%	87%					
5	93.9%	89.04%					
6	94.31%	89.04%					
7	93.91%	89.16%					
8	94.05%	88.89%					
9	93.03%	86.5%					
10	94.95%	92.91%					
11	98.69%	92.87%					

Table 4.1 – Volume occupancy – data set DS1 overall results

From a certain perspective this appears to be a good result, however we must take into account that the linear solver quality of results is highly dependable on the way the constraints are introduced and how the solver is used in the context of the application. The py132 script, on the other hand, still uses the same linear solver from OR-Tools, however the slight differences in the way the solver is used in the application returns a maximized volume occupancy of 99.51%, which is 9% higher than the same solver used in script py104. Figure 4.1 only represents py132 script results, as results for script py104 are deemed not relevant to this comparison, for the reasons mentioned in section 4.1.1.

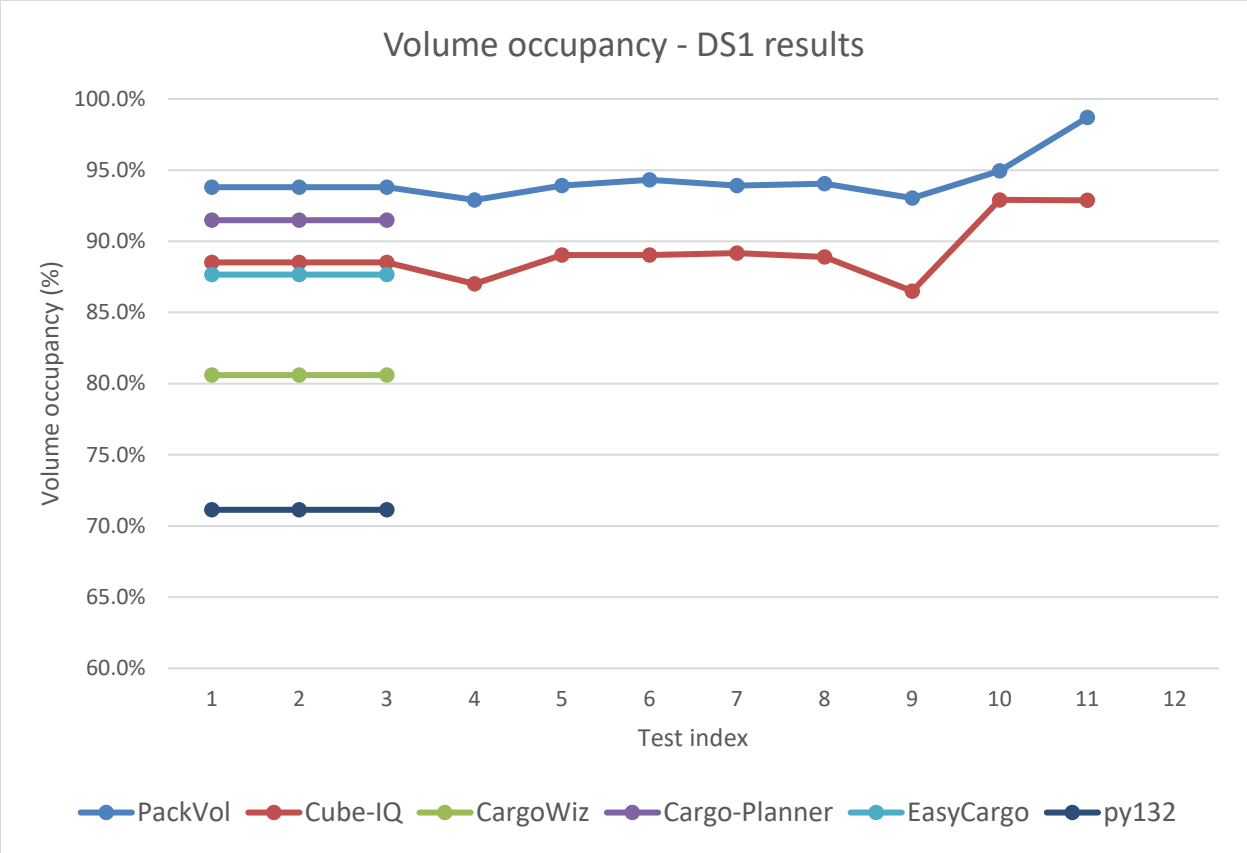

Figure 4.1 – Volume occupancy – data set DS1 overall results

4.1.1 Volume occupancy vs. box fitting for OR-Tools solver applications

Another pertinent observation based on the DS1 overall results table and chart is related to the solver's results in terms of volume occupancy compared to the results of a fitting algorithm. The fitting algorithm attempts to arrange all the boxes inside the container by positioning them next to each other. Even if the fitting algorithm attempts to achieve the results returned by the linear solver, the difference between the occupied volumes is obvious in this case, with the fitting algorithm we obtain only 71.14% occupancy (compared to 99.51% returned by the solver, based solely on volume).

4.1.2 Volume occupancy with default options

As stated in previous sections of this thesis, various applications present configurable options for algorithm tuning. These options are discussed under section 4.2. In the default testing results for data set DS1 representation from Figure 4.2 can be observed a high variation between commercial application results, from 80.6% to 93.8%. This is similar to the results obtained in section 2.1.1 that presents the results of the preliminary analysis, where we had a variation between 80% and 95% on the preliminary data set. In the same graphical representation can be observed that the OR-Tools implementation managed to fill the container to 71.4% occupancy.

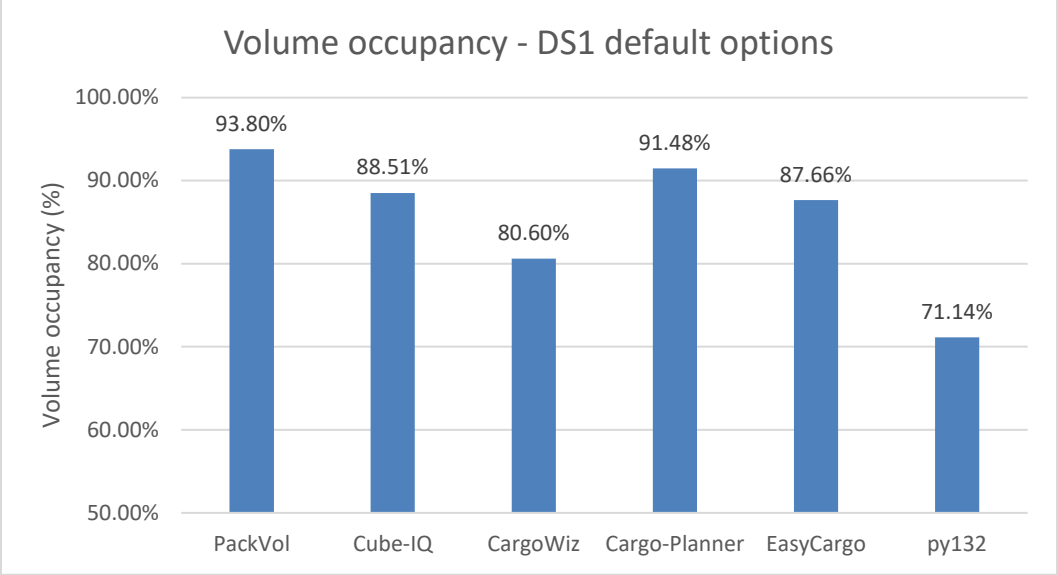

Figure 4.2 – Volume occupancy – data set DS1 with default application options

Data set DS2 is a particular choice (see section 3.2.5) as from a volume perspective the total sum of the item volumes is only 90% compared to the volume of the container. However, as can be observed in Figure 4.3 every application returned exactly the same results in terms of occupancy rate, and the value of the result is 21.6% lower than the maximum of 90% that could be achieved. This is due to the dimensions of the boxes, as they do not allow packing more than

68.4% by volume of the container. In a relatively simple scenario, all applications performed exactly the same.

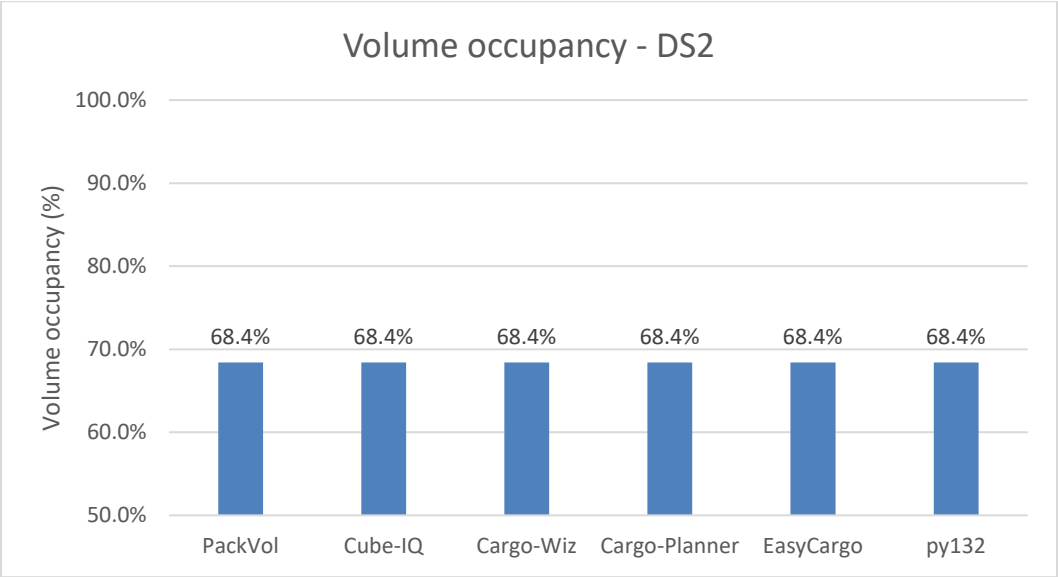

Figure 4.3 – Volume occupancy – data set DS2 with default application options

DS3 is a challenging set for the applications, observation that will become more obvious in section 4.2. In this section, however, the results are better than for DS1 and DS2. The challenge posed by DS3 comes from the high variability of the items in terms of size and the high count for every item in comparison to the other data sets. The somewhat surprising result appears inherently as a consequence of the high variability and high box type counts – a result of which is inevitably represented by sets of boxes with reduced dimensions. These boxes are used by the algorithms to fit smaller empty spaces inside the container, therefore maximizing occupancy of the available volume. The results can be observed represented in Figure 4.4, and we can observe that every application returned significantly better results in comparison with DS1 and DS2, with py132 returning approximately 14% higher occupancy for DS3 in comparison to DS1.

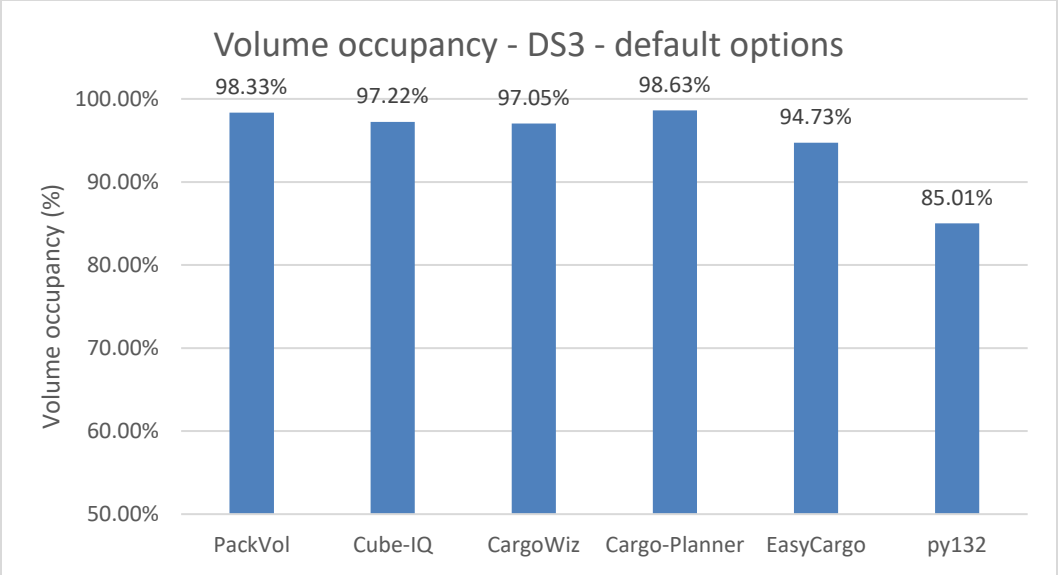

Figure 4.4 – Volume occupancy – data set DS3 with default application options

4.2 Algorithm efficiency

The research reveals several intriguing stop limits implemented in commercial applications. Maximum number of iterations, maximum number of iterations without improvement, time limits, percentage limits are some of the configurable aspects for commercial applications. To be noted that not all applications provide the user with these customization settings. The following sections address two of these settings, computation time and random start.

4.2.1 Computation time

The first item addressed in this section is related to computation time. In order to conduct this test, a longer time has been allowed for the applications to obtain an optimal result. To be observed that the time stop limit is not the only factor for what is considered an "optimal result" by the applications, also some of the applications do not allow any time limit adjustments. The

test has been conducted with a time limit of 100 seconds (where possible), with the results for data set DS1 presented in Table 4.3.

Software application	Time (s)	Volume occupancy (%)
PackVol	75	98.69
Cube-IQ	72	92.87
CargoWiz	1.5	80.60
Cargo-Planner	2	97.48
EasyCargo	1	87.66
py132	1.25	71.14

Table 4.2 – Computational time – DS1

Figure 4.5 displays a graphical representation of these results, for better visualization of computational time in rapport to volume occupancy. Cargo-Planner, EasyCargo did not allow time limit adjustments. For CargoWiz, the option “4 trials”, which represents 4 attempts, did not result in an increased occupancy result, therefore the time elapsed for the 4 trials was discarded, and only the time for a single trial used as a result. PackVol and Cube-IQ, with limits of 100 seconds, found the optimal results at second 75, respectively 72. The OR-Tools application falls into the first category, as it does not allow any time limit adjustment.

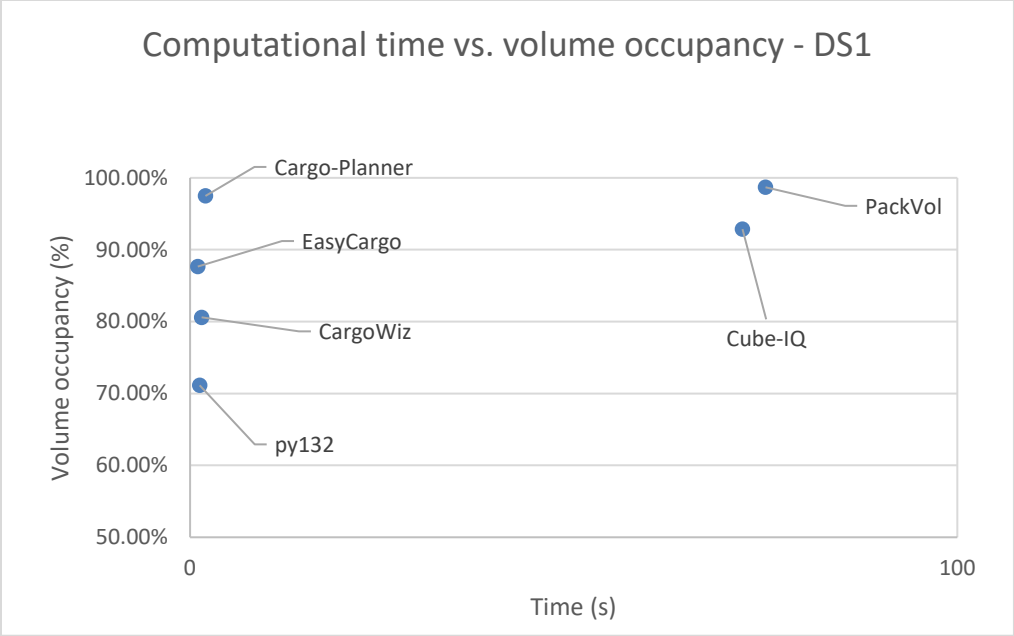

Figure 4.5 – Computational time vs. volume occupancy – DS1

The next test has been performed for data set DS3, which contains a high variability, high count of boxes. With time limits of 100 seconds for PackVol and 1000 seconds for Cube-IQ, and the "4 trials" option active for CargoWiz, the results vary and can be observed in Table 4.3. The py132 script does not have any adjustments that would allow it to extend the optimization time.

Software application	Time (s)	Volume occupancy (%)
PackVol	46	98.50
Cube-IQ	109	98.73
CargoWiz	22	97.76
Cargo-Planner	2	98.63
EasyCargo	17	94.73
py132	1109.25	85.01

Table 4.3 – Computational time – DS3

It can be observed that PackVol and Cube-IQ maintained a similar order of magnitude in terms of computation time, while CargoWiz and EasyCargo time increased by an order of magnitude. For CargoWiz, the "4 trials" led to different results between the 4 tries, with the best result being captured for the purpose of this test. In the case of EasyCargo, the high box count did pose some computational challenges, we can observe in this particular case the reduced occupancy rate in comparison with the other commercial applications. In this particular scenario, Cargo-Planner managed to calculate a high level of 98.63% occupancy, in less than 2 seconds. At the opposite end, py132 took 1109 seconds to calculate the result, and obtained the lowest volume occupancy between the tested applications. The results can be observed for a visual comparison in a graphical representation in Figure 4.6.

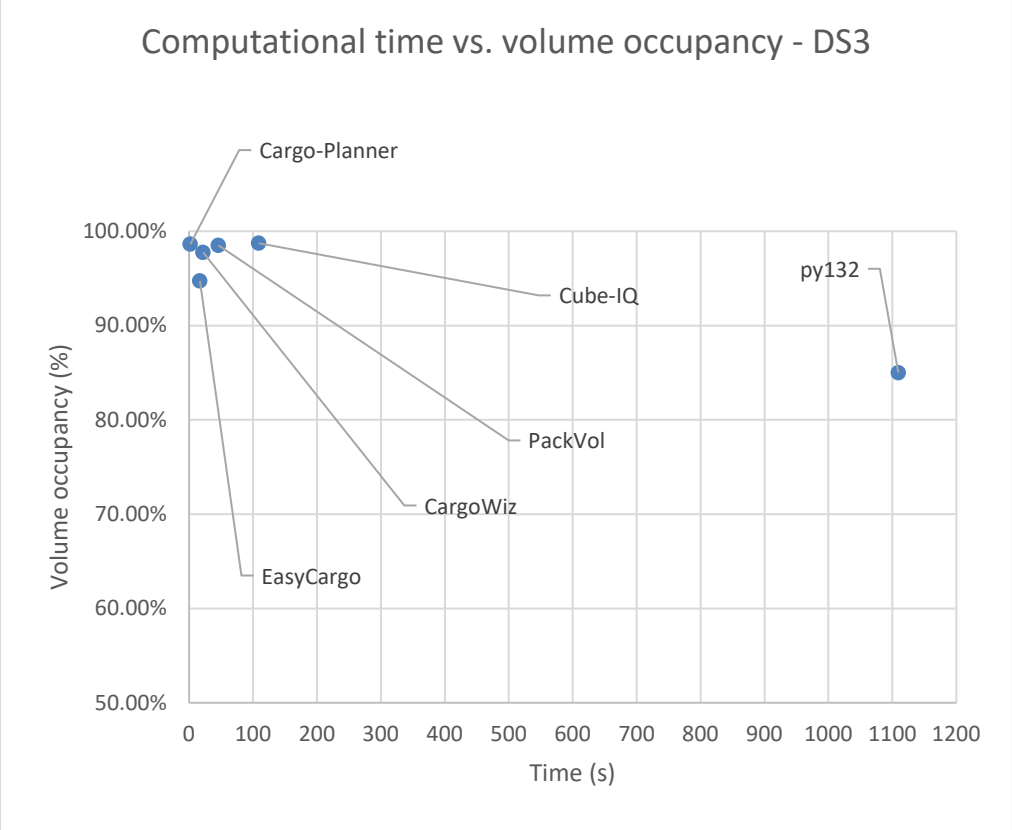

Figure 4.6 – Computational time vs. volume occupancy – DS3

4.2.2 Random start

An interesting observation can be made regarding PackVol, where opting for "random start" does appear to have an impact towards the final volume occupancy across all data sets, at least from an analytical perspective. With this option disabled, considering the same input data set, testing returned the same result multiple time. With the option enabled we can observe a consistent randomness in the results displayed in Table 4.4.

Test ID	**DS1**		**DS3**	
	default	**random start**	**default**	**random start**
1	93.8%	94.07%	97.72%	97.78%
2	93.8%	93.95%	97.72%	97.82%
3	93.8%	92.9%	97.72%	98.26%
4	93.8%	93.9%	97.72%	96.96%
5	93.8%	94.31%	97.72%	96.78%

Table 4.4 – Default vs random start – PackVol – occupancy results

As it can be observed, in the case of DS1 and DS3, testing returned mixed results, from a 1% decrease to a 0.5% increase in volume occupancy. There might be data sets and use cases where a random start could have a significant impact towards achieving better results, however during this study the results were negligeable. A visual representation of the results can be observed in Figure 4.7.

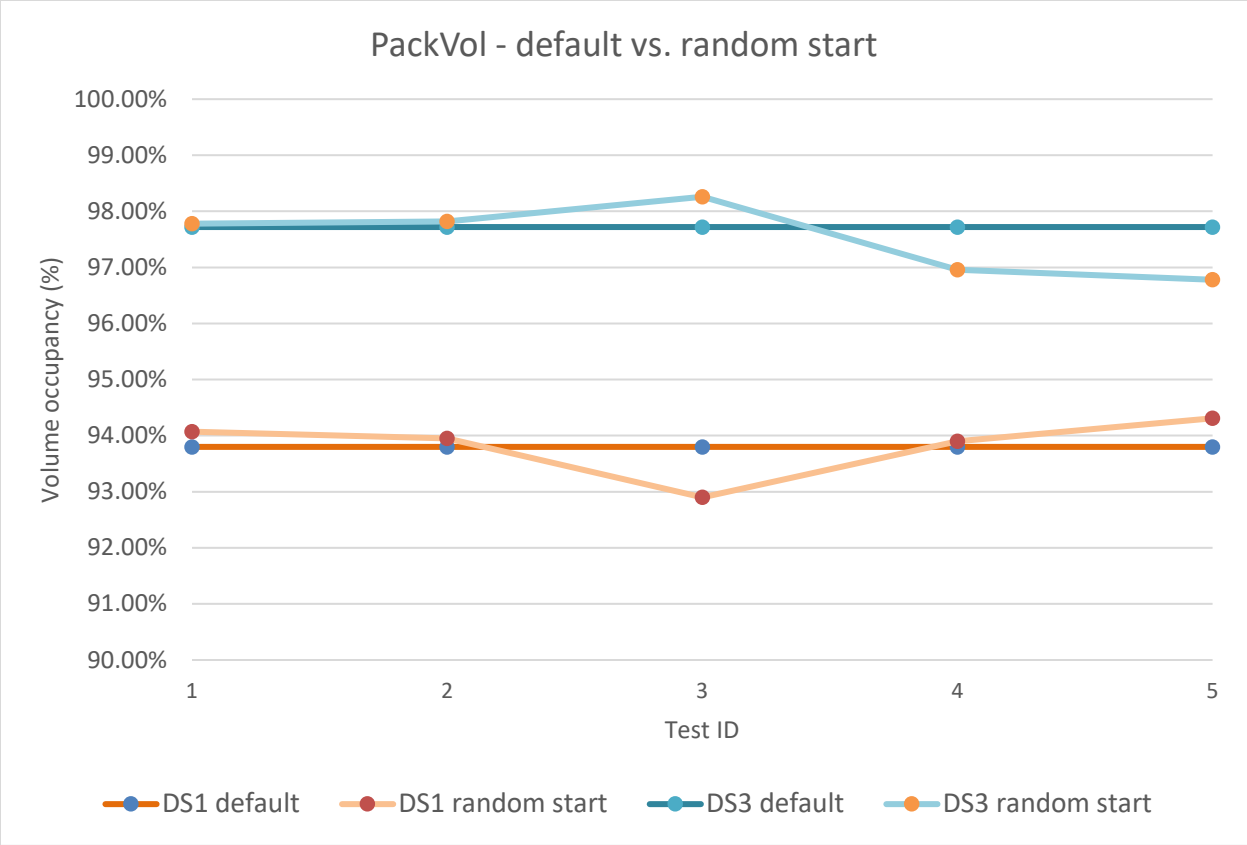

Figure 4.7 – Default vs. random start – PackVol – occupancy results

4.3 Results consistency

In terms of results consistency, a few observations can be made based on the test results. As detailed in Table 4.1 and Figure 4.1, there are patterns of consistency in the overall results. PackVol, for example, returned 93.8% multiple times, based on the input conditions remaining the same and the "random start" option not being selected. Cube-IQ and Cargo-Planner also display results consistency and repeatability, if the input data, constraints or stop limits stay the same. Results for Cube-IQ can be observed in Table 4.5. This leads to an important conclusion for this thesis, the attempts of the container loading applications to optimize results follow the same path over and over instead of randomizing the approach. This conclusion is supported by the opposite behavior of PackVol, where enabling "random start" does, in effect, lead to a

variability of the results, considering the same input data. This can be clearly observed in Figure 4.1 and Table 4.1. The py132 implementation based on the OR-Tools linear solver also show consistent results, as there is no randomization implemented at this point.

Test ID	Iterations stop constraint	Time constraint (s)	Actual iterations	Actual time	Volume occupancy (%)
1	1000	0		0.5	87
2	1000	0		0.5	87
3	1000	0		0.5	87
4	100000	0		0.5	87
5	100	100	513	5	88.51
6	100	100	513	5	88.51
7	100	100	513	5	88.51
8	100	100	513	5	88.51
9	1000	100	3633	34	89.04
10	1000	100	3633	34	89.04
11	1000	100	3633	34	89.04

Table 4.5 – Cube-IQ DS1 test results consistency in various scenarios

5 CONCLUSIONS AND FUTURE DEVELOPMENTS

To conclude this study, it is evident that the exploration of container loading optimization through commercial and open-source algorithms has provided a deeper understanding of the capabilities and limitations associated with different approaches in different scenarios. The analysis has demonstrated that while there are numerous strategies for improving efficiency of the bin packing process, such as implementing various stopping criteria and using different algorithm configurations, the results are often dependent on specific scenario details. The conducted testing has revealed that adjustments such as increasing the computation time or opting for randomized start does not uniformly lead to better volumetric occupancy of the container. A similar statement can be made in relation to stop limits such as the maximum number of iterations without improvements, where a significant increase of the time stop limit delivered only marginally improved results, as can be observed in section 4.2. In the particular case of PackVol the opportunity exists to randomize starting conditions, leading to slight outcome improvements over the tested data sets. This aligns with the hypothesis that not all optimization constraints and stop limits are effective across different implementation scenarios and different data sets.

The research can also support the conclusion that a relationship between algorithm complexity and practical application is not linear. Some enhancements can lead to improvements in volume occupancy, however the variance of these results highlights the necessity of a strategic approach to algorithm selection and configuration based on a specific scenario. Since this study extended over a timespan of almost 10 years, the preliminary results of the commercial

applications (however on a different data set at the beginning of this study, and on a different capability hardware system) were similar to the resent results in terms of consistency within the same application and variability between them. Where at the start we were looking at 85% to 95% occupancy (Table 2.2), the recent study reveals in general 80% to 94% occupancy on a similar data set, as presented in section 4.1 (with one exception of 98%).

5.1 Problem statements

The thesis explored three problem statements regarding the performance of commercial and open-source optimization algorithms. The first problem statement addressed how the algorithms compare in maximizing volume occupancy. The analysis suggest that commercially applied algorithms generally provide quicker and more consistent results, while the open-source algorithms offer flexibility and potential for better results through applications developed for specific scenarios. The second problem statement focuses on efficiency. This can be observed and analyzed form multiple perspectives. From a computation time perspective, it would appear that Cargo-Planner performs the best, and for the end user this might be the case. However we need to take into account that this application is server based, and the customer accesses a client interface, therefore the computational strain is not imposed on the same system as for a desktop application.

The second problem statement addresses results consistency and repeatability. The conclusion in this case is two-fold. For a specific set of data, with the same constraints and stop limits, the applications return exactly the same results, which leads to the hypothesis that the same decision path is followed every single time, without any pre-analysis or pre-selection of a specific algorithm or algorithm configuration. Over multiple sets of data, or with different

constraints and stop limits, the algorithms still tend to deliver generally uniform, consistent results.

The second and third problem statements are concerned with the speed, computational iterations count and other configuration settings and stop limits. What can be observed is that results are mixed in terms of execution time. Generally, commercial algorithms are slower than the OR-Tools solver, when the open-source algorithms is utilized solely for volume occupancy. However, when the generated application introduces a second layer of complexity by actually positioning all the boxes inside the container, the execution time increases significantly. As an example of efficiency: increasing the time limit for Cube-IQ, by using data set DS1, allowed the algorithm to improve from 88.51% to 92.87%. On the same dataset PackVol managed 93.8% occupancy with default configuration, and by setting a 100s time limit, with the “random start” option achieving an impressive value of 98.69% occupancy.

5.2 Future developments

This study has contributed to academic understanding of container loading optimization, provided practical insights that can aid in future application developments. Application testing and analysis led to significant conclusions related to specific aspects of the algorithms and their usage, and it could benefit from an expansion of the test data sets to encompass other critical scenarios. The commercial applications had no constraints in relation to box orientation, whereas the open-source based applications did not take such an aspect into account in this thesis. This is another opportunity for clear testing and analysis. In relation to execution speed, in certain aspects and scenarios, open-source implementation displayed a significant improvement in comparison to the commercial applications. The results can benefit from further investigation on this aspect.

Implementing multiple iterations, random start, based on the analysis of the results one can even hypothesize that a preliminary analysis of the data set (such as box and container dimensions) should benefit towards algorithm selection. The analysis section clearly displays examples of strengths and weaknesses of certain applications in certain situations, therefore with a significant input data set variation, enough conclusions can be drawn towards predetermining or estimating the benefits of certain parameters, stop limits, constraints, or algorithm types to be applied. Most recently, we do benefit from machine learning and artificial intelligence, it could play a significant role in rapid testing and adaptability to future scenarios based on the ones already tested.

6 BIBLIOGRAPHY

[1] Merriam-Webster, "Merriam-Webster - Optmization," [Online]. Available: http://www.merriam-webster.com/dictionary/optimization. [Accessed November 2016].

[2] S. J. Wright, "Encyclopaedia Britannica - Optimization," 03 12 2013. [Online]. Available: https://www.britannica.com/topic/optimization. [Accessed November 2016].

[3] I. P. J. C. A. a. N. P. W. Gent, "Complexity of n-Queens Completion," 30 August 2017. [Online]. Available: https://eprints.whiterose.ac.uk/141513/. [Accessed April 2024].

[4] N. J. A. Sloane, "A000170 Number of ways of placing n nonattacking queens on an n X n board.," 2016. [Online]. Available: https://oeis.org/A000170. [Accessed April 2024].

[5] T. B. Preußer, "The Q27 Project," 24 Oct 2017. [Online]. Available: https://github.com/preusser/q27?tab=readme-ov-file. [Accessed April 2024].

[6] S. D. N. Youn-Kyoung Joung, "Intelligent 3D packing using a grouping algorithm for automotive container engineering," *Journal of computational design and engineering,* vol. 1, no. 2, 2014.

[7] Ship A Car, Inc., "How Much Does It Cost To Ship a Container within the United States," November 2022. [Online]. Available: https://www.shipacarinc.com/how-much-does-it-cost-to-ship-a-container-within-the-united-states/. [Accessed April 2024].

[8] Omni Logistics, "How much does it cost to transport shipping containers?," [Online]. Available: https://omnilogistics.com/how-much-does-it-cost-to-transport-shipping-containers/#:~:text=With%20that%20in%20mind%2C%20here,costs%20can%20be%20considerably%20higher.. [Accessed April 2024].

[9] Hapag - Lloyd, "Hapag-Lloyd - Container Packing Brochure," 02 2010. [Online]. Available: https://www.hapag-lloyd.com/content/dam/website/downloads/press_and_media/publications/Container_Packing_Broschuere_engl.pdf. [Accessed December 2016].

[10 N. A. Z. M. R. M. S. Z. A. a. R. Y. U. Khairuddin, "Smart Packing Simulator for 3D Packing Problem Using Genetic Algorithm," *Journal of Physics: Conference Series,* vol. 1447, 2020.

[11 Wikipedia various, "Bin packing problem," April 2024. [Online]. Available: https://en.wikipedia.org/wiki/Bin_packing_problem. [Accessed 2017].

[12 Wikipedia various, "List of optmization software," [Online]. Available: https://en.wikipedia.org/wiki/List_of_optimization_software. [Accessed November 2016].

[13 G. Erdogan, "User Manual for BPP Spreadsheet Solver," 2015. [Online]. Available: User Manual for BPP Spreadsheet Solver. [Accessed October 2016].

[14 OpenOpt, "Home," OpenOpt, 2012. [Online]. Available: https://openopt.org/. [Accessed April 2024].

[15 OptaPlanner, "Solve planning and scheduling problems with OptaPlanner," 2023. [Online]. Available: http://www.optaplanner.org/. [Accessed April 2024].

[16 G. D. Smet, "OptaPlanner continues as Timefold," Timefold, 1 May 2023. [Online]. Available: https://timefold.ai/blog/optaplanner-fork. [Accessed April 2024].

[17 OptaPlanner team, "OptaPlanner user guide 9.44.0," OptaPlanner, [Online]. Available: https://www.optaplanner.org/docs/optaplanner/latest/planner-introduction/planner-introduction.html. [Accessed April 2024].

[18 Google, "Install OR-Tools," 28 03 2024. [Online]. Available: https://developers.google.com/optimization/install. [Accessed April 2024].

[19 "About OR-Tools," 2024. [Online]. Available: https://developers.google.com/optimization/introduction. [Accessed April 2024].

[20 Google, "Solving an LP problem," Google, 01 2023. [Online]. Available: https://developers.google.com/optimization/lp/lp_example. [Accessed April 2024].

[21 Google, "Advanced LP Solving," Google, February 2022. [Online]. Available: https://developers.google.com/optimization/lp/lp_advanced. [Accessed April 2024].

[22 F. Brandao, "Vector Packing Solver (VPSolver)," [Online]. Available: https://vpsolver.fdabrandao.pt/. [Accessed September 2016].

[23 F. Brandao and J. P. Pedroso, "Bin Packing and Related Problems: General Arc-flow Formulation with Graph Compression," September 2013. [Online]. Available: https://research.fdabrandao.pt/papers/arcflow_report.pdf. [Accessed September 2016].

[24 F. Brandao and J. P. Pedroso, "Arc-flow Formulation for Vector Packing - Poster," [Online]. Available: https://research.fdabrandao.pt/papers/arcflow_poster.pdf. [Accessed September 2016].

[25 J. Colin, "Comparing OR-Tools vs OptaPlanner for Multi Echelon Inventory Planning," Solvice, January 2022. [Online]. Available: https://www.solvice.io/post/comparing-or-tools-vs-optaplanner-for-multi-echelon-inventory-planning. [Accessed April 2024].

[26 Google, "Installing OR-Tools for C++ from Binary on Windows," 11 03 2024. [Online]. Available: https://developers.google.com/optimization/install/cpp/binary_windows. [Accessed April 2024].

[27 Google, "Building from source OR-Tools C++ on Windows," 15 04 2024. [Online]. Available: https://developers.google.com/optimization/install/cpp/source_windows. [Accessed April 2024].

[28 Google, "Installing OR-Tools for Python from Binary on Windows," 28 03 2024. [Online]. Available: https://developers.google.com/optimization/install/python/binary_windows. [Accessed April 2024].

[29 Softtruck, "Cargowiz User Guide," 2024. [Online]. Available: https://www.softtruck.com/CargoWizUsersGuide/CargoWizUsersGuide.pdf. [Accessed April 2024].

[30 Google OR-Tools, "The Knapsack Problem," 01 2023. [Online]. Available: https://developers.google.com/optimization/pack/knapsack. [Accessed April 2024].

[31 Google OR-Tools, "The Bin Packing Problem," 18 01 2023. [Online]. Available: https://developers.google.com/optimization/pack/bin_packing. [Accessed April 2024].

7 APPENDIX

Online repository located at this address: https://github.com/gabemihu/box-solver.

7.1 Implementation py104 listing

```
from ortools.linear_solver import pywraplp

print("Script 104")

def create_data_model():
    """Create the data for the example."""
    data = {}
    # Box dimensions: length, width, height, and quantity from
     ds1
    box_data = [
        (58, 58, 58, 150),
        (92, 50, 30, 100),
        (120, 80, 60, 50),
        (30, 30, 45, 200),
        (75, 75, 100, 25),
    ]
    # Container dimensions from c001
    container_dimensions = (1180, 230, 265)

    # Calculate the volumes of the boxes and the container
    data['volumes'] = [l * w * h for l, w, h, _ in box_data]
    data['quantities'] = [q for _, _, _, q in box_data]
    container_volume = container_dimensions[0] *
     container_dimensions[1] * container_dimensions[2]

    # Flatten the items list to account for multiple quantities
    data['items'] = []
    data['item_types'] = []
    for i, q in enumerate(data['quantities']):
        data['items'].extend([i] * q)
        data['item_types'].append(i)
    data['bins'] = [0]  # Only one container in this case
    data['bin_capacity'] = container_volume
    return data

def main():
```

```
data = create_data_model()

# Create the mip solver with the SCIP backend.
solver = pywraplp.Solver.CreateSolver("SCIP")

if not solver:
    return

# Variables
# x[i] = 1 if item i is packed in the container.
x = {}
for i in data['items']:
    x[i] = solver.IntVar(0, 1, 'x[%i]' % i)

# Constraints
# The total volume of items packed cannot exceed the
 container's capacity.
solver.Add(
    sum(x[i] * data['volumes'][data['item_types'][i]] for i
 in data['items']) <= data['bin_capacity']
)

# Each type of item can only be packed up to its available
 quantity.
for j in data['item_types']:
    solver.Add(
        sum(x[i] for i in data['items'] if
 data['item_types'][i] == j) <= data['quantities'][j]
    )

# Objective: maximize the total number of items packed.
solver.Maximize(solver.Sum([x[i] for i in data['items']]))

print(f"Solving with {solver.SolverVersion()}")
status = solver.Solve()

if status == pywraplp.Solver.OPTIMAL:
    total_items_packed = sum(x[i].solution_value() for i in
 data['items'])
    total_volume_packed = sum(x[i].solution_value() *
 data['volumes'][data['item_types'][i]] for i in
 data['items'])
    volume_occupancy_rate = (total_volume_packed /
 data['bin_capacity']) * 100
    print("\nContainer c001")
    print("Total number of items packed:",
 total_items_packed)
    for j in set(data['item_types']):  # Set to remove
 duplicates
        items_packed_of_type = sum(x[i].solution_value() for
 i in data['items'] if data['item_types'][i] == j)
```

```
            print(f"Items of type {j} packed:
      {items_packed_of_type}")
            print(f"Total volume of items packed:
      {total_volume_packed}")
            print(f"Volume occupancy rate:
      {volume_occupancy_rate:.2f}%")
            print(f"Time = {solver.WallTime()} milliseconds")
        else:
            print("The problem does not have an optimal solution.")

if __name__ == "__main__":
    main()
```

7.2 Implementation py132 listing

```
import time
from ortools.linear_solver import pywraplp

print("Script 132")
"calculate volume, calculate and pring 3D positioning summary"
"print volume calculation summary"

def read_box_data(filename):
    """Read box data from a file."""
    box_data = []
    with open(filename, 'r') as file:
        for line in file:
            parts = line.strip().split()
            if len(parts) == 4:
                l, w, h, q = map(int, parts)
                box_data.append((l, w, h, q))
    return box_data

def calculate_maximum_packing(box_data, container_volume):
    start_time = time.time()  # Start the timing for execution

    # Create solver
    solver = pywraplp.Solver.CreateSolver("SCIP")
    if not solver:
        print("Could not create solver.")
        return None

    # Variables for each type of box
    x = [solver.IntVar(0, q, f'x[{i}]') for i, (_, _, _, q) in
     enumerate(box_data)]

    # Total volume constraint
    total_volume = sum((l * w * h) * x[i] for i, (l, w, h, q) in
     enumerate(box_data))
    solver.Add(total_volume <= container_volume)
```

```
    # Objective: maximize the number of items packed.
    solver.Maximize(solver.Sum(x))

    status = solver.Solve()
    if status == pywraplp.Solver.OPTIMAL:
        results = [int(x[i].solution_value()) for i in
     range(len(x))]
        total_items = sum(results)
        total_packed_volume = sum((l * w * h) * results[i] for i,
     (l, w, h, _) in enumerate(box_data))
        volume_occupancy_percentage = (total_packed_volume /
     container_volume) * 100

        execution_time = time.time() - start_time  # Calculate
     execution time

        print(f"Total items packed: {total_items}")
        for i, qty in enumerate(results):
            print(f"Items from category {i} packed: {qty}")
        print(f"Total volume packed: {total_packed_volume}")
        print(f"Volume occupancy percentage:
     {volume_occupancy_percentage:.2f}%")
        print(f"Execution time: {execution_time:.3f} seconds")  #
     Print execution time

        return results
    else:
        execution_time = time.time() - start_time  # Calculate
     execution time if not optimal
        print(f"Execution time: {execution_time:.3f} seconds")  #
     Print execution time

    return None

def fit_box(container_dims, box_dims, occupied_positions):
    start_time = time. time();
    """Attempt to place the box in the first available position
     in the container."""
    for x in range(0, container_dims[0] - box_dims[0] + 1,
     box_dims[0]):
        for y in range(0, container_dims[1] - box_dims[1] + 1,
     box_dims[1]):
            for z in range(0, container_dims[2] - box_dims[2] +
     1, box_dims[2]):
                new_pos = (x, y, z)
                if can_place_box(new_pos, box_dims,
     occupied_positions, container_dims):
                    """execution_time = time.time() - start_time
                    with open("Output.txt", "w") as text_file:
                        #text_file.write(new_pos, box_dims)
                        #text_file.write(execution_time)
```

```
                        print(new_pos, box_dims, file=text_file)
                        print(f"Execution time:
      {execution_time:.2f} seconds", file=text_file)"""
                    #print(box_dims)
                    return new_pos
    return None

def can_place_box(position, box_dims, occupied_positions,
     container_dims):
    (x, y, z) = position
    (bx, by, bz) = box_dims
    if x + bx > container_dims[0] or y + by > container_dims[1]
     or z + bz > container_dims[2]:
        return False
    for pos, dims in occupied_positions.items():
        if not (x >= pos[0] + dims[0] or x + bx <= pos[0] or
                y >= pos[1] + dims[1] or y + by <= pos[1] or
                z >= pos[2] + dims[2] or z + bz <= pos[2]):
            return False
    return True

def main():
    start_time = time.time()
    container_dims = (1180, 230, 265)  # Container dimensions

    filename = "data_sets/ds6.txt"
    box_data = read_box_data(filename)
    if box_data:
        print("Box data loaded successfully:")
        for data in box_data:
            print(data)
    else:
        print("No box data loaded.")

    container_volume = container_dims[0] * container_dims[1] *
     container_dims[2]
    max_packing = calculate_maximum_packing(box_data,
     container_volume)

    occupied_positions = {}
    box_type_counts = [0] * len(box_data)
    total_volume = 0

    for index, (l, w, h, _) in enumerate(box_data):
        box_dims = (l, w, h)
        quantity_to_place = max_packing[index] if max_packing
     else 0
        for _ in range(quantity_to_place):
            position = fit_box(container_dims, box_dims,
     occupied_positions)
            if position:
                occupied_positions[position] = box_dims
```

```
                box_type_counts[index] += 1
                total_volume += l * w * h
                #print(f"Box {box_dims} placed at {position}")
            #else:
                #print(f"Failed to place box {box_dims}")

    volume_occupancy_rate = (total_volume / container_volume) *
     100
    execution_time = time.time() - start_time
    print("\nSummary:")
    print(f"Total number of items packed:
     {sum(box_type_counts)}")
    for i, count in enumerate(box_type_counts):
        print(f"Items of type {i} packed: {count}")
    print(f"Total volume of items packed: {total_volume}")
    print(f"Volume occupancy rate: {volume_occupancy_rate:.2f}%")
    print(f"Execution time: {execution_time:.2f} seconds")

if __name__ == "__main__":
    main()
```

7.3 Data set DS1 ds1.txt listing

```
58 58 58 150
92 50 30 100
120 80 60 50
30 30 45 200
75 75 100 25
```

7.4 Data set DS2 ds2.txt listing

```
60 60 60 300
```

7.5 Data set DS3 ds3.txt listing

```
20 20 15 900
30 20 15 1000
12 12 12 1200
14 14 30 1200
12 45 35 500
70 50 12 250
32 60 37 200
85 18 50 200
25 70 55 200
10 15 60 200
```