Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

General Solvers used on Combinatorial Problems

Obecné řešiče pro kombinatorické úlohy

MASTER'S THESIS

Study Program: Open Informatics
Specialization: Data Science

Author: Bc. Tomáš Omasta
Supervisor: Ing. Josef Grus
Year: 2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Omasta Tomáš** Personal ID number: **483740**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Data Science**

II. Master’s thesis details

Master’s thesis title in English:

General Solvers used on Combinatorial Problems

Master’s thesis title in Czech:

Obecné řešiče pro kombinatorické úlohy

Guidelines:

The student will analyze the combinatorial problems of Resource-Constrained Project Scheduling and Rectangle Packing, study the appropriate formulations, and implement reference solvers for these problems. The student will focus on approaches that use general methods such as Constraint programming and Genetic algorithms. The student will compare its results with those reported in the literature (e.g., PSPLIB dataset).
The student will design, implement and verify a system for evaluating algorithms for combinatorial problems. The system will parse specifications of benchmark problem instances, process them using the selected algorithm, and verify, visualize, and export results. The system will support adding new problems and algorithms. The user will only need to add, e.g., a problem-specific heuristic, while the system will provide the reusable parts of the solvers, e.g., evolutionary operators.
The system will be verified using the solving methods student will develop for the studied combinatorial problems.

Bibliography / sources:

Laborie, Philippe, Jérôme Rogerie, Paul Shaw, and Petr Vilím. “IBM Ilog CP Optimizer for Scheduling.” Constraints 23, no. 2 (2018): 210–50. https://doi.org/10.1007/s10601-018-9281-x.
Joshi, Kanchan, Karuna Jain, and Vijay Bilolikar. “A VNS-Ga-Based Hybrid Metaheuristics for Resource Constrained Project Scheduling Problem.” International Journal of Operational Research 27, no. 3 (2016): 437. https://doi.org/10.1504/ijor.2016.078938.
Bortfeldt, Andreas. “A Genetic Algorithm for the Two-Dimensional Strip Packing Problem with Rectangular Pieces.” European Journal of Operational Research 172, no. 3 (2006): 814–37. https://doi.org/10.1016/j.ejor.2004.11.016.

 © ČVUT v Praze, Design: ČVUT v Praze, VIC

Name and workplace of master's thesis supervisor:

Ing. Josef Grus Department of Control Engineering FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **05.09.2023** Deadline for master's thesis submission: **09.01.2024**

Assignment valid until: **16.02.2025**

Ing. Josef Grus
Supervisor's signature

Head of department's signature

prof. Mgr. Petr Páta, Ph.D.
Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt

Student's signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

Prague, ...

Bc. Tomáš Omasta

Acknowledgements

I want to thank my Supervisor, Ing. Josef Grus, for all the little things he has done to make this possible, particularly his patience. My thanks belong to my family and girlfriend for their support, to my friend Ing. Jakub Med for reviewing this thesis, and to FEE CTU for all the joys and sorrows it enriched me with during my studies.

Bc. Tomáš Omasta

Title:
General Solvers used on Combinatorial Problems

Author: Bc. Tomáš Omasta

Study Program: Open Informatics
Specialization: Data Science

Supervisor: Ing. Josef Grus
Department of Control Engineering, CTU FEE
Abstract: This thesis presents a practical framework for Combinatorial Optimization. It contributes to the field by providing an effective means for researchers to explore, analyze, and solve combinatorial problems more efficiently. It is designed to facilitate unified benchmark handling and simplify the integration of custom solvers and their comparison. Prioritizing simplicity and accessibility, it supports Genetic Algorithms and Constraint Programming solvers defined with pymoo and IBM's DOCplex CP Optimizer, respectively. The framework facilitates handling complex problems such as RCPSP, Job Shop Scheduling problems, and Packing problems. It is not intended to provide state-of-the-art solutions but rather to offer a reliable and easy-to-use tool for researchers. Tool is based on modular architecture that makes it easy extend it with new benchmarks, problems or solvers. To demonstrate the features and effectiveness of the framework, we provide a list of experiments, including a report on user testing.
Key words: Combinatorial Optimization, Genetic Algorithms, Constraint Programming, RCPSP, 2D-SP

Název práce:
Obecné řešiče pro kombinatorické úlohy

Autor: Bc. Tomáš Omasta

Abstrakt: Tato práce představuje systém pro ucelenou práci nad problémy v kombinatorické optimalizaci, usnadňuje implementaci nových řešičů a jejich porovnávání. Tomuto oboru a výzkumným pracovníkům přispívá poskytnutím účinného prostředku k efektivnějšímu zkoumání, analýze a řešení kombinatorických problémů. Upřednostňuje jednoduchost a přístupnost, a proto nabízí genetické algoritmy a řešiče programování s omezujícími podmínkami definované pomocí pymoo, resp. optimalizátoru IBM DOCplex CP Optimizer. Systém poskytuje sadu problémů, jako je plánování projektů s omezenými zdroji (RCPSP), Job Shop nebo Packing problém. Jeho cílem není poskytovat nejmodernější řešiče, ale spíše nabídnout spolehlivý a snadno použitelný nástroj pro výzkumné pracovníky. Nástroj je založený na modulární architektuře, která umožňuje snadné rozšíření o nové datové sady, problémy nebo řešiče. Abychom demonstrovali vlastnosti a účinnost systůmu, uvádíme řadu experimentů včetně popisu uživatelského testování.
Klíčová slova: Kombinatorická optimalizace, Genetické algoritmy, Programování s omezujícími podmínkami, RCPSP, Packing problém

Contents

Abbreviations xi

List of Algorithms xii

List of Tables xiii

List of Figures xiv

1 Introduction 1

2 Operations Research Problems 3
- 2.1 Operations Research . . . 3
- 2.2 Combinatorial Optimization . . . 4
- 2.3 Scheduling . . . 4
  - 2.3.1 Graham's notation . . . 5
  - 2.3.2 Resource Constrained Project Scheduling Problem (PS|prec|Cmax) . . . 6
  - 2.3.3 Jobshop Scheduling Problem (J||Cmax) . . . 8
- 2.4 Cutting & Packing . . . 8
  - 2.4.1 Strip packing . . . 9
  - 2.4.2 Bin Packing . . . 10

3 General Optimization Solver 11
- 3.1 Overview . . . 11
- 3.2 Simple user flow . . . 11
- 3.3 Project structure . . . 13
- 3.4 Data structure . . . 14
- 3.5 Parsing . . . 16
- 3.6 Handling instances . . . 17
  - 3.6.1 Benchmark . . . 17
- 3.7 Solvers . . . 17

4 Solvers 19
- 4.1 Constraint Programming . . . 19
  - 4.1.1 IBM ILOG CP Optimizer . . . 20
  - 4.1.2 RCPSP . . . 21
  - 4.1.3 Multi-Modal Resources Constrained Project Scheduling Problem . . . 22
  - 4.1.4 Job Shop Problem . . . 23
  - 4.1.5 2D-Leveled Strip Packing Problem . . . 24

4.1.6 2D Strip Packing Problem Not Oriented . . . 25
4.1.7 2D Strip Packing Problem Oriented . . . 26
4.1.8 1D Bin Packing Problem . . . 27
4.2 Genetic Algorithms . . . 28
4.2.1 Biased Random-Key Genetic Algorithm . . . 29
4.2.2 RCPSP . . . 30
4.2.3 2D Strip Packing . . . 31
4.2.4 1D Bin Packing . . . 34

5 Experiments 35
5.1 User testing . . . 35
5.2 RCPSP . . . 39
5.2.1 Experiment 1: Running Default Solvers Against State-of-the-Art Algorithms . 39
5.2.2 Experiment 2: Impact of Population Sizes on GA Performance . . . 40
5.2.3 Experiment 3: BRKGA Population Composition . . . 41
5.2.4 Experiment 4: Impact of GA Crossover Strategies . . . 41
5.2.5 Experiment 5: Evaluating CP Model Performance . . . 43
5.3 2D Strip Packing Problem . . . 43
5.3.1 Experiment 1: Initial CP and GA Comparison . . . 44
5.3.2 Experiment 2: Time-Constrained CP Models . . . 45
5.3.3 Experiment 3: Comparison of Time-limited GAs with CPs . . . 45
5.3.4 Experiment 4: Hybrid Model Integration . . . 46
5.4 MM-RCPSP . . . 47

6 Conclusion 49
6.1 Evaluating the thesis' accomplishments . . . 49
6.2 Reflection on the Research Process, Limitations, and Future Research . . . 50

Bibliography 51

Appendix I - User Guide 57
A Adding new benchmark data . . . 57
B Adding new parser . . . 58
B.1 Implementing parser . . . 59
B.2 Add parser into framework API . . . 59
C Reusing benchmark data . . . 60
D Introducing new optimization problem class . . . 60
E Adding new solver . . . 61
E.1 Adding new solver paradigm . . . 61
E.2 Adding new solver instance to problem . . . 62
E.3 Adding new solver model . . . 62
F Using API . . . 63
G Running solver . . . 63
H Running multiple solvers . . . 64
I Comparing outputs visually . . . 64
J Comparing solvers performances . . . 65

Appendix II 66
K Sample Visualizations of Problem Solutions . . . 66

L Complete table covering the 2D Strip Packing experiments 67
M Leveled 2D Strip Packing Fitness Function 67

Abbreviations

OR Operations Research

CO Combinatorial Optimization

CP Constraint programming

GA Genetic Algorithm

BRKGA Biased Random-Key Genetic Algorithm

SWO Squeaky Wheel Optimization

RCPSP Resource Constrained Project Scheduling Problem

MM-RCPSP Multi-Mode Resource Constrained Project Scheduling Problem

JSSP Job Shop Scheduling Problem

2D-SP 2D Strip Packing

1D-BP 1D Bin Packing

2D-BP 2D Bin Packing

List of Algorithms

1 Classical Genetic Algorithm 30

2 CONSTRUCT ACTIVE SCHEDULE ([42]) 32

3 Pseudo-code of the SWO packing methodology, changed for GA 33

4 Pseudo-code of the SWO packing methodology ([42]) 34

5 1D Bin Packing Fitness Function 34

List of Tables

5.1 Comparison of default CP and GA solvers with state of the art metaheuristics 40

5.2 GA population size performance comparison 40

5.3 Recommended configurations for BRKGA in RCPSPs 41

5.4 Comparison of different BRKGA configurations on j120.sm RCPSP benchmark 42

5.5 Comparison of different GA crossover strategies on j120.sm RCPSP benchmark 42

5.6 Comparison of different CP timelimits on j120.sm RCPSP benchmark . . 43

5.7 Comparison of 2 GA with different fitness functions and 2 CP models on BKW benchmark 45

5.8 Comparison of different CP timelimits on BWK 2D_SP Problem benchmark 45

5.9 Comparison of initial experiments with time limited GAs on BWK 2D_SP Problem benchmark 46

5.10 Comparison of hybrid GA+CP algorithms with level based and best fit GA heuristic on BWK 2D_SP Problem benchmark 46

6.1 Example of ‘generate_solver_comparison_markdown_table‘ output . . . 65
6.2 Example of ‘generate_solver_comparison_percent_deviation_markdown_table‘ output . . . 65
6.3 Full table covering 2D Strip Packing experiments . . . 67

List of Figures

2.1 Gant chart visualizing an example schedule of 3 jobs J1, J2, J3 each with three operations scheduled on 3 different machines, [12]. . . . 8
2.2 A general view of the rectangular 2D-SPP [17]. . . . 9

3.1 Hierarchy of solvers in the framework. The rightmost solvers visualizes extensibility of the module. . . . 18

4.1 Transitional process between consecutive generations [34] . . . 31
4.2 Strip Packing solver using level based heuristic . . . 33
4.3 Strip Packing solver not using level based heuristic . . . 33

5.1 Comparison of different GA population sizes performances in time . . . 41
5.2 Comparison of different BRKGA population compositions on the performance in time . . . 42
5.3 Comparison of different GA crossover strategies on the performance in time 43
5.4 Strip Packing solver using level based heuristic . . . 47
5.5 Strip Packing solver not using level based heuristic . . . 47

6.1 Example of Job Shop Scheduling Problem solution . . . 64
6.2 Solution of the cv25 RCPSP instance, containing 2 renewable resources . 66
6.3 Solution of the c154_3 MM-RCPSP instance, containing 2 renewable and 2 non-renewable resources . . . 66
6.4 Solution of the BKW3 2D Strip Packing instance . . . 66
6.5 Solution of the abz5 Job Shop Problem instance, containing 10 jobs with 10 tasks to be done on 10 machines . . . 66

Chapter 1

Introduction

Combinatorial Optimization, a crucial field in both theoretical and applied mathematics, plays an integral role in various disciplines, from Operations Research to computer science. This field is dedicated to finding the optimal solution from a finite set of possibilities, often in complex scenarios with a vast array of potential solutions. Its significance extends across various sectors, including transportation, production, and scheduling, establishing itself as a critical component in decision-making and problem-solving processes.

The challenges in Combinatorial Optimization, particularly the complexity of problems and the computational resources required to solve them necessitate the continual development of effective solvers. These algorithms and tools are designed to navigate complex solution spaces efficiently, seeking optimal or near-optimal solutions within practical timeframes.

This thesis aims to develop a comprehensive, modular, and extensible framework that facilitates the integration of multiple research benchmarks, problems, and solvers from various libraries into a single system for Combinatorial Optimization. The goal is to provide a user-friendly environment where users can quickly integrate, run, and compare different custom solvers, focusing on Genetic Algorithms and Constraint Programming on various complex problems, validating the solutions, and visualizing them. The scope of the thesis includes an in-depth study and implementation of reference solvers for Resource-Constrained Project Scheduling (RCPSP), multi-mode RCPSP, Job Shop Scheduling Problems, Strip Packing Problems, and Bin Packing Problems. This approach enhances the understanding of solver capabilities and significantly contributes to the field by providing an efficient means to assess and compare various Combinatorial optimization techniques.

The research methodology of this work adopts a broad rather than a deep approach. The primary focus is to create a framework that is not only extensive in its capabilities but also user-friendly and intuitive to use, which we demonstrate in various parts of this work. This approach aims to enable simple integration of new solvers and problems and inter-solver comparability. In terms of experiments, we emphasize demonstrating the features and potential of the framework, and only then do we compare our RCPSP

and 2D Strip Packing Problem solvers with state-of-the-art solutions. On top of these experiments, we also concluded one user testing to receive feedback on user experience. Through this methodology, the research aims to highlight the framework's ability to serve as a valuable tool in exploring and understanding the landscape of combinatorial problems and their respective solving techniques.

We provide an introduction to the problems covered in the framework in Chapter 2, the framework itself is covered in Chapter 3, and the solvers implemented in the framework are described in Chapter 4. We present the experiments concluded in Chapter 5 and attach the user guide in Appendix I.

Chapter 2

Operations Research Problems

In this section, we will introduce the topics this work deals with. We start with the most broad one: Operations Research, and make our way through describing Optimization Problem and end up with specific Scheduling and Cutting & Packing Problems on which we demonstrate the capabilities of General Optimization Solver. The models used for solving problems outlined in this chapter, including the discussion on their complexity and other algorithms from literature, are described in Chapter 4.

The specific problems explained in this chapter are (Multi-Mode) Resource Constrained Project Scheduling Problem, Job Shop Scheduling Problem, Strip Packing Problem, and Bin Packing Problem.

2.1 Operations Research

Operations Research (OR, [1]) is a mathematical field that sets out to either decide feasibility or to find a best available solution, that is, schedule that maximizes or minimizes the objective criterion (profits, costs, efficiency) given the corresponding constraints hold, employing mathematical and analytical methods. Examples of fields in which OR is used are the following. In production, operations research is applied to minimize material waste. In Scheduling, it ensures acceptable shift schedules that conform to law and collective agreements [2]. In resource allocation, operations research is used for making decision whether the tasks are better assigned with multiple junior workers or only one more skilled worker. In transportation, operations research aims to fulfill demand in the shortest time while minimizing warehouse capacities needed with just-in-time delivery. Formally, this is done by defining an objective function and constraints the solution must conform to. Written as

$$\begin{aligned} \text{Maximize / minimize:} \quad & \text{Objective Function} \\ \text{subject to:} \quad & \text{Constraints} \end{aligned}$$

The model can be either feasible if there is at least one solution that satisfies all the constraints or infeasible otherwise. If a solution is found, it is called optimal if the

objective value is the best possible (maximum or minimum). However, the ability to find these optimal solutions and the time needed depends on the represented model's quality and completeness.

Operations research significantly affects the real-life industry, quantifying practical problems and determining the best solution given real-life constraints. Note that there is still a difference between the theoretical and practical optimal solutions as the models may not capture the whole real-world environment.

Some subfields of OR are Linear Programming and Combinatorial Optimization. In Linear Programming the constraints are defined as linear inequalities and the variable domains are continuous. Whereas for Combinatorial Optimization, the constraints can be much more complex, and the domains of some variables can be discrete.

This work deals with Combinatorial Optimization problems. We describe Combinatorial Optimization and specific branches and problems in the following sections.

2.2 Combinatorial Optimization

Combinatorial Optimization (CO, [3]) is a subfield of OR focused on solving problems characterized by discrete, often finite, domains. Similarly to OR, CO finds an objective solution subjected to specific rules and constraints. Typical applications include Scheduling, which aims to create efficient timetables under legal and contractual constraints, and packing problems, where it looks for the best space placement of items within given limitations.

2.3 Scheduling

Scheduling [4] is an CO discipline focused on the optimal allocation of resources over time. The primary goal is to generate a schedule that optimizes certain objective, such as minimizing total operation time, reducing costs or penalties, or maximizing resource utilization.

Scheduling considers a wide range of constraints like resource capacity constraints, time constraints, resource constraints, precedence constraints, task setup constraints, and others to which the resulting schedule must conform.

- Capacity constraints ensure that the workload does not exceed the capacity of the resources (like machines or workers). For example, a machine can only handle a certain number of tasks simultaneously.

- Time constraints include release dates, deadlines (when the job has to be completed), or due dates (when the job should be completed). Schedules must be designed to complete tasks within the specified time frames.

- Resource constraints refer to the limitation of resources, such as human resources, machinery, or materials. The schedule must account for the availability of these resources.

- Precedence constraints define an order in which specific tasks must be completed. The tasks later in the order can not be started unless the preceding tasks are finished.
- Setup time and cost constraints specify the effect of changing from one task to another.

Scheduling covers multiple sub-problems. To create a notion of possible problems, we show some basic examples. The formal typology is provided later on in the work.

- Single-machine Scheduling: Tasks are scheduled on a single machine or resource. The focus is on optimizing the order of tasks to minimize delays or maximize throughput.
- Parallel-machine Scheduling: Multiple identical or different machines are involved, and tasks must be distributed among them efficiently.
- Job shop Scheduling: Involves multiple machines with different tasks, where each task requires a specific sequence of operations to be performed on different machines.
- Flow shop Scheduling: Similar to Job Shop Scheduling, but with a fixed order of machines for all tasks.

Solutions to Scheduling problems differ significantly. Exact methods guarantee the best possible schedule but are mostly not applicable in practice because of the NP-complex nature of these problems. On the other hand, heuristic and metaheuristic approaches find good solutions within a reasonable timeframe, which is especially important for complex or large-scale problems.

The following section introduces Graham's notation, a classification system for Scheduling problems.

2.3.1 Graham's notation

Graham's notation [5] is a classification system for describing and categorizing different types of machine Scheduling problems that became a standard way of describing them.

Graham's notation classifies Scheduling Problems by "resources | tasks | criterion" configuration, often referred to as "$\alpha|\beta|\gamma$".

Following lines explain the meaning of each.

α describes machine resources. Graham introduces set of options {1 - single machine, P - identical parallel machines, Q - uniform parallel machines, R - unrelated parallel machines, O - open shop, J - job shop, F - flow shop}. Optionally, specifying α_2 as number of resources is permitted.

β describes task characteristics. Some of the task characteristics are:

- *prec* specifying that there are precedence relations,

- *pmtn* specifying that preemption (job splitting) is allowed. Preemption happens when job is interrupted and resumed at a later time

- r_j specifying release dates, the first available time slot on which the activity may start

- $\{p_j, p_L \leq l_j \leq p_U, \}$ specifying uniform processing time, bounded processing time

- $\tilde{d}_j, d_j$ specifying deadline (when activity needs to be finished) and due date (when activity should be finished) of the activity

γ specifies optimality criterion: schedule length C_{max}, completion time $\sum_{j \in J} C_j$, maximum lateness $L_{max} = \max_{j \in J}(C_j - d_j)$, tardiness $T_{max} = \max_{j \in J}(\max(0, C_j - d_j))$.

For example, a Scheduling Problem described as $P|prec, p_j = 1|C_{max}$ would be interpreted as a problem involving identical parallel machines with precedence constraints and unit processing times per job, with the objective of minimizing the makespan.

The Graham's notation unfortunately does not specify α for project scheduling. The extension of Graham's notation for project scheduling is defined in [6]. It extends α with {PS - project scheduling, MPS - multi-mode project scheduling}.

The subsections following this one describe specific Scheduling Problems this work deals with.

2.3.2 Resource Constrained Project Scheduling Problem (PS|prec|Cmax)

Resource Constrained Project Scheduling Problem (RCPSP) is NP-hard problem [7] defined by a set of activities J = {0, 1, ..., n, n + 1} with precedence dependencies. The activities must be scheduled within the limited capacities of resources $R_{k \in K}$, K = {1, ..., k}. Resource capacity amounts to "how much work" can given resource support at a given time in the case of standard RCPSP. Each activity has a processing time p_j and requires specific resources $r_{j,k}$ to be completed.

The goal is to find an optimal schedule that minimizes project completion time (the end time of the terminal activity) C_{max} while respecting resource constraints.

The RCPSP solution - schedule starting and terminating at 0-processing time dummy activities j_0 and j_{n+1} respectively - is a list of activities containing their starting times s_j.

There are multiple specifications of possible RCPSP variants ([8], [9]):

- resource type - the resources themselves may have different constraints. Some of the most well-known are

  - renewable - The resource's capacity is consumed only during the execution of activity requiring that resource. After the activity ends, the resource's capacity is renewed. The sum of resources needed by activities at a particular time t must be lower or equal to the resource's k capacity R_k. A good example

of renewable resources is workers - they finish the job within a predefined period and are free to work on another job.

$$\sum_{j \in J | s_j \leq t < s_j + p_j} r_{j,k} \leq R_k \quad \forall k \in K, \forall t \in \{0, ..., C_{max}\} \tag{2.1}$$

- non-renewable - The part of the resource's capacity consumed by a specific activity j is never renewed. No more activities can use this resource once the activities use all the resource capacity. An excellent example of a non-renewable resource is money in cases when the whole budget is already available at the start of the job.

$$\sum_{j \in J} r_{j,k} \leq R_k \quad \forall k \in K \tag{2.2}$$

- doubly constrained - There are capacity constraints for each given period and the total resource life span. This constraint can hold in cases where the budget is known, but the whole budget is not available at the beginning, but the parts of the budget are rather received throughout the delivery of the project.

$$\sum_{j \in J | s_j \leq t < s_j + p_j} r_{j,k} \leq R_k^{\text{period}} \quad \forall k \in K, \forall period \in periods \tag{2.3}$$

$$\sum_{j \in J} r_{j,k} \leq R_k^{\text{total}} \quad \forall k \in K \tag{2.4}$$

- maximum allowed completion time - The project may have another constraint for maximum allowed completion time T specified. The schedule becomes infeasible for schedules with $C_{max} > T$. In some scenarios, the objective may be to maximize the revenue during the maximum allowed completion time.

$$C_{max} \leq T \tag{2.5}$$

- activity release date r_i and deadline $\tilde{d_j}$ - The constraint specifies the time range when a specified activity can be finished. For example, with a rented-out machine, we know the delivery of the machine and the time when we need to return it; neither is free to be changed.

$$r_j \leq s_j \quad \wedge \quad s_j + p_j \leq \tilde{d_j} \quad \forall j \in J \tag{2.6}$$

- number of activity modes - Activity can have more than one mode with various processing times and resource requirements specified. If multiple modes for a specific activity are specified, the model selects only one mode to be executed which has the. This RCPSP variant is called MM-RCPSP, denoted as MPS|prec|Cmax. A good example is task assignment. One senior executive can do the job quickly, but it will be expensive. On the other hand, a team of junior workers will do it for a longer time but for a cheaper price.

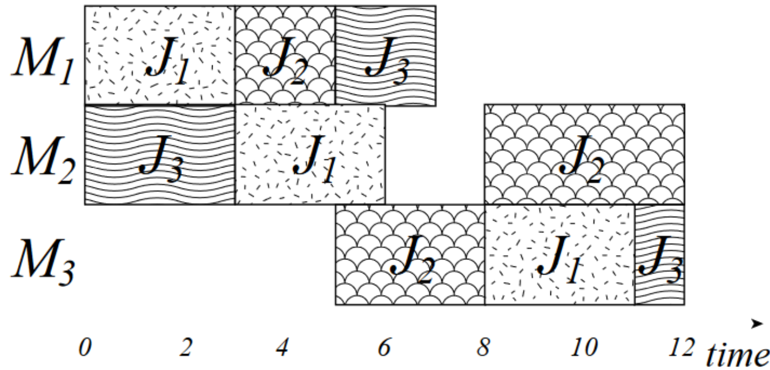


Figure 2.1: Gant chart visualizing an example schedule of 3 jobs J1, J2, J3 each with three operations scheduled on 3 different machines, [12].

- objective function used - By default, the goal is to find the schedule minimizing the project completion time. Other possible objectives may be the maximization of project profit or the minimization of penalties for missed activity deadlines.

2.3.3 Jobshop Scheduling Problem (J||Cmax)

Job Shop Scheduling Problem (JSSP, [10], [11], [12], see an example in Figure 2.1) is NP-hard optimization problem [13] that involves scheduling a finite set of different jobs $J = \{1, ..., n\}$ on a finite set of machines $M = \{1, ..., n\}$. The machines are shared and the jobs are competing for them. Each job i consists of multiple operations $O_i = (O_{i1}, ..., O_{in})$. Each operation needs to be processed during an uninterrupted time period. Such operation O_{ij}, j^{th} of a job i, is assigned a machine $m \in M$, processing time p_{ij}, and must be executed in a correct technological order given by a job operations. Each machine can handle, at most, one activity at a time and no two operations assigned to the same job can run at the same time. The objective is to minimize the makespan of the schedule (set of completion times), i.e., the total time required to complete all jobs.

2.4 Cutting & Packing

Cutting & Packing Problems are two types of optimization problems that are closely related. In case of cutting (cutting coil wire into smaller parts) we want to minimize the cost of the material used and the amount of left over material (maximize the amount of material that is used for the products). In case of packing (packing packages into logistic container) we want to minimize the packing space used (and prevent whatsoever wasted space at all cost).

The Cutting & Packing Problems can be divided by number of characteristics [14], we focus on three ways of sub-problem specification.

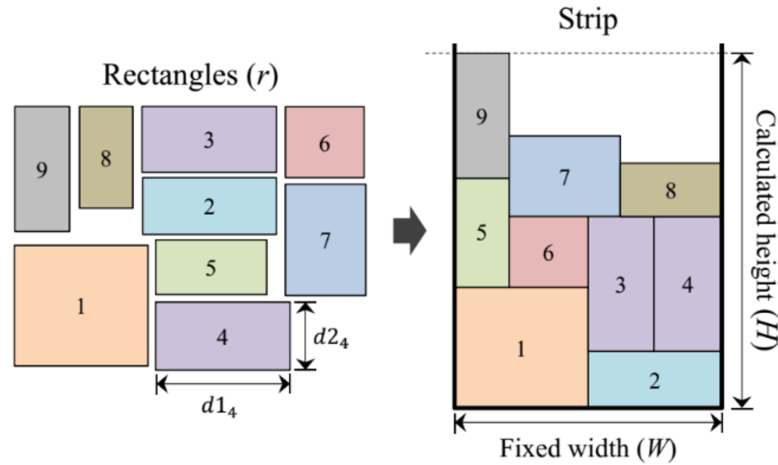


Figure 2.2: A general view of the rectangular 2D-SPP [17].

- Dimensionality - Depending on dimensionality of the input, we differ mainly between 1D (cutting coil wire), 2D (furniture or clothing production) or 3D (container packing). We cover 1D and 2D problems in this work.

- Shape - Different use-cases require different item shapes - rectangles, circles, hexagons, etc. in 2D, or corresponding shapes in 3D. We assume rectangles in 2D for this work.

- Rotation - In higher dimension problems, the orientation matters as well. In basic scenarios, and in this work, we assume original orientation (no rotation allowed) and one rectangular rotation (0°and 90°).

In the following sections, we describe Cutting & Packing sub-problems we deal with in this work: 2D Strip Packing and Bin Packing (both 1D and 2D).

2.4.1 Strip packing

Strip packing [15] is NP-hard [16] hard packing optimization problem where a set of items of different sizes and shapes must be packed into a strip or container of fixed width without overlapping. The goal is to minimize the strip's height or to maximize space utilization. In this paper, we scoped the focus down on subproblems with 2D rectangular items J = {1, ..., n} with height h_j and width w_j specified for each, called 2D Strip Packing (2D-SP). In some cases, we allowed for item rotation. These items are meant to be placed into a strip with a predefined width W, while minimizing its height H. We provide example in Figure 2.2.

2D Level Strip Packing

Two-dimensional level strip packing ([15], [17]) is a variant of 2D-SP Problem where items' bottoms must be aligned on the so-called levels. Levels are virtual shelves placed on certain strip heights. The level's vertical placement is defined by the addition of the tallest rectangle placed on previous level plus the vertical placement of that previous level. The objective is to find an arrangement that minimizes the total strip area required to pack all items. This approach searches much more restricted search space, but often produces solutions with objective value quite close to the similar solutions of the unrestricted packing problem.

2.4.2 Bin Packing

Similarly, Bin packing [15] is a NP-hard [18] packing optimization problem where items of different sizes and shapes must be packed into rectangular bins of fixed width without overlapping. The goal is to minimize the number of bins used. Bin packing sub-problems are

1. 1D Bin Packing (1D-BP) Problem, where the items and bins have the same width and thus only the height is evaluated. Imagine manufacturing example with components of different lengths (2 meters, 4 meters, 3 meters, etc.) needed to be cut from a standard material length of 10 meters.

2. 2D Bin Packing (2D-BP) Problem, where 2D items are meant to be placed into 2D rectangular bins. 2D-BP is similar to Strip Packing, but it introduces constraint on bin height and as a result it expects more than 1 bin to be used

Chapter 3

General Optimization Solver

The following chapter describes the architecture of the General Optimization Solver framework, which is the core part of this work. The discussed framework is available on https://github.com/Omastto1/General-Optimization-Solver.

3.1 Overview

The motivation for this work is to offer the end user a comfortable way to integrate multiple custom solvers for benchmarking into one system. That is: load the desired instance in whatever format, select the solver (or multiple ones), find the solution, validate and visualize the solution, and compare the solution to the reference, other configurations, or possibly other solvers executions from the past.

The framework is divided into 3 logical parts: Parsing, Instance, and Solver.

The first part, Parsing, consists of multiple small parsers, each implemented for specific benchmark file format. They load the input instances and its reference solutions into an internal format defined in the instance part.

The instances form a hierarchy of the core classes representing CO problems. Classes store problem-specific parameters. Through the instance, the user interacts with the whole framework.

In the solvers, there are Constraint Programming and Genetic Algorithm models implemented, each in its own class that requires '_solve' method implementation in place.

For the Constraint Programming solvers and Genetic Algorithm solvers, we used the Python API for IBM ILOG CP Optimizer and python library pymoo, respectively.

3.2 Simple user flow

The interaction with the framework should require only a minimum amount of steps.

Below, we show the example user flow of 1D-BP Problem on a benchmark, solved with a Genetic Algorithm implemented in pymoo that we provide with a simple fitness function.

We provide and discuss the solvers implemented in the framework in the Chapter 4.

1. In Python, import one of ‘load_raw_instance‘ or ‘load_raw_benchmark‘ functions from the ‘src.general_optimization_solver‘, and executes the function to load the input data from the specified instance/benchmark path.

   ```
   from src.general_optimization_solver import load_raw_benchmark

   benchmark = load_raw_benchmark("path_to_benchmark")
   ```

   The framework provides functions to load instances already stored in the internal .json framework format - ‘load_instance‘ and ‘load_benchmark‘. These .json’s dumped by framework are, by default, stored in ‘data/{benchmark_name}‘

2. Specify Genetic Algorithm, fitness function, and termination criterion. In our case we alter only the size of the Genetic Algorithm population. The fitness function assigns the rectangle to the bin specified in the corresponding gene. The Genetic Algorithm run ends after evaluating 100 generations.

   ```
   from pymoo.algorithms.soo.nonconvex.ga import GA

   algorithm = GA(
       pop_size=100
   )

   def fitness_func(instance, x, out):
       bins = {}
       for idx, bin_idx in enumerate(x):
           bin_idx = int(bin_idx)
           bins[bin_idx] = bins.get(bin_idx, 0) + instance.weights[idx]

       out["F"] = len(bins)

       return out

   termination_criteria = ("n_gen", 100)
   ```

3. Specify solver with corresponding Genetic Algorithm, fitness function and termination criteria.

   ```
   solver = BinPacking1DGASolver(algorithm, fitness_func, term_criteria)
   ```

4. Execute the solver on provided benchmark.

   ```
   solver.solve(benchmark)
   ```

5. Optionally, generate markdown table with lower bound deviance for each solver-instance pair, or average deviance for each solver-benchmark pair, and dump data into predefined path inside framework directory.

```
comp1 = benchmark.generate_solver_comparison_markdown_table()
comp2 = benchmark.generate_solver_comparison_percent_deviation\
                _markdown_table()
benchmark.dump(f"example_output_dir")
```

3.3 Project structure

The project uses four main directories - ‘raw_data‘ and ‘data‘ for storing raw and unified .json data, respectively, ‘src‘ for framework implementation and ‘examples‘ providing examples of how to use the framework.

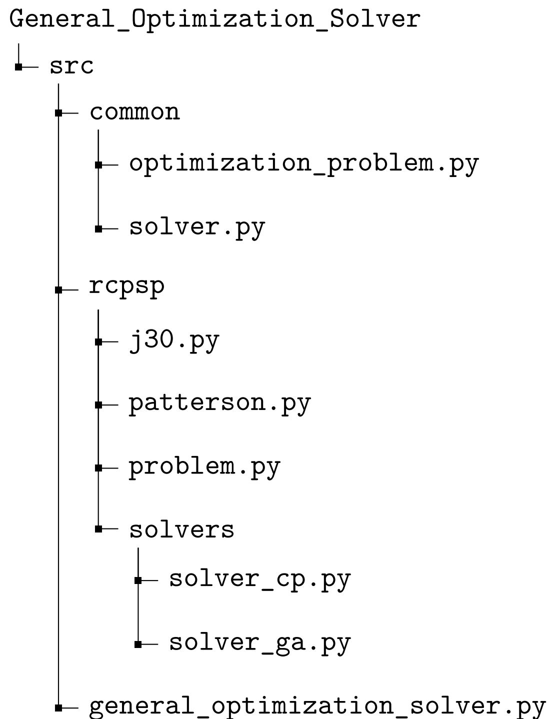
```
General_Optimization_Solver
├─ data
├─ examples
├─ raw_data
└─ src
```

By default, the raw data should be stored in the ‘raw_data/{problem_type}/{benchmark_name}‘ directory. Although it is not forced, we recommend keeping the problem type-benchmark name directory division approach. The .json framework dump is by default saved into ‘data‘ directory. As in previous case, this is not a requirement and the user can specify his own path in the framework directory and pass it as parameter to ‘dump‘ method.

```
General_Optimization_Solver
├─ data
│   └─ RCPSP
│       └─ j30.sm
│           └─ j301_1.json
└─ raw_data
    └─ RCPSP
        └─ j30.sm
            └─ j301_1.sm
```

The source code of the General Optimization Solver framework is divided into modules related to specific optimization problems (RCPSP, 2D-SP, etc.), 'common' module defining abstract Genetic Algorithm, Constraint Programming solver, and generic instance and benchmarks classes, and 'general_optimization_solver.py' which defines the data loading functions as the entry point for the framework.

Each optimization problem module comes with problem specific input data parsers ('j30.py' and 'patterson.py' in this example), problem interface 'problem.py' and 'solvers' directory containing solvers implemented to solve the problem instances.



In the following sections, we will describe the data structure used for handling the problem instances and describe the specific parts of the framework - Parsing, Instances, and Solvers. In the chapter after, we are going to describe the models used for specific OR problems discussed in this work.

3.4 Data structure

In order to sanitize different input formats and handle them in the same way, we have come up with a unified data format for storing the single instances. The instances are stored in the format described below into the 'json' file. The instances are saved either after solver execution or after the '{instance}.dump' function call. The internal instance history is updated every time the user executes a new solver evaluation. Optionally, the user can dump the instance after each solver run. Similarly for benchmarks, all instances

are dumped into corresponding file when prompted.

```
{
    "benchmark_name":
    "instance_name":
    "instance_kind":    // One of "RCPSP", "MM-RCPSP", "JOBSHOP",
                        // "2DSTRIPPACKING", "1DBINPACKING", "2DBINPACKING"
    "data": { ... }     // Problem-specific dictionary of parameters
    "reference_solution": {
        "feasible":     // One of true/false
        "optimum":      // Either known optimum integer of 'null'
        "cpu_time":     // Time (s) to find the optimum if available,
                        // otherwise key not present
        "bounds": {     // Present if no 'optimum' is known.
                        // Stores 'lower' and 'upper' referential bound.
            "lower":
            "upper":
        }
    }
    "run_history": [    // List of dictionaries with instance run history
        {
            "timestamp":        // Timestamp of run
                                // e.g. "2023-06-01 15:37:49.763827"
            "solver_type":      // One of "CP" / "GP"
            "solver_name":      // Custom solver name given by a user
            "solver_config": {
                "TimeLimit":
                "NoWorkers":
                "SolverVersion":
            }
            "solve_status":     // "Optimal" / "Feasible" / "Infeasible"
            "solve_time":       // Time spent solving the instance
                                // (can be limited by for CP)
            "solution_value":   // Best objective value found
            "solution_info":    // solver specific string solution desc.
                                // for a given best solution
            "solution_progress":// List of pairs containing ,
            [                   // the obj. value, obj. value encounter
                [..., ...],     // time, # generation (GA)
                ...
            ]
        },
    ]
}
```

3.5 Parsing

This section discusses parsers. The aim of the parser module is to provide the user with a set of functions parsing different input formats. Each parser contains an input loader and a solution loader which initiates the corresponding problem class.

To add an implemented parser into framework (described in detail in Section B), add a new conditional branch specified for a given parser name in the ‘src.general_optimization_solver.load_raw_instance‘ function and load the instance as follows:

```
load_raw_instance("instance_path", "solution_path", "instance_format")
```

So far, the framework contains these parsers

- RCPSP
  - PSPLIB[19] - j30.sm, j60.sm, j90.sm, and j120.sm under the “j120” format parameter
  - Patterson[20] - sD [21], CV [22], NetRes [23], etc.; use “patterson” when loading
- MM-RCPSP
  - PSPLIB[19] - j10.mm, c15.mm, and c21.mm under the “c15” format parameter, use “c15”
  - MMLIB[24] - MMLIB50, MMLIB100, MMLIB+ - “mmlib” format
- JSSP [1] - “jobshop” format
- 2D-SP Problem [2] (load with “strippacking” format parameter)
  - ZDF[25] - number of elements on the first line, width of strip on the second line, (index, width, heigth) triple on following lines - “strippacking”
  - BKW[26] - Json with “Objects” (1 strip) and “Items” (rectangles) keys - “bkw”
- 1D-BP Problem - first line containing number of items, second line containing capacity of each bin, third and later bins containing items weights
- 2D-BP Problem - extension of 1d bin packing - Second line contains width and height of each bin, each row starting from the third one contains width and height of the items

[1] First row contains the number of jobs and the number of machines pair. Then there are ‘number of jobs‘ rows, with each containing ‘number of machines‘ pairs. Each pair contains the machine that is meant to be run and the processing time of that task. The ‘4 95‘ pair in the third column and the first row means that as the third task for the first job, the fifth machine (we are using zero-based indexing) is meant to run for 95 units of time.

[2] The first row containing the number of units meant to be placed inside a strip, the second row containing the width of a strip and ’number of units’ rows with triple - units index, unit height, unit width

3.6 Handling instances

The instance module is the crucial part for the work with the framework. The only thing not being handled by the instance class is importing. Everything else, like validation, visualization, and result comparison, is handled by the instance class itself.

The instance module is a hierarchy of the core classes representing CO problems. The parent class ‘OptimizationProblem‘, which is abstract instance, is implemented in the ‘s rc.optimization_problem‘ module. There are 6 other modules (‘binpacking1d‘, ‘binpacking2d‘, ‘jobshop‘, ‘mm-rcpsp‘, ‘rcpsp‘ and ‘strippacking2d‘ each containing ‘problem.py‘ module) with classes representing specific problems.

The ‘OptimizationProblem‘ parent class contains general parameters common to all other problem’s classes. These parameters are ‘benchmark_name‘, ‘instance_name‘, ‘instance_kind‘ (“RCPSP”, “MM_RCPSP”, “JOBSHOP”, “2DSTRIPPACKING”, “1DBINPACKING” and “2DBINPACKING”), ’solution’, ’run_history’ and problem specific ’data’ dictionary. Note that the ‘OptimizationProblem‘ class structure clones the internal json container data structure.

Furthermore, the ‘OptimizationProblem‘ class implements the logic for ‘dump‘, which exports the class parameters into json stored in the ‘data/{benchmark_name}/‘ directory, ‘compare_to_reference‘ which compares the objective value of just executed solver and is automatically executed after each solver execution, the logic for skipping the instance solver execution if the last run has already found an optimal solution and logic for updating the ‘run_history‘ variable.

The problem-specific instance inheritants specify the problem type-specific methods - ‘validate‘, ‘visualize‘, and the variables specific for the OR problem. In case of RCPSP, the instance stores number of activities, number of renewable resources and their capacities, list of activity processing times, resource requests and predecessors. The problem specific classes are, of course, further extensible for other problem specifications. This is already used in the ‘MM-RCPSP‘ class which extends ‘RCPSP‘.

3.6.1 Benchmark

In order to work with the benchmarks as a whole, there is also the ‘Benchmark‘ class implemented in the ‘src.common.optimization_problem.py‘ module. ‘Benchmark‘ works as a wrapper of multiple ‘Instance‘ instances. It contains the ’dump‘ method, which saves all the instances into their corresponding locations and methods for comparing solvers performance on the benchmark as a whole, or on specific instances - ‘generate_solver_comparison_markdown_table‘ and ‘generate_solver_comparison_percent_deviation_markdown_table‘. The ‘dump‘ method is called by default after each ‘solve‘ benchmark execution.

3.7 Solvers

Solvers are the place where the integration occurs. So far, the supported solver integrations are either Python API for IBM ILOG CP Optimizer for Constraint Pro-

gramming or the Multi-objective Optimization library pymoo for Genetic Algorithms and similar, see Figure 3.1.

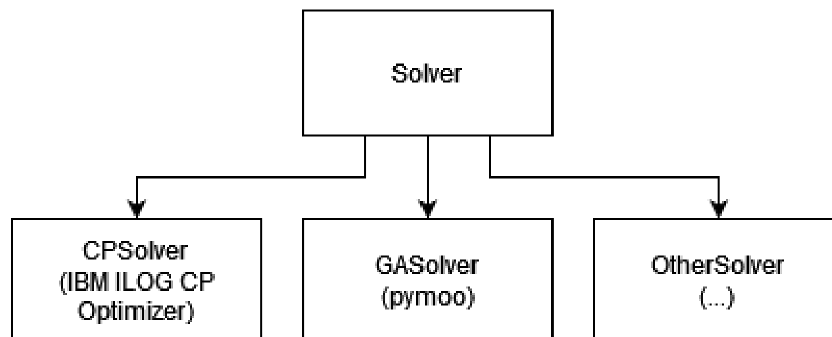


Figure 3.1: Hierarchy of solvers in the framework. The rightmost solvers visualizes extensibility of the module.

In both Constraint Programming and Genetic Algorithm cases, the user provides a model in a '_solve' method defined in a solver class that inherits either from CPSolver of GASolver. The method needs to accept, among other things, instance variable and validate and visualize flags.

```
def _solve(self, instance, validate=False, visualize=False, *args):
    pass
```

In case of Constraint Programming, the built-in '_solve' method defines a Constraint Programming Model with specified variables, constraints and objective method. When instantiating a CPSolver, the user can also specify some Constraint Programming Optimizer parameters - time limit, number of workers and log verbosity.

In case of Genetic Algorithm, the user needs to implement a wrapper around a pymoo 'Problem' class in the '_solve' method. The GASolver class is instantiated with a pymoo algorithm config, fitness function and termination criteria. For example, see Section 3.1.

This has been the architecture chapter. In the next one, We will speak about the individual solver models in detail.

Chapter 4

Solvers

In this chapter, we discuss the specific solver models implemented in the General Optimization Solver framework. The solvers cover the typical Combinatorial Optimization problems (Scheduling and Cutting & Packing in particular) - RCPSP, MM-RCPSP, Job Shop Problem, 2D Strip Packing and 1D Bin Packing.

The framework contains a hierarchy of problem-solver classes. Currently, the framework supports Constraint Programming(CP) and Genetic Algorithm(GA) solvers. CP solvers have been selected thanks to their flexible and intuitive approach to modeling constraints. On the other hand, GA, in most cases, requires only fitness method to be replaced, and as such, is highly modular and allows for great extensibility and interoperability. For CP models, we used the Python API for IBM ILOG CP Optimizer and for GA's we have used python library pymoo.

In the subsequent sections, we will provide an in-depth analysis of each specific solver implemented within our framework. In the first section, we will cover the CP models implemented in the framework, and in the second section, we will present GA fitness functions and configurations.

4.1 Constraint Programming

Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.[27]

Constraint programming (CP, [28]) is a subfield of OR. It is a programming paradigm providing tools and methods to solve problems defined in an almost human language in a declarative way, specifying what constraint the solution should hold rather than how these constraints should be enforced.

Among the methods used by constraint programming are search algorithms (backtracking, FDS [29]), consistency techniques (AC-3), and consistency propagation algorithms.

Contrary to Linear programming, which uses linear inequality constraints only, CP

offers more powerful symbolic constraints - such as ‘endBeforeStart‘ (forcing precedence between two intervals by prohibiting the previous task from starting after the successor task), ‘noOverlap‘ (prohibiting the execution of two specified tasks at the same time) or ‘alternative‘ (providing support for one out of several task choice, or so-called ‘optional‘ tasks).

Overall, instead of specific algorithms, CP relies on search. However, more expressive high-level constraints allow modeling to be more accessible and the number of variables smaller than LP.

In this work, we do not focus on improving methods used by Constraint Programming, but rather on providing models for multiple different problems to the solver described in the next section.

4.1.1 IBM ILOG CP Optimizer

IBM ILOG CP Optimizer [30] is a powerful tool for solving complex optimization problems. It is particularly known for its Constraint Programming capabilities, a method for solving combinatorial problems like scheduling, planning, and resource allocation.

In the last ten years, the CP Optimizer average speed has increased ten-fold [31]. In addition to its simple and intuitive way of modeling the problem, the software enables non-expert users to achieve results comparable with the state-of-the-art methods while being scalable to more than (10^6) activities compared to (10^3) possible with the most classical benchmarks [31].

Ongoing performance improvements of the state-of-the-art combinatorial solvers, such as Gurobi for Integer Linear Programming (ILP) and CP Optimizer for Constraint Programming (CP), make these approaches applicable to increasingly larger problems. Combined with the fact that formulating a given problem using either ILP or CP and running the dedicated solver is usually much less time-consuming than creating a handcrafted branch-and-bound algorithm or a heuristic that performs well enough.

Below we list variable types and high level constraints provided by the solver that are used in our solvers. In the models, we make the constraints bold to highlight them.

The variable types are:

- **Integer variable** represents a single integer. Optionally, the user can specify inclusive lower and upper bounds of variable’s domain.
- **Binary variable** represents a single binary value, which is a special case of an integer variable with a domain {0, 1}.
- **Interval variable** represents an interval of time or a range of consecutive integers. It’s particularly useful in Scheduling Problems to represent the processing time of tasks or activities.

Among the most used constraints are:

- **pulse** - cumulative function between start and end of the interval variable - allows resource constraints to be formulated. Compare the difference between explicit

formulation in the problem definition (Equation 2.1) and the way how the same constraint can be defined imiplicitly without enumerating through the time with the Python API for IBM ILOG CP Optimizer (Equation 4.2),

- **end_before_start** - forces the first interval variable to end before the other one starts (Equation 4.3),
- **alternative** - prohibits any multiple of interval variables given from being scheduled, only one alternative interval variable is allowed to be scheduled (Equation 4.5,
- **presence_of** - binary function returning 1 only if interval variable is scheduled, 0 otherwise (Equation 4.6),
- **no_overlap** - prohibits multiple interval variables from overlapping each other (Equation 4.14),
- **end_of** - returns value of the end of the interval variable, it is used for objective specification or for limiting maximum value of a set of interval variables (Equation 4.1).

Following subsections list different Combinatorial Optimization problem models.

4.1.2 RCPSP

Below we describe CP solver for Scheduling Problems covered in this work, starting with a model of RCPSP, as described in Section 2.3.2.

Model assumes n + 2 (additional 2 activities include 0-processing time dummy start and end activity) activitites $j \in J, J = \{0, 1, ..., n, n+1\}$ with following input variables. Predecessors $predecessors_j$, activity processing times p_j and amount of renewable resource k needed for activity j to be executed $r_{j,k}$. The model uses jobs instead of activities, but the meaning is the same.

CP optimizer model uses following variables to finish the schedule:

job_j interval variable representing time interval when job_j, $j \in J$, is scheduled.

It aims to minimize the maximum completion time of all activities in a project schedule.

$$\textbf{Minimize}: \quad \max_{\forall j \in J}(\textbf{end_of}(job_j)) \tag{4.1}$$

The model contains two key constraints:

1. capacity constraint:

$$\sum_{j \in J} (\boldsymbol{pulse}(job_j, r_{j,k})) \leq R_k \quad \forall k \in R_{renewable} \tag{4.2}$$

which ensures that the resource usage remains lower than the available capacity at any point in time, and

2. precedence constraint:

$$\boldsymbol{end_before_start}(pred, job_j) \quad \forall pred \in predecessors_j, \forall j \in J \tag{4.3}$$

which ensures that activity j cannot start before any predecessor activity is completed. The variables in the model represent the activities in the project, with each activity defined as an interval variable with a specific processing time.

4.1.3 Multi-Modal Resources Constrained Project Scheduling Problem

Following is the CP model of MM-RCPSP.

Model assumes n + 2 (two additional activities include 0-processing time dummy start and end activity) activities $j \in J, J = \{0, 1, ..., n, n+1\}$ with activity modes M_j, predecessors $predecessors_j$, mode processing times $p_{j,m}$ specific for each activity and mode and amount of renewable resource k needed for activity's j mode m to be executed $r_{j,m,k}$. Similarly to RCPSP, the model uses job instead of activity, the meaning is the same.

CP optimizer searches for the following interval variables:

job_j Interval variable representing selected activity's mode schedule, $j \in J$. In final schedule, each activity job_j holds the value of a scheduled activity mode m, $job_mode_{j,m}$.

$job_mode_{j,m}$ Interval variable representing a mode m of a given activity j, $j \in J$, $m \in modes_j$, optional.

The objective of this model is to minimize the maximum completion time of all activities while considering multiple modes for each activity.

$$\textbf{Minimize}: \quad \max_{\forall j \in J}(\boldsymbol{end_of}(job_j)) \tag{4.4}$$

The model includes following constraints:

1. Alternative constraints:

$$\boldsymbol{alternative}(job_j, \{job_mode_{j,m} \mid m \in modes_j\}) \quad \forall j \in J \tag{4.5}$$

which ensure that only one mode is selected for each activity,

2. renewable resource capacity constraints:

$$\sum_{j \in J} \sum_{m \in M_j} (\boldsymbol{presence_of}(job_mode_{j,m}) \cdot \boldsymbol{pulse}(job_mode_{j,m}, r_{j,m,k})) \leq R_k \qquad \forall k \in R_{renewable} \tag{4.6}$$

   In this formula we check the presence of mode (whether it has been scheduled), and count its resource requirements only in that case. Pulse is a cumulative function that accounts only for times in which the activity has been scheduled.

3. Non-renewable resource capacity constraints:

$$\sum_{j \in J} \sum_{m \in M_j} (\boldsymbol{presence_of}(job_mode_{j,m}) \cdot r_{j,m,k})) \leq R_k \qquad \forall k \in R_{non_renewable} \tag{4.7}$$

   Similarly, to the previous case, we count resource requirements of a given mode only if it has been scheduled.

4. Precedence constraints are also defined to keep the order of activity execution correct.

$$\boldsymbol{end_before_start}(pred, job_j) \quad \forall pred \in predecessors_j, \forall j \in J \tag{4.8}$$

4.1.4 Job Shop Problem

Following section describes the CP model of Job Shop Scheduling Problem (Section 2.3.3).

The variables in the model consist of interval variables representing the job operations, with each interval variable having a specific processing time.

$O_{i,j}$ Interval variable for operation assigned to job i, placed on j^{th} position of the job sequence.

The objective is to minimize the maximum completion time for the last operation of each job j, $O_{j,n}$.

$$\textbf{Minimize}: \quad \max_{\forall j \in J}(\boldsymbol{end_of}(O_{j,n})) \tag{4.9}$$

The model includes:

1. Precedence constraint:

$$\boldsymbol{end_before_start}(O_{j,m}, O_{j,m+1}) \quad \forall j \in J, \forall m \in \{1, ..., n-1\} \tag{4.10}$$

   which ensures that the job tasks are scheduled in the correct machine order, and

2. No-overlap constraint:

$$\boldsymbol{no_overlap}(\{o | machine(o) = m, o \in O\}) \quad \forall m \in M \tag{4.11}$$

   which ensures that only one operation is scheduled on a machine at a given time.

4.1.5 2D-Leveled Strip Packing Problem

In the remainder of the section on CP we describe the Cutting & Packing models.

The provided model represents a simplified version of the 2D Strip Packing Problem (Section 2.4.1), known as the 2D Level Strip Packing Problem. This approach was chosen due to the primordial difficulty of modeling the original Strip Packing Problem using constraint programming. Improved 2D Strip Packing Problem models are provided further below. The 2D Level Strip Packing Problem takes a different approach then the standard 2D Strip Packing CP model, employing a similar methodology to the MM-RCPSP. MM-RCPSP selects one of the activity modes to be scheduled, 2D-Leveled Strip Packing Problem model selects the level on which the rectangle should be placed. For 2D-Leveled Strip Packing Problem, we conservatively assume that maximum number of levels is the same as the number of rectangles available J.

$rect_j$	Interval variable representing selected rectangle placement on horizontal coordinate, $j \in J$. In final schedule, each $rect_j$ holds the horizontal coordinate of a rectangle placed on some n^{th} level, $rects_on_level_{j,n}$.
$rects_on_levels_{j,n}$	Interval variable representing a jth rectangle placed on an nth level in a strip. Each rectangle has processing time equal to its width. Optional. $\forall j \in J, \forall n \in J$.

In this model, the objective is to minimize the sum of the maximum heights of the rectangles placed on each level, considering different levels within the strip.

Minimize:

$$\sum_{level \in \{1,...,n\}} \max(\{h_j | \boldsymbol{presence_of}(rects_on_levels_{j,level}) \quad j \in J\}) \tag{4.12}$$

The constraints include

1. alternative constraints:

$$\boldsymbol{alternative}(rect_j, \{rects_on_levels_{j,level} \mid level \in J\}) \quad \forall j \in J \tag{4.13}$$

ensuring that each rectangle is assigned to only one level, and

2. no overlap constraints:

$$\textbf{no_overlap}(\{rects_on_levels_{j,level} \mid j \in J\}) \quad \forall level \in J \tag{4.14}$$

ensuring that rectangles on each level do not overlap with one another.

Additionally, there is a

3. constraint limiting the maximum end position of rectangles on each level to be within the strip width W.

$$\max_{rect \in J}(\boldsymbol{end_of}(rects_on_levels_{rect,level})) \leq W \quad \forall level \in J \tag{4.15}$$

4.1.6 2D Strip Packing Problem Not Oriented

Model of not oriented 2D Strip Packing Section 2.4.1 assumes n rectangular items $j \in J, J = \{1, ..., n\}$ to be placed into strip of width W.

CP Model uses two lists of interval variables as parameters:

$rectangle_j^X$	Interval variable for horizontal coordinates of rectangle j, $j \in J$.
$rectangle_j^Y$	Interval variable for vertical coordinates of rectangle j, $j \in J$.

Model minimizes the maximum heigth of the strip with the following objective function:

$$\textbf{Minimize}: \quad \max_{j \in J}(\boldsymbol{end_of}(rectangle_j^Y)) \tag{4.16}$$

While being constrained by following constraints:

1. No overlap constraint - At least one of width or height do not intersect (in order for two rectangles to intersect, both width and height need to intersect):

$$\boldsymbol{no_overlap}(rectangle_i^X, rectangle_j^X) \vee \boldsymbol{no_overlap}(rectangle_i^Y, rectangle_j^Y)$$
$$\forall i \in \{1, ..., no_elements - 1\}, \forall j \in \{i+1, ..., no_elements\} \tag{4.17}$$

And

2. Width constraint - Constraint limiting the maximum end position of rectangle to be within the strip width W.

$$\max_{j \in J}(\boldsymbol{end_of}(rectangle_i^X)) \leq W \tag{4.18}$$

4.1.7 2D Strip Packing Problem Oriented

Model assumes n rectangular items $j \in J, J = \{1, ..., n\}$.

Model uses six lists of interval variables as parameters, two lists for items with original rotation, two for rotated items, and two interval lists that hold either original rotation rectangle position or rotated one, depending on the rotation that was selected.

$rect_no_rot_j^X$ Interval variable for horizontal coordinates of rectangle j, $j \in J$, optional.

$rect_no_rot_j^Y$ Interval variable for vertical coordinates of rectangle j, $j \in J$, optional.

$rect_rot_j^X$ Interval variable for horizontal coordinates of rotated rectangle j, $j \in J$, optional.

$rect_rot_j^Y$ Interval variable for vertical coordinates of rotated rectangle j, $j \in J$, optional.

$rect_j^X$ Interval placeholder variable for one of original rotation or rotated rectangle horizontal coordinates, $j \in J$.

$rect_j^Y$ Interval placeholder variable for one of original rotation or rotated rectangle vertical coordinates, $j \in J$.

Other than that, the model uses list of binary variables, that represent orientation of each rectangle. This is important, because it locks both oriented and not oriented variants of specific rectangle together. It prohibits rectangle being selected with one rotated variable and the second one not rotated.

O_j binary variable representing orientation of rectangle j, $j \in J$.

And similarly to previous 2D Strip Packing models, this one minimizes height of strip. The slight difference is that in this case we take one of original and rotated rectangle - hidden in alternative variable $rect^Y$.

$$\textbf{Minimize}: \quad \max_{j \in J}(\boldsymbol{end_of}(rect_j^Y)) \tag{4.19}$$

Moreover, constraints similar to previous Strip Packing Problem models need to hold, with a few of additions because of orientations:

1. Only one rotation needs to be selected

$$\begin{gathered}\boldsymbol{alternative}(rect_j^X, [rect_no_rot_j^X, rect_rot_j^X]) \\ \boldsymbol{alternative}(rect_j^Y, [rect_no_rot_j^Y, rect_rot_j^Y]) \quad \forall j \in J\end{gathered} \tag{4.20}$$

   But, since this constraint does not enforce that either pair of original or pair of rotated orientation is selected, we need to enforce it.

2. Select pair of original oriented rectangles, or pair of rotated rectangles

$$\begin{gathered}O_j = 0 => all_of([presence_of(rect_rot_j^X), presence_of(rect_rot_j^Y)]) \\ O_j = 1 => all_of([presence_of(rect_no_rot_j^X), presence_of(rect_no_rot_j^Y)]) \\ \forall j \in J\end{gathered} \tag{4.21}$$

3. For every rectangles pair, assert that the first rectangle in its selected orientation does not overlap with the second rectangle in its selected orientation

$$\begin{gathered}\boldsymbol{no_overlap}(rect_i^X, rect_j^X) \lor \boldsymbol{no_overlap}(rect_i^Y, rect_j^Y) \\ \forall i \in \{1,..., |J|-1\}, \forall j \in \{i+1, ..., |J|\}\end{gathered} \tag{4.22}$$

4.1.8 1D Bin Packing Problem

CP model looks for following decision variables. We assume the same number of bins as is the number of items J in case that every item has the same dimensions as each bin:

$item_bin_assignment_{i,j}$	Binary variable representing in which bin j the item i is placed. $i \in J, j \in J$.
$is_bin_used_j$	Binary variable representing whether the bin j is occupied by any item. $j \in J$.

The CP model finds a solution such that it minimizes the number of bins used:

$$\textbf{Minimize}: \quad \sum_{j \in J} is_bin_used_j \tag{4.23}$$

The model includes these constraints:

1. Each item should be in exactly one bin:

$$\sum_{bin \in J} item_bin_assignment_{item,bin} = 1 \quad \forall item \in J \tag{4.24}$$

2. The sum of weights for items in each bin should not exceed the bin capacity:

$$\sum_{item \in J} (weight_{item} \cdot item_bin_assignment_{item,bin}) \leq cap_{bin} \cdot is_bin_used_{bin} \quad \forall bin \in J \tag{4.25}$$

This is the end of the section covering Constraint Programming models used in the framework. In the following section we will talk about Genetic Algorithms that we implemented. Contrary to this section, where we described the environment with the constraints and let the solver do its job, in the Genetic Algorithms section we either come up with our own naive approaches, or we replicate an algorithm from scientific papers. In case of replicated algorithm, we will also list out a little of previous work as well as the current state-of-the-art methods.

4.2 Genetic Algorithms

Genetic Algorithm (GA, [32]) is a meta-heuristic used in all sorts of optimization problems but is not limited to them only.

Meta-heuristics are higher-level strategies that guide the search process. They provide a general framework designed to be problem-independent and can be applied to various problems. Examples of meta-heuristics are GA, Simulated Annealing, Particle Swarm Optimization, Tabu Search, or others, usually inspired by nature. Meta-heuristics are often provided with problem-specific heuristics. Heuristics are based on domain knowledge and provide rules or guidelines used to improve the quality of the solution and the speed of its retrieval.

The strength of GA lies in its customizability connected with an exhaustive search that GA performs. On the other hand, customizability comes at a significant cost, with many parameters that must be fine-tuned. Fine-tuning is performed when the user needs to evaluate many parameter combinations and selects the best-performing configuration.

While GAs usually do not provide the best results, they are frequently used for either finding baseline solution, where comparison is needed, or for good-enough initial solutions that more mature and fine-tuned algorithms will later use. Moreover, as GAs are extensively modular, they offer an excellent and straightforward addition to the framework.

A GA population consists of individuals, each represented by a chromosome. Chromosomes are composed of genes, which can take various forms, including strings, numbers, or bits. The specific form of these genes is chosen by the user based on the nature

of the problem being addressed. One of the key advantages of using chromosomes is their abstract representation, particularly when represented as bits. This abstraction simplifies the computational process and enhances the speed of operations such as selection, crossover, and mutation.

GAs are a population-based algorithm steered towards the individuals with the best properties, often referred to as survival of the fittest. GA's population is managed with the following operators (modules) that must be specified: selection, crossover, mutation, and evaluation (fitness function)

The user explicitly pressures the algorithm's convergence behaviour by selecting the selection operator. The selection operator selects the individuals that will participate in the crossover to generate a new offspring. Some selection operators are roulette wheel selection, rank selection, tournament selection, or elitist selection.

Provided with a pair of parents, the crossover operator creates new offspring by combining both parent's chromosomes. The well-known crossover operators are (1, 2, K)-point crossover, uniform crossover, simulated binary crossover, or exponential crossover.

The mutation operator alters each individual to support population diversity. It is executed on each individual separately, usually even on separate genes with a given probability. The most known operators swap gene positions or change the gene value altogether. Examples of mutation operators are bit flip mutation, swap mutation, inversion mutation, and Gaussian mutation.

Both crossover and mutation depend on chromosome representation. Some operators are suited for binary chromosomes, some for real-valued chromosomes, and others may be problem-specific.

The evaluation operator uses a given fitness function to assign an individual with an objective value. The evaluation operator is problem and model specific.

Algorithm 1 describes the GA framework workflow. After initially generating the population, the population's individuals are evaluated, pairs of individuals are selected, crossover creates offspring from each pair, and finally, offspring are mutated, creating a new population.

Depending on the specific GA used, the GA's flow can also change, creating various variants of GA. The most frequent additions are probabilities of mutation or some of the parents sustaining in the following generations (elite-preserving).

Overall, GAs frequently find a satisfiable solution. However, the solution quality depends on several parameters and operators. To acquire the best parameter combination, the user needs to evaluate all combinations, which is rarely the case, and in the end, there is a need for a tradeoff to be done between the configuration quality and the time spent finding it.

4.2.1 Biased Random-Key Genetic Algorithm

One of the variants of GAs are Biased Random-Key Genetic Algorithms (BRKGA, [33]).

Compared to general GAs, the BRKGA's chromosomes are represented by a floating point numbers in range [0, 1). The new generations of the individuals are more resistent

Algorithm 1: Classical Genetic Algorithm, [32]

Input : Population size n, Maximum number of iterations, MAX
Output: Global best solution, Y_{best}

1 Generate initial population of n chromosomes $Y_i, i \in \{1, ..., n\}$
2 Set iteration counter t = 0
3 Compute the fitness value of each chromosome
4 **while** $t < MAX$ **do**
5   Select pairs of chromosomes from the population based on fitness value
6   Apply crossover on selected pairs with a given crossover probability
7   Apply mutation on the offsprings with a given mutation probability
8   Replace old population with newly generated population
9   Increment current iteration counter t
10 **end**
11 Return the best-found solution Y_{best}

against premature convergence as in each generation a portion of population is generated at random. This is accomplished with the elitist selection which duplicates a ratio of best performing individuals to the new generation, and thanks to the mutation being replaced with a generation of new, random, individuals.

In the BRKGA, each chromosome is represented by a set of floating-point numbers that fall in the range [0, 1). To counter the premature convergence that may occur in GAs, BRKGA generates *BOT* % mutants in each generation to bring more diversity into the population. Premature convergence occurs when an algorithm converges on a sub-optimal solution early. Furthermore, BRKGA integrates elitist selection, which involves carrying over a specified percentage (denoted as TOP) of the highest-performing individuals from the current generation to the next. This ensures that the qualities of the best solutions are preserved.

The workflow of creating a new generation is defined as follows (also described in Figure 4.1). First, number of the best individuals (also called TOP) is copied into new generation. Then, we generate new individuals (called mutants, number of mutants is defined by BOT number) to prevent convergence. This replaces mutation step. Finally, we select one parent from TOP and one from the old population (including TOP) and execute parametrized uniform crossover (for each gene, we generate number in [0, 1], and select the gene of the first parent if the number is lower than crossover probability C_{prob}, and the gene of the second parent otherwise). Number of individuals created by the crossover is $pop_size - TOP - BOT$.

4.2.2 RCPSP

RCPSP (Section 2.3.2, [35]) is NP-hard problem [7]. As such, survey presented in [36] shows that exact methods have hard time to solve an instance with more than 60 activities. For this reason, the use of heuristics and meta-heuristics (Section 4.2) are

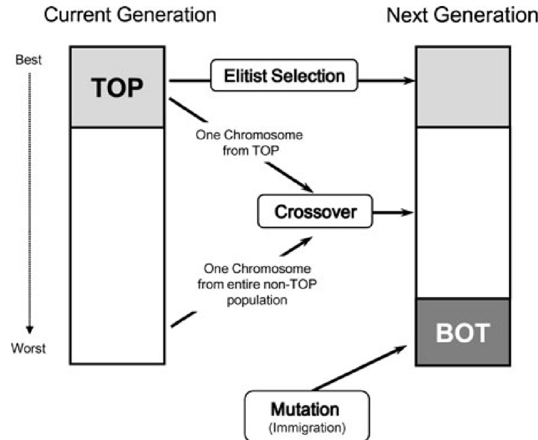


Figure 4.1: Transitional process between consecutive generations [34]

preferred.

Heuristic methods are categorized as either single-pass heuristics (constructive heuristics) or multiple-pass heuristics (improvement heuristics). Examples of meta-heuristic are GA ([37]) and difference evolution (DE) ([38], [39]). Compared to the exact methods, meta-heuristics are much better at solving higher activity instances, but at a cost of multiple hyper parameters that need to be fine-tuned, often demanding much more time from the users.

Among the current state of the art, we take top three algorithms evaluated on j120 benchmark ([19]) in [36], the best two are using GA ([40], [37]) and the third one is using combination of ant colony optimization [41].

In our work, we reproduce the forward pass part of the [34]. The basis of this algorithm is BRKGA meta-heuristic.

The forward procedure is provided in Algorithm 2. The algorithm receives the chromosome representing rectangle priorities ranging from 0 to 1. In each iteration, the algorithm calculates tasks whose precedences have been already scheduled, and available resources. Then it selects the task with the highest priority that hasn't been scheduled yet. For this task, it calculates the earliest finish time considering only precedence constraints, and then refines it considering resource capacities. This earliest feasible finish time is then used to schedule the task. Once all tasks are scheduled, the makespan is computed as the finish time of the last task in the schedule.

4.2.3 2D Strip Packing

2D strip packing has NP-hard [16] complexity. There are propositions for exact algorithms ([43], [44], [45]), but overall the practical solutions rely on heuristics providing a good enough solution.

The work in [46] classifies strip-packing heuristics into construction (Positioning-based, Fitness-based heuristics, Level-based, Profile-based) and improvement heuristics

Algorithm 2: CONSTRUCT ACTIVE SCHEDULE ([42])

Input : chromosome with rectangle priorities with [0, 1] domain
Output: makespan F_{n+1}

```
1 Initialize F_0 = 0, S_0 = 0
2 for g = 1 to n do
3     Calculate D_g, Γ_g and RD_k(t)(k ∈ K, t ∈ Γ_g)
      /* select activity with highest priority                          */
4     j* ← arg max_{j∈D_g}(PRIORITY_j)
      /* compute earliest Finish time (in terms of precedence only) */
5     EF_{j*} ← max_{i∈P_j}(F_i) + p_{j*}
      /* compute earliest Finish time (in terms of precedence and
         capacity)                                                        */
6     F_{j*} ← min{t ≥ EF_{j*} − p_{j*} | r_{j*,k} ≤ RD_k(τ), k ∈ K | r_{j*,k} > 0, τ ∈
        [t, t + p_{j*}]} + p_{j*}
      /* update S_g                                                       */
7     S_g ← S_{g−1} ∪ j^x
8 end
  /* compute makespan (equal to finish time of activity n + 1)     */
9 F_{n+1} = max_{j∈P_{n+1}}(F_j)
```

(Search over sequences, Search over the layout).

One of the oldest positioning-based heuristics is called “Bottom-up left-justified”, or just BL, by [47]. Although being one of the most known and having a simple approach, it has been shown that the results are heavily affected by the order in which the input rectangles are received [48], and the performance can be improved by up to 10% with correct ordering.

Another significant leap in performance was done in [26], which introduces the so-called best-fit (BF) heuristic. BF identifies the lowest available place in the strip, keeps only rectangles with widths smaller than the width of the space, and places the widest rectangle in that space. BF uses the so-called skyline to identify empty places where rectangles should be placed next.

The skyline represents the height of the strip column. As such, the algorithm no longer needs to store complex geometrical representations of the strip but rather the heights of columns only, making the strip handling significantly easier. Skyline implicitly allows for “wasted spaces” (spaces where rectangles can no longer be placed). When a gap (the lowest empty plane in the skyline) is found that is too narrow to place any rectangle in it, it is called a wasted space, and the skyline is raised to have the same height as the lowest neighbor.

Later, bidirectional algorithm modification was presented by [49].

Amongst current state-of-the-art methodologies belong GRASP (meta-heuristic, [50], ISH (heuristic, [51]), and fairly new IN-H* (deterministic tree search procedure, [52]).

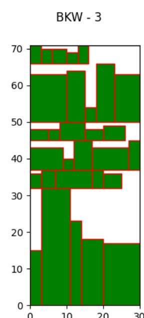


Figure 4.2: Strip Packing solver using level based heuristic

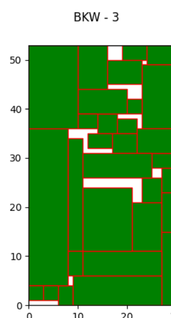


Figure 4.3: Strip Packing solver not using level based heuristic

As the SOTA algorithms are often complex and require many parameters to be set up, the focus of this work lies in showcasing the framework's features that support easy integration and benchmarking. We decided to implement the algorithm from [42] that uses the squeaky wheel optimization approach. Squeaky wheel optimization (SWO, [53]) is an approach that iteratively creates a solution, and in each iteration, it penalizes the items that negatively affect the objective the most. In the subsequent iterations, the items with the biggest penalty are prioritized to be placed first. It turns out to be competitive and sometimes better than one of the SOTA, GRASP.

To showcase the extensible capabilities of the framework with GA that is supported by the framework, we adapted the original algorithm implementation provided in Algorithm 4, [42], which traditionally relied on iterative penalty updates. Our adaptation involves implementing the algorithm as a fitness function within a Genetic Algorithm (GA). Instead of updating penalties after each iteration, we let GA generate the penalties as the chromosome.

In the Algorithm 3, we present the adapted pseudo-code of the algorithm that we use as the fitness function.

Algorithm 3: Pseudo-code of the SWO packing methodology, changed for GA

Input : chromosome with rectangle penalties with [0, 1] domain
Output: skyline

```
1 while exists pieces not packed do
2 |   Find lowest slot 's'
3 |   if s is too narrow for any piece then
4 |   |   Raise s to level of lowest neighbour and merge them both
5 |   else
6 |   |   Find piece with highest penalty, from those that fit into s
7 |   |   Assign this piece to s, next to the tallest neighbour
8 |   end
9 end
```

Algorithm 4: Pseudo-code of the SWO packing methodology ([42])

```
   Output: best skyline found
 1 Initialise each piece's penalty value to zero
 2 while elapsed time < time limit do
 3 |  Initialise empty skyline
 4 |  while exists pieces not packed do
 5 |  |  Find lowest Slot 's'
 6 |  |  if s is too narrow for any piece then
 7 |  |  |  Raise s to level of lowest neighbour and merge them both
 8 |  |  else
 9 |  |  |  Find piece with highest penalty, from those that fit into s
10 |  |  |  Assign this piece to s, next to the tallest neighbour
11 |  |  end
12 |  end
13 |  forall Piece 'p' in current instance do
14 |  |  if p.lower - edge - y - coordinate + p.height > instance lowerbound then
15 |  |  |  p.penalty = p.penalty + p.height
16 |  |  end
17 |  end
18 end
```

4.2.4 1D Bin Packing

To extend the list of examples we provide with the framework and to have some "simple go to" example. We also create a naive fitness function (Algorithm 5) for 1D Bin Packing Problem. The GA generates list of genes as the chromosome and items are assigned to the respective bins specified by the genes. The objective is to minimize the number of bins used. If there are multiple individuals with the same number of bins used, the algorithm prioritizes the one individual that has bigger maximum bin load to motivate the most effective bins assignment.

Algorithm 5: 1D Bin Packing Fitness Function

Input : chromosome, |items| integers in domain [1, ..., |items|]
Output: number of bins, maximum bin load

1 initialize $B_i = 0 \quad (i \in \{1, ..., |items|\})$
2 $B_i = \sum_{gene=i \mid gene \in chromosome} gene$
3 bin_used $= \sum_{b \in B} (b > 0)$
4 max_bin_load $= \max(B)$

This has been solvers chapter, in the next one we present the benchmarking and experiments we have evaluated on the framework.

Chapter 5

Experiments

The experimental chapter of this thesis is structured to demonstrate the capabilities and flexibility of the proposed framework in combinatorial optimization. We divide this section into distinct parts, each focusing on different aspects and applications of the framework to offer a comprehensive understanding of its functionalities.

Initially, we delve into a user testing experiment, a critical research component. In this experiment, a test subject integrates a Maximal Independent Set Problem and custom solvers for that problem into the framework. This exercise is instrumental in evaluating the framework's user experience and adaptability in accommodating new problems and solvers. The insights gained from this user testing are invaluable for refining the framework and ensuring it meets the needs of its users.

Following this, we present experiments on two major problems covered in this thesis: Resource-Constrained Project Scheduling (RCPSP) and 2D-Strip Packing (2D-SP) Problem and one minor experiment on MM-RCPSP. These experiments showcase how the framework handles these complex combinatorial problems, highlighting its problem-solving and solution comparison efficiency. The results and discussions from these experiments will illustrate the practical applications of the framework and its effectiveness in a research context.

The experiments were run on CentOS Linux release 7.5 cluster node, which uses Slurm as a workload manager and job scheduler. The cluster node has Intel Xeon E5-2690 v4 CPU, 14 Cores/CPU; 2.6GHz with 35 MB SmartCache. The RAM was limited to 16 GB. The software used was the following: CPLEX/22.11, GCCcore-10.2.0, and Python-3.8.6. The experiments were run on a single CPU node.

5.1 User testing

We conducted a user testing session to evaluate the user experience and gather feedback on our framework. The participant in this test was Ing. Josef Grus, the author's supervisor. This session focused on integrating the Maximum Independent Set Problem into our framework. The test user selected this problem because we did not cover it in the framework problem set.

Maximum Independent Set Problem [54] assesses to find a maximum independent set in a provided graph $G = (V, E)$ with vertex set $V = \{1, ..., n\}$ and edge set E. Independent set $\tilde{V} \subseteq V$, that the edge set $\tilde{E} \subseteq E$, of the graph induced by $\tilde{V}$, $G(\tilde{V})$, is empty.

An induced graph $G(\tilde{V})$ is such graph that contains a subset of original graph vertices V, $\tilde{V} \subseteq V$ and the edge set of the induced graph $G(\tilde{V})$ contains such edges, that both its adjacent vertices are in $\tilde{V}$.

A Maximum Independent Set is an independent set that is not a subset of any other independent set.

The user had a dataset, CP model, and GA fitness function prepared for the user testing. The user testing consisted of integrating the dataset and CP and GA models into the framework so that the user could simultaneously compare the performance of both approaches on the provided benchmark dataset.

Other than the problem to solve, we asked the user to select a specific benchmark for evaluation, develop a fitness function for the GA, and prepare a CP model for the IBM ILOG CP Optimizer. We placed the code base prepared by the user in the attachments of the digital copy of this work.

During the integration user testing, the user was provided with the user guide (available in the Chapter 6.2) and was able to discuss the potential issues with the author of this work.

In the process of testing, the user implemented the dataset parser:

```
def parse_instance(path: pathlib.Path):
    with open(path) as file:
        _, _, vertices, edges = file.readline().strip().split()
        vertices, edges = int(vertices), int(edges)
        neighbors = [[] for i in range(vertices)]
        for i in range(edges):
            _, u, v = file.readline().strip().split()
            u, v = int(u) - 1, int(v) - 1

            neighbors[u].append(v)
            neighbors[v].append(u)

    return {
        "no_vertices": vertices,
        "no_edges": edges,
        "neighbors": neighbors
    }
```

The user introduced a new class of optimization problems.

```
class MISProblem(OptimizationProblem):
    def __init__(self, benchmark_name, instance_name, data, solution,
```

```
                        run_history):
        super().__init__(benchmark_name, instance_name, "MIS", data,
                        solution, run_history)

        self.no_vertices = data["no_vertices"]
        self.no_edges = data["no_edges"]
        self.neighbors = data["neighbors"]

    ...
```

The user integrated the CP model into a custom CP solver for the Maximum Independent Set Problem.

```
class MISCPSolver(CPSolver):
    def _solve(self, instance, validate=False, visualize=False,
                        force_execution=False):
        if not force_execution and len(instance._run_history) > 0:
            if instance.skip_on_optimal_solution():
                return None, None

        # Specify the model
        model = CpoModel(name="BinPacking")
        model.set_parameters(params=self.params)

        variables = cp.binary_var_list(instance.no_vertices)

        # Specify the model constraints
        for k, neighborhood in enumerate(instance.neighbors):
            for l in neighborhood:
                if k < l:
                    model.add_constraint(cp.logical_or(
                        variables[k] == 0, variables[l] == 0
                    ))
        model.add_constraint(cp.maximize(cp.sum(variables)))

        # Solve the model
        solution = model.solve()
```

As well as implemented a new custom GA solver.

```
class MISGASolver(GASolver):
    def _solve(self, instance, validate=False, visualize=False,
                        force_execution=False):
        class MISProblem(ElementwiseProblem):
```

```
        def __init__(self, no_vertices, neighbors, fitness_func):
            super().__init__(n_var=no_vertices,
                             n_obj=1,
                             n_constr=0,
                             elementwise_evaluation=True)
            self.neighbors = neighbors
            self.no_vertices = no_vertices
            self.fitness_func = fitness_func

        def _evaluate(self, x, out, *args, **kwargs):
            out = self.fitness_func(self, x, out)

    problem = MISProblem(instance.no_vertices, instance.neighbors,
                         self.fitness_func)
    res = minimize(problem, self.algorithm, self.termination,
                         verbose=True, seed=self.seed)
```

The testing was completed in less than two hours and yielded positive user feedback. We expect that with a growing number of integrations done by the user, the duration of the process will be significantly reduced to under one hour as the user becomes more aware of the framework internals.

The resulting pipeline of load, execution, and the results dump looks like the following; note that we have left out the imports on purpose:

```
algorithm = GA(
    pop_size=100,
    n_offsprings=50,
    sampling=PermutationRandomSampling(),
    crossover=OrderCrossover(),
    mutation=NoMutation(),
    eliminate_duplicates=True
)

def construct_solution(instance, x, out):
    available = set(range(instance.no_vertices))
    number_successes = 0

    solution = []
    for k in x:
        if k in available:
            number_successes += 1
            solution.append(k)

            for neighbor in instance.neighbors[k]:
```

```
                if neighbor in available:
                    available.remove(neighbor)

    out["F"] = -number_successes

# SPECIFIC BENCHMARK INSTANCE
instance = load_raw_instance("raw_data/MIS/dataset1/C125.9.mis",
                "", "MIS")

# CP SOLVER EXAMPLE
value, assignment, sol = MISCPSolver(5, 1).solve(instance)
print(value, assignment)

# GA SOLVER EXAMPLE
MISGASolver(algorithm, construct_solution, ("n_gen", 20),
                seed=1).solve(instance)

instance.dump()
```

This section covered user testing. In the next section, we cover extensive testing of RCPSP and 2D-SP Problem solvers and compare them to state-of-the-art solutions.

5.2 RCPSP

In this subsection, we evaluate solvers of the RCPSP on PSPLIB's [19] j120.sm benchmark, focusing on the comparison of our CP model (Section 4.1.2) against the GA approach (model described in Section 4.2.2) and the state-of-the-art metaheuristic algorithms. We evaluate the impact of different time limits on the performance of IBM ILOG CP Optimizer's performance and examine the effects of different crossover strategies and population sizes on the performance of implemented fitness functions for GA and BRKGA.

We used a forward pass from [34] as the fitness function.

If not specified otherwise, we use the following configuration for the GA: Two Point Crossover executed with 90% probability, Polynomial Mutation, Tournament Selection, and duplicate elimination.

5.2.1 Experiment 1: Running Default Solvers Against State-of-the-Art Algorithms

In the initial experiment, we benchmark selected GA [1] and BRKGA [2] configurations against the state-of-the-art metaheuristics [36]. We assess the average deviation from

[1] 120 individuals in each generation , 90% crossover probability, limited to 5000 schedules

[2] TOP=12 individuals, BOT=36 individuals , 72 offsprings 72, C_{PROB}=70%, limited to 5000 schedules

the critical path lower bound over 5000 schedules, highlighting the performance of the algorithms on comparable milestones.

The critical path lower bound [55] is the first, “naive”, lower bound on the project duration. It does not consider the resource capacities to compute the longest available path from the starting activity to the terminal activity. No shorter valid schedule exists because the activities from the longest available path would overlap, so the metric is valid as the lower bound. This metric is often used for datasets without known optimal solutions.

To consider the algorithm comparison in the whole context of our work, we include the result of our CP model, limited by 60 seconds.

solver	**deviation (%)**	**time (s)**
Goncharov and Leonov (Specialist GA(FBI)) [37]	30.50	–
Proon and Jin GA(LS) [40]	31.51	–
CP 60 seconds	32.3	34.1
Chen et al. (Decomposition-based(ACO(SS))) [41]	32.48	–
BRKGA	43.4	245.3
GA	44.0	249.5

Table 5.1: Comparison of default CP and GA solvers with state of the art metaheuristics

This experiment (Table 5.1) demonstrates that while straightforward, a simple, one-pass forward generation heuristic does not yield competitive results in this complex domain. We also highlight that the experiments are implemented in Python, which significantly contributes to at least an order of magnitude worse execution times.

5.2.2 Experiment 2: Impact of Population Sizes on GA Performance

In the following experiments, we benchmark different configurations of GAs and BKRGAs to determine whether configuration changes affect the performance and whether the effect is major or minor.

First, we compare the effect of different population sizes on the performance of GA. We keep the evaluation to the 5000 generated schedules.

solver	**deviation (%)**	**time (s)**
GA 240 offsprings	43.8	254.6
GA 120 offsprings	44	249.5
GA 60 offsprings	45.6	246.8

Table 5.2: GA population size performance comparison

The results in Table 5.2 and Figure 5.1 show an improvement trend with bigger population size. However, with 240 offsprings, the difference is only minor against 120 offsprings, compared to the difference between 60 and 120 offsprings.

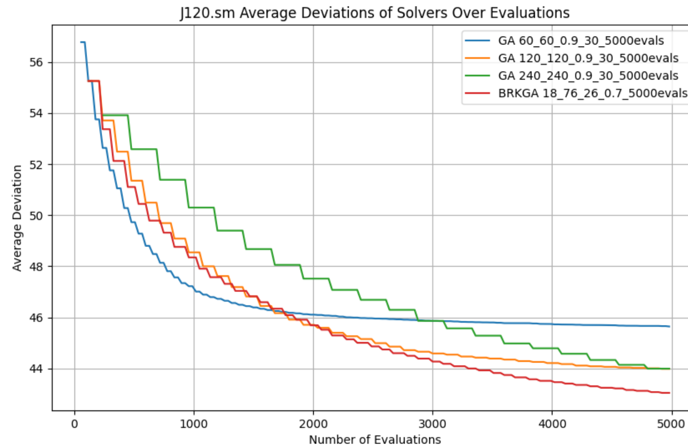


Figure 5.1: Comparison of different GA population sizes performances in time

5.2.3 Experiment 3: BRKGA Population Composition

Contrary to GA experiments, we focus on the population composition of the BRKGA in Experiment 3. For the BRKGA, the authors of [34] recommend the BRKGA parameters to be setup as described in Table 5.3.

Parameter	**Interval**
TOP **(# elites)**	0.10 - 0.20
BOT **(# mutants)**	0.15 - 0.30
C_{Prob}	0.70 - 0.80

Table 5.3: Recommended configurations for BRKGA in RCPSPs

In our experiment, provided in Table 5.4 and visualized in Figure 5.2, we observed the following results on provided setups after 5000 evaluations. The number TOP of elite individuals moved to the new generation affects the performance of the algorithm, high percentage of elite individuals kept probably hurts the diversity, whereas low amount of elite individuals kept limits the convergence. There is no clear implication that BOT number in 15 % - 30 % range significantly affects the performance, but we expect the performance degradation outside this range as the number of completely random individuals affects the convergence and diversity.

5.2.4 Experiment 4: Impact of GA Crossover Strategies

This subsection introduces an experiment focused on the impact of varying crossover strategies on the GA performance on RCPSPs. We test the following crossover strategies: Single Point Crossover, Two Point Crossover, Simulated Binary Crossover (SBX),

solver	deviation (%)	time (s)
TOP 15%, BOT 22.5%	42.5	243.0
TOP 25%, BOT 15%	42.5	245.3
TOP 15%, BOT 22.5%	43.0	242.1
TOP 25%, BOT 20%	43.1	245.3
TOP 10%, BOT 30%	43.4	245.3

Table 5.4: Comparison of different BRKGA configurations on j120.sm RCPSP benchmark

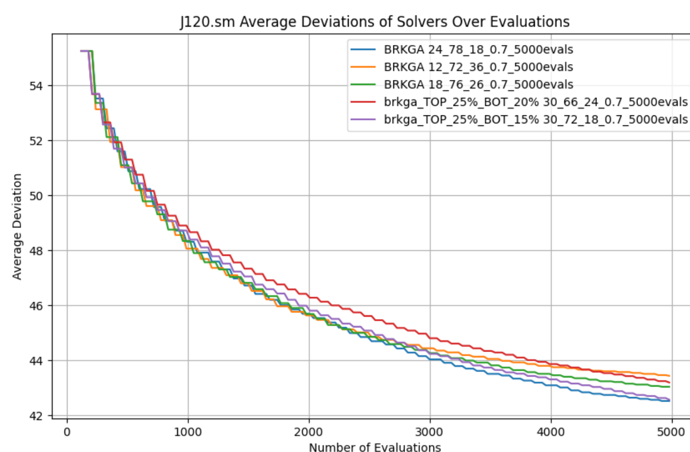


Figure 5.2: Comparison of different BRKGA population compositions on the performance in time

Uniform Crossover, and Half Uniform Crossover. Each GA has 120 individuals in each generation and we compare the performance after 5,000 evaluations.

solver	deviation (%)	time (s)
GA HalfUniformCrossover	42.4	249.2
GA UniformCrossover	42.5	249.2
GA SBX	42.8	249.2
GA TwoPointCrossover	44.0	249.5
GA SinglePointCrossover	44.9	249.6

Table 5.5: Comparison of different GA crossover strategies on j120.sm RCPSP benchmark

The tested crossover strategies, evaluation provided in Table 5.5 and in-time progress visualized in Figure 5.3, show a compelling trend. Going from the worst-performing crossover strategy to the best-performing one, the complexity of the crossover strategy increases, possibly influencing offspring diversity and, thus, the GA's performance. In the context of Single Point Crossover and Two Point Crossover, the crossover combines two or three parts of the parents, which may limit the diversity in the offspring generation.

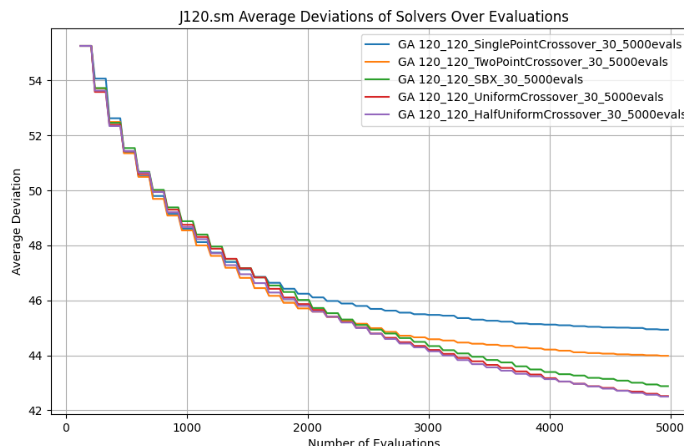


Figure 5.3: Comparison of different GA crossover strategies on the performance in time

The Simulated Binary Crossover is known for its exploration and exploitation balance, maintaining more diversity while improving the objective. Finally, the Uniform and Half Uniform Crossover strategies show the best performance as the crossover strategies generate offspring as a random, gene-level combination of both parents.

5.2.5 Experiment 5: Evaluating CP Model Performance

In the last experiment, we extend the CP model to different time limits to determine how much the performance changes.

solver	deviation (%)	time (s)
CP 600s	30.9	324.1
CP 180s	31.5	99.6
CP 60s	32.3	34.1
CP 15s	34.6	9.1

Table 5.6: Comparison of different CP timelimits on j120.sm RCPSP benchmark

We highlight that the IBM ILOG CP Optimizer, despite being a general solver for scheduling, delivers comparable results to the SOTA algorithms for RCPSP in the 600 seconds time limit configuration (Table 5.6).

5.3 2D Strip Packing Problem

This section provides a comparative study on CP and GA in the 2D-SP Problem context.

While CP historically saw great results in Scheduling, it sees the opposite performance in Cutting & Packing Problems. We assume that the disparity stems from the notable difference between constraint restrictiveness between Scheduling and Cutting & Packing. Take RCPSP, for example; only a handful of scheduled activities can usually be scheduled. Each newly scheduled activity restricts the domain of other variables through the precedence dependencies. In Packing problems on the other hand, placing an item into the strip restricts the solution domain space significantly less compared to the RCPSP. In instances with hundreds of more items, the domain, and the search tree of the Strip Packing Problem is huge, not limiting the CP search enough.

We utilize the BKW benchmark ([26]) to compare the CP and GA methodologies, including hybrid models combining both approaches.

BKW benchmark contains 13 instances ranging from 10 to 3152 instances. These instances are designed to challenge and evaluate the performance of various algorithms in terms of their ability to find an optimal or near-optimal arrangement of items within the strip. The instances were generated by a guillotine cutting and the optimal solutions are known. In our experiments, we use the first 12 instances.

We provide resulting performances in the 3 column table in the paragraphs below. The first column provides the name of the used solver, the second column provides the average deviation from the optimal solution in %, and the last column contains the average time consumed.

If not specified otherwise, we use the following configuration for the : Two Point Crossover executed with 90% probability, Polynomial Mutation, Tournament Selection, and duplicate elimination.

5.3.1 Experiment 1: Initial CP and GA Comparison

In the initial evaluations, we compare "naive" (leveled fitness function, Section M) and "prioritized best fit" (using chromosomes as priorities for placing rectangles. Section 4.2.3) fitness functions for GA [3], and two CP models (limited to 60 seconds and one worker). The first CP model (CP Default Not Oriented) does not allow rectangle rotation, but the second one (CP Default Oriented) allows it. We include a comparison to the algorithm from the literature, which outperforms our models.

The result (Table 5.7) indicates that the assumption of weak CP performance in 2D-SP Problems is correct.

The primary challenge for CP models in the context of Strip Packing Problems lies in their requirement to find at least one good solution that will extensively prune the solution search space. However, it takes a long time to find it, given the size of the packing problem domain space. The oriented CP model demonstrates a two times better solution quality. We assess the spike in the performance to the ability to rotate items, which, in the case of the high items, reduces the objective value drastically.

[3] 200 individuals in a single generation, duplicates elimination, Tournament Selection, Two Point Crossover with 100% probability, Polynomial Mutation with 90% probability and termination after 30 generations without best objective value improvement.

solver	deviation (%)	time (s)
GA + BLF [26]	3.7	–
best-fit GA	19.9	191.8
naive GA	60.2	309.5
CP Default Oriented 60	189.5	55.0
CP Default Not Oriented 60	482.3	55.0

Table 5.7: Comparison of 2 GA with different fitness functions and 2 CP models on BKW benchmark

Compared to CP models, which almost blindly search the domain space, GAs benefit from the deterministic nature of constructive heuristics that leverage the rectangle in the strip placement knowledge (skyline, levels).

The result shows that the best fit heuristic is vastly superior to CP models and the level fitness heuristic. The best fit heuristic shows better convergence and space utilization, with more than one-third of the deviation that level fitness has and is 50% faster to converge.

5.3.2 Experiment 2: Time-Constrained CP Models

After reducing the time limits of CP models, the results in Table 5.8 reveal only a tiny impact on performance. This highlights the CP limitations of managing extensive solution domains without an early, high-quality solution. This strongly contrasts the deterministic and flexible nature of GAs, which benefit from random individual generation covering bigger solution space and subsequent reproduction driving the convergence.

solver	deviation (%)	time (s)
CP Default Oriented 60	189.5	55.0
CP Default Oriented 15	204.0	13.8
CP Default Not Oriented 60	482.3	55.0
CP Default Not Oriented 15	491.1	13.8

Table 5.8: Comparison of different CP timelimits on BWK 2D_SP Problem benchmark

5.3.3 Experiment 3: Comparison of Time-limited GAs with CPs

Here, We compare CP models to the same GAs as in the first experiment but we limit GAs time limit to 60 seconds (Table 5.9). The time-limited GAs perform slightly worse than the non-limited GAs but are significantly superior to the CP models. This further underscores CP's challenges in vast solution spaces where immediate high-quality solutions are inaccessible.

solver	deviation (%)	time (s)
best-fit GA	19.9	191.8
best-fit GA 60sec	20.6	61.8
best fit GA 30 individuals 60sec	21.2	60.2
naive GA	60.2	309.5
naive GA 30 individuals 60sec	68.3	60.0
CP Default Oriented 60	189.5	55.0
CP Default Not Oriented 60	482.3	55.0

Table 5.9: Comparison of initial experiments with time limited GAs on BWK 2D_SP Problem benchmark

5.3.4 Experiment 4: Hybrid Model Integration

As hinted in the previous experiment, the CP models suffer from not having access to a "good enough solution in a short enough time". One possible fix to this problem is to use the hybrid algorithm. For this use case, we employ the leveled fitness function for GA that runs for 20 generations and then CP continues with the initial solution from GA. The great thing about this approach is that the GA heuristic fitness function will assign the rectangles into levels, not providing perfect results. However, its solution will be good enough to prune a significant space of the search space for CP, thus allowing CP to find the solutions in the "good" subspace of the solution domain.

solver	deviation (%)	time (s)
best fit GA 30_1.0_60sec	21.2	60.2
CP Default Not Oriented Hybrid 15	53.1	13.8
CP Default Not Oriented Hybrid 60	44.9	55.0
naive GA 30_1.0_60sec	68.3	60.0
CP Default Oriented Hybrid 15	45.6	13.8
CP Default Oriented Hybrid 60	44.6	55.1

Table 5.10: Comparison of hybrid GA+CP algorithms with level based and best fit GA heuristic on BWK 2D_SP Problem benchmark

More importantly, the CP model can "beat" the leveled fitness function GA with this improvement.

This study (Table 5.10) reveals a fundamental limitation of CP in Strip Packing and broader packing problems: the vast solution space and weak constraint restrictiveness. Contrary to scheduling problems, where each variable assignment significantly restricts the search space, in packing problems, CP's ability to prune the search space is significantly less effective due to each assignment having an impact only on a small subset of the solution domain.

We provide an example of the outputs of running a hybrid solver below. Figure 5.4 shows an output of the first part of the hybrid solver - GA, with the objective value 71. The GA uses constructive level fitness function, placing the rectangles on the levels only,

and its result is inferior. In the second step of the hybrid solver, we feed the output of GA into the CP solver, which outputs the solution with the objective value 53, improving the solution by more than 25 %. We provide the output of the hybrid solver in Figure 5.5.

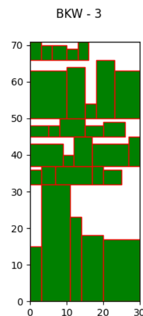


Figure 5.4: Strip Packing solver using level based heuristic

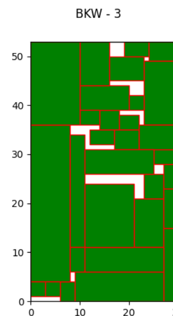


Figure 5.5: Strip Packing solver not using level based heuristic

Even though the CP models do not reach the performance of the classic Strip Packing heuristic, it is visible that with a little "push", the performance of CP models can be significantly improved. This suggests that CP's performance in packing problems can be improved with strategic modifications and the incorporation of GA elements.

The full table of Strip Packing experiments is included in Section L.

5.4 MM-RCPSP

Additionally, we extended both the CP (Section 4.1.3) and BRKGA[4] (Algorithm 2 updated for MM-RCPSP) models to handle the multi-mode variant of RCPSP. Below, we provide a comparison of our models against the state-of-the-art models from the literature benchmarked on the MMLIB50 [24].

First, we compare the GA model against the state-of-the-art models on 1,000 schedules.

solver	**deviation (%)**	**time (s)**
VANP11 [56]	28.2	–
VANP10 [57]	34.1	–
LOVA09 [58]	34.2	–
GA	36.3	65.9

The % deviation was calculated using the critical path lower bound. For the critical path lower bound of MM-RCPSP, we assumed the mode with the minimal processing time to be the "job processing time", as used in Section 5.2.

[4]TOP=15 individuals, BOT=15 individuals , 70 offsprings, C_{PROB}=70%, limited to 5000 schedules

In this experiment, the GA algorithm did not find a feasible solution for 175 out of 540 instances. For the remaining instances, the average deviation from the critical path lower bound was 36.3 %.

Second, we compare CP model performance against the state-of-the-art algorithms on 50,000 schedules.

solver	**deviation (%)**	**time (s)**
CP	23.6	31.4
VANP11 [56]	23.8	–
VANP10 [57]	24.9	–
LOVA09 [58]	26.7	–

In this case, the commercial CP solver has comparable results to the state-of-the-art but with a significantly higher number of evaluations completed.

Chapter 6

Conclusion

This work implements a Python framework supporting cross-paradigm and cross-library solver comparison called General Optimization Solver, available at https://github.com/Omastto1/General-Optimization-Solver. The framework is built to be easily extensible, contains almost 20 different benchmarks, has built-in support for 6 CO problems with more than 10 GA and CP solvers, and provides a solution to compare completely different solvers in one place with ease.

6.1 Evaluating the thesis' accomplishments

The assignment of the thesis was set in two parts. In the first part, the main objective was to analyze the CO problems, their formulations, and solvers, focusing on RCPSP and Rectangle Packing. The solvers analyzed should have used CP or GA methods to provide an extensive foundation for the second part of the assignment. Based on the theoretical research, the thesis assignment expects the author to design, implement, and verify a system for evaluating CO problem solvers. The framework is expected to load different benchmarks, evaluate, validate, visualize the instances, and export the results. Finally, the system is expected to support easy integration of new datasets, problems, algorithms, or problem-specific heuristics in such a way that the majority of the reusable parts are already implemented in the framework. The system is expected to be verified with the methods developed by the author and the results compared to the literature.

In this work, we analyzed CO with a focus on Scheduling Problems (RCPSP, MM-RCPSP, JSSP) and Cutting & Packing Problems (2D-SP Problem and 1D-SP Problem). For each problem, we provided a necessary theory, problem variants, and their applications.

We designed the framework for evaluating algorithms for combinatorial problems. We described the unified data structure the framework uses, how the instances can be processed, verified, visualized and exported. We discussed in detail how each module works and how it can be extended.

With a simple user flow example, we showcased on the example of 1D Bin Packing that the whole benchmark evaluation can have less than 20 lines of code.

For selected Scheduling and Cutting & Packing Problems, we implemented Constraint Programming and Genetic Algorithm solvers and analyzed the current state of art solvers within the corresponding branches.

We verified reusability of the framework with a user testing and compared our solvers with the results reported on PSPLIB, MMLIB and KBW benchmarks.

We included the user guide in Appendix I to provide more in-depth instructions on the framework usage. We forward everyone looking for the most frequent use cases for extending the framework there.

6.2 Reflection on the Research Process, Limitations, and Future Research

We expect that some of the framework's features and overall approaches to some problems will become outdated as the new problems or solver paradigms, whose edge-cases we did not predict, will be added.

For the future research, we recommend conducting user-testing more frequently as the scope of the solving methods will be expanded and there will be more space for generalization. Some of the nice-to-add approaches that can be added in the future are Tabu Search, Ant Colony Optimization, or Particle Swarm Optimization.

We propose to move the benchmarks from the framework repository to another online source to lower the memory requirements of the framework as the number of benchmarks will rise.

Bibliography

1. TAHA, H.A. *Operations Research: An Introduction, Global Edition*. Pearson Education, 2018. ISBN 9781292165561. Available also from: https://books.google.cz/books?id=xb6GEAAAQBAJ.
2. SCHOENFELDER, Jan; BRETTHAUER, Kurt M.; WRIGHT, P. Daniel; COE, Edwin. Nurse scheduling with quick-response methods: Improving hospital performance, nurse workload, and patient experience. *European Journal of Operational Research*. 2020, vol. 283, no. 1, pp. 390–403. ISSN 0377-2217. Available from DOI: https://doi.org/10.1016/j.ejor.2019.10.047.
3. DU, D. Z.; PARDALOS, Panos M.; HU, Xiaodong; WU, Weili. Introduction to Combinatorial Optimization. *Springer Optimization and Its Applications*. 2022. Available also from: https://api.semanticscholar.org/CorpusID:35959893.
4. Classification of Scheduling Problems. In: *Scheduling Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–10. ISBN 978-3-540-69516-5. Available from DOI: 10.1007/978-3-540-69516-5_1.
5. GRAHAM, R.L.; LAWLER, E.L.; LENSTRA, J.K.; KAN, A.H.G.Rinnooy. Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. In: HAMMER, P.L.; JOHNSON, E.L.; KORTE, B.H. (eds.). *Discrete Optimization II*. Elsevier, 1979, vol. 5, pp. 287–326. Annals of Discrete Mathematics. ISSN 0167-5060. Available from DOI: https://doi.org/10.1016/S0167-5060(08)70356-X.
6. BRUCKER, Peter; DREXL, Andreas; MÖHRING, Rolf; NEUMANN, Klaus; PESCH, Erwin. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*. 1999, vol. 112, no. 1, pp. 3–41. ISSN 0377-2217. Available from DOI: https://doi.org/10.1016/S0377-2217(98)00204-5.
7. BLAZEWICZ, J.; LENSTRA, J.K.; KAN, A.H.G.Rinnooy. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*. 1983, vol. 5, no. 1, pp. 11–24. ISSN 0166-218X. Available from DOI: https://doi.org/10.1016/0166-218X(83)90012-4.
8. HARTMANN, Sönke; BRISKORN, Dirk. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*. 2010, vol. 207, no. 1, pp. 1–14. ISSN 0377-2217. Available from DOI: https://doi.org/10.1016/j.ejor.2009.11.005.

9. HARTMANN, Sönke; BRISKORN, Dirk. An updated survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*. 2022, vol. 297, no. 1, pp. 1–14. ISSN 0377-2217. Available from DOI: `https://doi.org/10.1016/j.ejor.2021.05.004`.
10. LENSTRA, J. K. Job Shop Scheduling. In: AKGÜL, Mustafa; HAMACHER, Horst W.; TÜFEKÇI, Süleyman (eds.). *Combinatorial Optimization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 199–207. ISBN 978-3-642-77489-8.
11. XIONG, Hegen; SHI, Shuangyuan; REN, Danni; HU, Jinjin. A survey of job shop scheduling problem: The types and models. *Computers & Operations Research*. 2022, vol. 142, p. 105731. ISSN 0305-0548. Available from DOI: `https://doi.org/10.1016/j.cor.2022.105731`.
12. YAMADA, Takeshi; NAKANO, Ryohei. Job-Shop Scheduling. 2000.
13. GROMICHO, Joaquim A.S.; VAN HOORN, Jelke J.; SALDANHA-DA-GAMA, Francisco; TIMMER, Gerrit T. Solving the job-shop scheduling problem optimally by dynamic programming. *Computers & Operations Research*. 2012, vol. 39, no. 12, pp. 2968–2977. ISSN 0305-0548. Available from DOI: `https://doi.org/10.1016/j.cor.2012.02.024`.
14. DYCKHOFF, Harald. A typology of cutting and packing problems. *European Journal of Operational Research*. 1990, vol. 44, no. 2, pp. 145–159. ISSN 0377-2217. Available from DOI: `https://doi.org/10.1016/0377-2217(90)90350-K`. Cutting and Packing.
15. LODI, Andrea; MARTELLO, Silvano; MONACI, Michele. Two-dimensional packing problems: A survey. *European Journal of Operational Research*. 2002, vol. 141, no. 2, pp. 241–252. ISSN 0377-2217. Available from DOI: `https://doi.org/10.1016/S0377-2217(02)00123-6`.
16. WEI, Lijun; OON, Wee-Chong; ZHU, Wenbin; LIM, Andrew. A skyline heuristic for the 2D rectangular packing and strip packing problems. *European Journal of Operational Research*. 2011, vol. 215, no. 2, pp. 337–346. ISSN 0377-2217. Available from DOI: `https://doi.org/10.1016/j.ejor.2011.06.022`.
17. NEUENFELDT JÚNIOR, Alvaro. *The Two-Dimensional Rectangular Strip Packing Problem*. 2017. Available from DOI: `10.13140/RG.2.2.27201.04965`. PhD thesis.
18. GAREY, Michael R.; JOHNSON, David S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990. ISBN 0716710455.
19. KOLISCH, Rainer; SPRECHER, Arno. PSPLIB - A project scheduling problem library: OR Software - ORSEP Operations Research Software Exchange Program. *European Journal of Operational Research*. 1997, vol. 96, no. 1, pp. 205–216. ISSN 0377-2217. Available from DOI: `https://doi.org/10.1016/S0377-2217(96)00170-1`.

20. PATTERSON, James H. A comparison of exact approaches for solving the multiple constrained resource, project scheduling problem. *Management science*. 1984, vol. 30, no. 7, pp. 854–867.
21. COELHO, José; VANHOUCKE, Mario. New resource-constrained project scheduling instances for testing (meta-)heuristic scheduling algorithms. *Computers & Operations Research*. 2023, vol. 153, p. 106165. ISSN 0305-0548. Available from DOI: `https://doi.org/10.1016/j.cor.2023.106165`.
22. COELHO, José; VANHOUCKE, Mario. Going to the core of hard resource-constrained project scheduling instances. *Computers & Operations Research*. 2020, vol. 121, p. 104976. ISSN 0305-0548. Available from DOI: `https://doi.org/10.1016/j.cor.2020.104976`.
23. VANHOUCKE, Mario; COELHO, José. A tool to test and validate algorithms for the resource-constrained project scheduling problem. *Computers & Industrial Engineering*. 2018, vol. 118, pp. 251–265. ISSN 0360-8352. Available from DOI: `https://doi.org/10.1016/j.cie.2018.02.001`.
24. VAN PETEGHEM, Vincent; VANHOUCKE, Mario. An experimental investigation of metaheuristics for the multi-mode resource-constrained project scheduling problem on new dataset instances. *European Journal of Operational Research*. 2014, vol. 235, no. 1, pp. 62–72. ISSN 0377-2217. Available from DOI: `https://doi.org/10.1016/j.ejor.2013.10.012`.
25. LEUNG, Stephen; ZHANG, Defu. A fast layer-based heuristic for non-guillotine strip packing. *Expert Syst. Appl.* 2011, vol. 38, pp. 13032–13042. Available from DOI: `10.1016/j.eswa.2011.04.105`.
26. BURKE, Edmund K.; KENDALL, Graham; WHITWELL, Glenn. A New Placement Heuristic for the Orthogonal Stock-Cutting Problem. *Oper. Res.* 2004, vol. 52, pp. 655–671. Available also from: `https://api.semanticscholar.org/CorpusID:39746518`.
27. FREUDER, E. In Pursuit of the Holy Grail. *ACM Comput. Surv.* 1996, vol. 28, no. 4es, 63–es. ISSN 0360-0300. Available from DOI: `10.1145/242224.242304`.
28. BARTÁK, Roman. Constraint programming: In pursuit of the holy grail. *Proceedings of WDS99 (invited lecture)*. 1999.
29. VILÍM, Petr; LABORIE, Philippe; SHAW, Paul. Failure-Directed Search for Constraint-Based Scheduling. In: 2015. ISBN 978-3-319-18007-6. Available from DOI: `10.1007/978-3-319-18008-3_30`.
30. LABORIE, Philippe; ROGERIE, Jerome; SHAW, Paul; VILÍM, Petr. IBM ILOG CP optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG. *Constraints*. 2018, vol. 23. Available from DOI: `10.1007/s10601-018-9281-x`.
31. LABORIE, Philippe. *Industrial project and machine scheduling with Constraint Programming*. 2021. Available from DOI: `10.13140/RG.2.2.30470.70726`.

32. KATOCH, Sourabh; CHAUHAN, Sumit Singh; KUMAR, Vijay. A review on genetic algorithm: past, present, and future. *Multim. Tools Appl.* 2021, vol. 80, no. 5, pp. 8091–8126. Available from DOI: `10.1007/S11042-020-10139-6`.

33. RESENDE, Mauricio G.C.; RIBEIRO, Celso C. Biased Ranom-Key Genetic Algorithms: An Advanced Tutorial. In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. Denver, Colorado, USA: Association for Computing Machinery, 2016, pp. 483–514. GECCO '16 Companion. ISBN 9781450343237. Available from DOI: `10.1145/2908961.2926996`.

34. GONÇALVES, José Fernando; RESENDE, Mauricio G. C.; MENDES, Jorge J. M. A biased random-key genetic algorithm with forward-backward improvement for the resource constrained project scheduling problem. *Journal of Heuristics*. 2011, vol. 17, no. 5, pp. 467–486. Available from DOI: `10.1007/s10732-010-9142-2`. Cited by: 66; All Open Access, Green Open Access.

35. LIU, Yongping; HUANG, Lizhen; LIU, Xiufeng; JI, Guomin; CHENG, Xu; ONSTEIN, Erling. A late-mover genetic algorithm for resource-constrained project-scheduling problems. *Information Sciences*. 2023, vol. 642, p. 119164. ISSN 0020-0255. Available from DOI: `https://doi.org/10.1016/j.ins.2023.119164`.

36. PELLERIN, Robert; PERRIER, Nathalie; BERTHAUT, François. A survey of hybrid metaheuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*. 2020, vol. 280, no. 2, pp. 395–416. ISSN 0377-2217. Available from DOI: `https://doi.org/10.1016/j.ejor.2019.01.063`.

37. GONCHAROV, E.N.; LEONOV, V.V. Genetic algorithm for the resource-constrained project scheduling problem. *Automation and Remote Control*. 2017, vol. 78, no. 6, pp. 1101–1114. Available from DOI: `10.1134/S0005117917060108`. Cited by: 29.

38. SALLAM, Karam M.; CHAKRABORTTY, Ripon K.; RYAN, Michael J. A two-stage multi-operator differential evolution algorithm for solving Resource Constrained Project Scheduling problems. *Future Generation Computer Systems*. 2020, vol. 108, pp. 432–444. ISSN 0167-739X. Available from DOI: `https://doi.org/10.1016/j.future.2020.02.074`.

39. CHENG, Min-Yuan; TRAN, Duc-Hoc; WU, Yu-Wei. Using a fuzzy clustering chaotic-based differential evolution with serial method to solve resource-constrained project scheduling problems. *Automation in Construction*. 2014, vol. 37, pp. 88–97. ISSN 0926-5805. Available from DOI: `https://doi.org/10.1016/j.autcon.2013.10.002`.

40. PROON, Sepehr; JIN, Mingzhou. A genetic algorithm with neighborhood search for the resource-constrained project scheduling problem. *Naval Research Logistics (NRL)*. 2011, vol. 58, no. 2, pp. 73–82. Available from DOI: `https://doi.org/10.1002/nav.20439`.

41. CHEN, Wang; SHI, Yan-jun; TENG, Hong-fei; LAN, Xiao-ping; HU, Li-chen. An efficient hybrid algorithm for resource-constrained project scheduling. *Information Sciences*. 2010, vol. 180, no. 6, pp. 1031–1039. ISSN 0020-0255. Available from DOI: https://doi.org/10.1016/j.ins.2009.11.044. Special Issue on Modelling Uncertainty.

42. BURKE, Edmund K.; HYDE, Matthew R.; KENDALL, Graham. A squeaky wheel optimisation methodology for two-dimensional strip packing. *Computers & Operations Research*. 2011, vol. 38, no. 7, pp. 1035–1044. ISSN 0305-0548. Available from DOI: https://doi.org/10.1016/j.cor.2010.10.005.

43. HIFI, Mhand. Exact algorithms for the guillotine strip cutting/packing problem. *Computers & Operations Research*. 1998, vol. 25, no. 11, pp. 925–940. ISSN 0305-0548. Available from DOI: https://doi.org/10.1016/S0305-0548(98)00008-2.

44. LABBÉ, Martine; LAPORTE, Gilbert; MARTELLO, Silvano. Upper bounds and algorithms for the maximum cardinality bin packing problem. *European Journal of Operational Research*. 2003, vol. 149, no. 3, pp. 490–498. ISSN 0377-2217. Available from DOI: https://doi.org/10.1016/S0377-2217(02)00466-6.

45. KENMOCHI, Mitsutoshi; IMAMICHI, Takashi; NONOBE, Koji; YAGIURA, Mutsunori; NAGAMOCHI, Hiroshi. Exact algorithms for the two-dimensional strip packing problem with and without rotations. *European Journal of Operational Research*. 2009, vol. 198, no. 1, pp. 73–83. ISSN 0377-2217. Available from DOI: https://doi.org/10.1016/j.ejor.2008.08.020.

46. OLIVEIRA, José; NEUENFELDT JÚNIOR, Alvaro; SILVA, Elsa; CARRAVILLA, Maria. A surveyonheuristics for the two-dimensional rectangular strip packing problem. *Pesquisa Operacional*. 2016, vol. 36, pp. 197–226. Available from DOI: 10.1590/0101-7438.2016.036.02.0197.

47. BAKER, Brenda; COFFMAN, Ed; RIVEST, Ronald. Orthogonal Packings in Two Dimensions. *SIAM J. Comput.* 1980, vol. 9, pp. 846–855. Available from DOI: 10.1137/0209064.

48. HOPPER, E; TURTON, B.C.H. An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem. *European Journal of Operational Research*. 2001, vol. 128, no. 1, pp. 34–57. ISSN 0377-2217. Available from DOI: https://doi.org/10.1016/S0377-2217(99)00357-4.

49. AŞIK, Önder; ÖZCAN, Ender. Bidirectional best-fit heuristic for orthogonal rectangular strip packing. *Annals of Operations Research*. 2009, vol. 172, pp. 405–427. Available from DOI: 10.1007/s10479-009-0642-0.

50. ALVAREZ-VALDES, R.; PARREÑO, Francisco; TAMARIT, J.M. Reactive GRASP for the strip-packing problem. *Computers & Operations Research*. 2008, vol. 35, pp. 1065–1083. Available from DOI: 10.1016/j.cor.2006.07.004.

51. WEI, Lijun; HU, Qian; LEUNG, Stephen C.H.; ZHANG, Ning. An improved skyline based heuristic for the 2D strip packing problem and its efficient implementation. *Computers & Operations Research*. 2017, vol. 80, pp. 113–127. ISSN 0305-0548. Available from DOI: `https://doi.org/10.1016/j.cor.2016.11.024`.

52. REIS ARRUDA, Vitor Pimenta dos; MIRISOLA, Luiz Gustavo Bizarro; SOMA, Nei Yoshihiro. Rectangle packing with a recursive Pilot Method. *Computers & Operations Research*. 2024, vol. 161, p. 106447. ISSN 0305-0548. Available from DOI: `https://doi.org/10.1016/j.cor.2023.106447`.

53. CLEMENTS, David P.; JOSLIN, David. Squeaky Wheel Optimization. *CoRR*. 2011, vol. abs/1105.5454. Available from arXiv: `1105.5454`.

54. BUTENKO, Sergiy; PARDALOS, Panagote M. *Maximum Independent Set and Related Problems, with Applications*. USA: University of Florida, 2003. PhD thesis. AAI3120100.

55. DEMEULEMEESTER, E.L.; HERROELEN, W.S. Project Scheduling: A Research Handbook. In: Springer US, 2002, chap. 4.1.1.1. International Series in Operations Research & Management Science. ISBN 9781402070518. Available also from: `https://books.google.cz/books?id=fdrX3Ts4O9YC`.

56. PETEGHEM, Vincent; VANHOUCKE, Mario. Using resource scarceness characteristics to solve the multi-mode resource-constrained project scheduling problem. *J. Heuristics*. 2011, vol. 17, pp. 705–728. Available from DOI: `10.1007/s10732-010-9152-0`.

57. PETEGHEM, Vincent Van; VANHOUCKE, Mario. A genetic algorithm for the preemptive and non-preemptive multi-mode resource-constrained project scheduling problem. *European Journal of Operational Research*. 2010, vol. 201, no. 2, pp. 409–418. ISSN 0377-2217. Available from DOI: `https://doi.org/10.1016/j.ejor.2009.03.034`.

58. LOVA, Antonio; TORMOS, Pilar; CERVANTES, Mariamar; BARBER, Federico. An efficient hybrid genetic algorithm for scheduling projects with resource constraints and multiple execution modes. *International Journal of Production Economics*. 2009, vol. 117, no. 2, pp. 302–316. ISSN 0925-5273. Available from DOI: `https://doi.org/10.1016/j.ijpe.2008.11.002`.

59. OLIVEIRA, Oscar; GAMBOA, Dorabela; SILVA, Elsa. Resources for Two-Dimensional (and Three-Dimensional) Cutting and Packing Solution Methods Research. In: 2019, pp. 131–138. Available from DOI: `10.33965/ac2019_201912L016`.

Appendix I - User Guide

In the following chapter we list the most frequent use-cases that the user may encounter. We try to describe the scenarious in the order they may pop up, starting with adding new benchmark data (Section A, Section B), implementing problem classes for these benchmarks (Section D), in cases they are not already available, integrating solvers into benchmark (Section E) and running the whole framework pipeline (Section G). In the end we list some more experienced use-cases: reusing benchmark data (Section C), visualizing output (Section I) or running multiple solvers at once (Section H).

The Framework provides multiple Combinatorial Optimization problems such as: (MM-)RCPSP, JSSP, 2D-SP Problem, 1D-BP Problem and 2D-BP Problem. Framework contains selected benchmarks for provided problems. We offer default Constraint Programming solvers for all problems and default Genetic Algorithm solvers for subset of the problems. Steps to extend each part - benchmark data, problem definition, solver model - are described on following pages.

A Adding new benchmark data

By default, the raw benchmarks are stored in '/raw_data' folder. If you find yourself looking for the benchmark that we definitely should have provided but for whatever reason we did not, don't worry, you are free to extend the set of benchmarks the framework is using. if you are struggling to find the data for Cutting & Packing Problems, we suggest using [59] or https://www.euro-online.org/websites/esicup/data-sets/.

To extend the framework with the benchmark of your choice, paste your benchmark's raw data into corresponding folder in 'raw_data' directory in framework's root folder. There, either create a new folder if you are introducing a new problem (in that case, look into Section D section as well) or select already existing problem folder. In the problem type folder, create a folder corresponding to the benchmark name and paste the benchmark instances into currently created folder. If the benchmark folder is already existing, shout "hooray", because you saved yourself some time, and go get yourself a treat.

Now, depending on your taste, these are further topics that may interest you Section F, Section B or Section C

B Adding new parser

Hello, seeing you here means that you have probably added a benchmark, for which there is no parser implemented, or you did not find one. Either case, this is a list of supported benchmark formats:

- RCPSP
  - PSPLIB[19] - j30.sm, j60.sm, j90.sm, and j120.sm under the "j120" format parameter
  - Patterson[20] - sD [21], CV [22], NetRes [23], etc.; use "patterson" when loading
- MM-RCPSP
  - PSPLIB[19] - j10.mm, c15.mm, and c21.mm under the "c15" format parameter, use "c15"
  - MMLIB[24] - MMLIB50, MMLIB100, MMLIB+ - "mmlib" format
- JSSP [1] - "jobshop" format
- 2D-SP Problem [2] (load with "strippacking" format parameter)
  - ZDF[25] - number of elements, width of strip, (index, width, heigth) - "strippacking"
  - BKW[26] - Json with "Objects" (1 strip) and "Items" (rectangles) keys - "bkw"
- 1D-BP Problem - first line number of items, second line number capacity of each bin, third+ items
- 2D-BP Problem - extension of 1d bin packing - capacity as width, height, each item row contains width, height

If you have not find a format that you are looking for, we will guide you through implementing one.

Implementing a new parser requires two things:

1. implement the parser itself,

[1] First row contains the number of jobs and the number of machines pair. Then there are ‘number of jobs‘ rows, with each containing ‘number of machines‘ pairs. Each pair contains the machine that is meant to be run and the processing time of that task. The ‘4 95‘ pair in the third column and the first row means that as the third task for the first job, the fifth machine (we are using zero-based indexing) is meant to run for 95 units of time.

[2] The first row containing the number of units meant to be placed inside a strip, the second row containing the width of a strip and ’number of units’ rows with triple - units index, unit height, unit width

2. add implemented parser into framework logic - ‘general_optimization_solver.load_raw_instance‘ and instantiate a problem instance with parsed data.

We provide an example how to do both in the following subsections.

B.1 Implementing parser

For writing raw data parser, you need to follow a simple rule. That is, the function (the name is not important) should return a dictionary with important instance features. Lets take a look at a simple 1D bin packing parser example:

```
def load_1dbinpacking(path, verbose):
    with open(path, "r") as file:
        line = file.readline()
        no_rectangles = int(line.strip())

        line = file.readline()
        bin_capacity = int(line.strip())

        weights = []
        for _ in range(no_rectangles):
            weight = int(file.readline().strip())
            weights.append(weight)

    parsed_input = {
        "no_rectangles": no_rectangles,
        "bin_capacity": bin_capacity,
        "weights": weights,
    }

    return parsed_input
```

That’s all! Now lets go and implement the parser!

B.2 Add parser into framework API

If you are new to framework API, go and take a look at Section F. Or not, should be easy as is.

The heart of framework logic is the ‘general_optimization_solver.load_raw_instance‘ function, which depending on ‘format‘ parameter selects which parser to run and which class should be instantiated with the data.

At this point we expect that you yet need to add your format and parser into the logic. To show you an example, this is how logic for loading raw 1d Bin Packing Problem instances without run history and solution is implemented:

```
...
elif format == "1Dbinpacking":
    data = load_1dbinpacking(path, verbose)

    instance = BinPacking1D(
        benchmark_name, instance_name, data, solution={}, run_history=[])
...
```

Nothing hard, is it? Of course given that you are using problem that has already been introduced in the framework. If not, you need to work a little harder. For implementing new optimization problem type. See Section D.

C Reusing benchmark data

The raw benchmarks' data that you have loaded into framework (and ideally run or explicitly dumped) are available in framework's uniform json structure. We list a simplified structure without run history details below.

```
{
    "benchmark_name":
    "instance_name":
    "instance_kind":    // One of "RCPSP", "MM-RCPSP", "JOBSHOP",
                        // "2DSTRIPPACKING", "1DBINPACKING", "2DBINPACKING"
    "data": { ... }     // Problem-specific dictionary of parameters
    "reference_solution": {
        "feasible":     // One of true/false
        "optimum":      // Either known optimum integer of 'null'
        "cpu_time":     // Time (s) to find the optimum if available,
                        // otherwise key not present
        "bounds": { }   // Present if no 'optimum' is known.
                        // Stores 'lower' and 'upper' referential bound.
    }
    "run_history": [ ... ]   // List of dictionaries with historical runs
}
```

To reuse these datas, the framework offers the ‘general_optmization_solver.load_instance‘ function. The function loads dumped data from the path provided.

D Introducing new optimization problem class

Problem classes are a foundation stone of the framework.

Start with creating a new folder named after the optimization problem class in the ‘src‘ folder. Then, move the corresponding parser to the folder (assuming you have

already created one, if not see Section B, then create ‘solvers‘ folder which will contain solvers (see Section E) and create ’problem.py‘ file in the root problem type folder.

In this file, create new class that inherits ‘src.common.optimization_problem.OptimizationProblem‘ and its ‘___init___‘ method accepts following parameters ‘benchmark name‘, ‘instance name‘, ‘data‘, ‘solution‘, ‘run history‘. Then pass this arguments to OptimizationProblem initiator, add problem name as third parameter. Assign instance configuration to instance variables (these instance variables will be later on used in solver).

```
class BinPacking1D(OptimizationProblem):
    def __init__(self, benchmark_name, instance_name, data, solution,
                 run_history) -> None:
        super().__init__(benchmark_name, instance_name, "1DBINPACKING",
                 data, solution, run_history)

        self.bin_capacity = self._data["bin_capacity"]
        self.weights = self._data["weights"]
        self.no_items = len(self._data["weights"])
```

Other than ‘___init___‘ you are meant to implement ‘validate‘ and ‘visualize‘ methods, these are meant to be called from solver.

E Adding new solver

Difficulty of adding a new solver varies can be categorized into 3 groups

- Adding a new solver paradigm - here by paradigm we mean Genetic Algorithms, Constrained Programming, Tabu Search, etc.
- Adding new instance of a solver paradigm not yet implemented for a specific problem - say Constraint Programming model for Travelling Salesman Problem.
- Altering solver model in cases where the paradigm and benchmark solver instance is already implemented - more restrictive constraints for Constraint Programming model, extend model to accept initial solution etc.

E.1 Adding new solver paradigm

Abstract solver and solver paradigm base classes are defined in ‘src.common.solver‘. When implementing new paradigm, keep in mind following best practices.

- Choose appropriate ‘solver_name‘ class variable to provide better understanding of result dumps.
- ‘___init___‘ should accept solver paradigm-specific configuration parameters and optionally model specific ‘solver_name‘. We provide Genetic Algorithm solver as an example below:

```
class GASolver(Solver):
    def __init__(self, algorithm, fitness_func, termination,
                solver_name, seed=None):
        super().__init__()

        self.solver_name = solver_name
        self.algorithm = algorithm
        self.fitness_func = fitness_func
        self.termination = termination
        self.seed = seed
        self.callback= HistoryCallback(algorithm)
```

- Each solver paradigm class needs specialized ‘add_run_to_history‘ method. The method extends the instance’s run history and accepts problem instance, objective value, solution info, solution progress, and execution time as parameters. This method needs to execute the instance’s update_run_history with the corresponding parameters.

E.2 Adding new solver instance to problem

To add a new solver paradigm instance to specific benchmark, add a new module to ‘{problem_type}/solvers‘ folder. The module needs to contain solver inheriting specific paradigm base class. Then, implement ‘_solve‘ method, which accepts ‘instance‘ parameter and ‘validate‘ or ‘visualize‘ optional flags which signalize whether ‘validation‘ or ‘visualization‘ is meant to be run

For running implemented solver, see Section G.

We provide a minimalist example of example of CP solver function below:

```
def _solve(self, instance, validate=False, visualize=False):
    model, model_variables = self.build_model(instance)

    solution = model.solve()
```

E.3 Adding new solver model

To alter solver model (in the sense of changing the model’s behaviour with model parameters) is so far supported in GAs only. To ‘alter‘ a CP model, you need to create a new modul, duplicate the code from original solver and alter it in the new one. To alter Genetic Algorithm, you need to alter the one of pymoo.algorithm, fitness function or termination criteria which are accepted as parameter of GASolver instances.

```
from pymoo.algorithms.soo.nonconvex.ga import GA

algorithm = GA(
    pop_size=100
```

```
)

def fitness_func(instance, x, out):
    bins = {}
    for idx, bin_idx in enumerate(x):
        bin_idx = int(bin_idx)
        bins[bin_idx] = bins.get(bin_idx, 0) + instance.weights[idx]

    # Objective: Minimize the number of bins used
    out["F"] = len(bins)

    return out

termination_criteria = ("n_gen", 100)
```

The fitness function needs to accept ‘instance‘, ‘x‘ input and ‘out‘ output dictionary (pymoo specific). The objective value is meant to be stored in out["F"] and constraints in out["G"], constraint with a value less then 0 means the constraint has not been broken.

F Using API

The General Optimization Solver framework offers 4 API endpoints which are meant for loading (raw) instances or (raw) benchmarks. These endpoints return either instance inheriting ‘OptimizationProblem‘ or ‘Benchmark‘ instance. Benchmarks are loaded like this:

```
from src.general_optimization_solver import load_raw_benchmark

benchmark = load_raw_benchmark("raw_data/rcpsp/j120.sm")
```

G Running solver

Running CP solver that uses 10 workers and runs for 60 seconds, solving 1D Bin Packing instance is as easy as:

```
from src.general_optimization_solver import load_raw_instance
from src.binpacking1d.solvers.solver_cp import BinPacking1DCPSolver

instance = load_raw_instance("raw_data/1d-binpacking/scholl/N1C1W1_A.BPP")

BinPacking1DCPSolver(TimeLimit=60, no_workers=10).solve(problem)
```

Depending on solver paradigm, the specific solver accepts either ‘TimeLimit‘ and ‘no_workers‘ parameters for constrained programming (docplex.cp) or ‘algorithm‘, ‘fitness_func‘, ‘termination‘, ‘seed‘ parameters for genetic algorithms (pymoo)

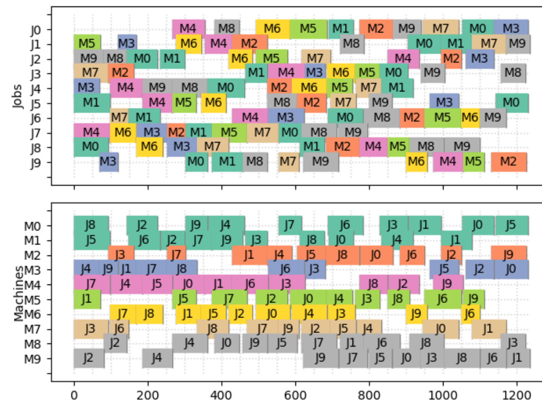


Figure 6.1: Example of Job Shop Scheduling Problem solution

The ‘solve‘ method is same for all solvers and accepts either instance or benchmark, and optionally ‘validate‘, ‘visualize‘ and ‘force_execution‘ flags

H Running multiple solvers

Below we are providing an example of running Constrained Programming and Genetic Algorithm solvers on a 1D Bin Packing Problem instance with no known solution.

```
instance = load_raw_instance("raw_data/1d-binpacking/scholl/N1C1W1_A.BPP",
                              "", "1Dbinpacking")

BinPacking1DCPSolver(TimeLimit=10).solve(instance, validate=False,
                            visualize=False, force_execution=True)
BinPacking1DGASolver(algorithm, fitness_func, ("n_gen", 100), seed=1)
                            .solve(instance)

instance.dump()
```

I Comparing outputs visually

To visualize (and compare) two solutions of a problem instance, user needs to visualize each solution separately and pass an exported solution variables in ‘visualize‘ method to:

```
instance.visualize(model_variables_export)
```

J Comparing solvers performances

To compare performances of solvers, the framework implements two methods, ‘generate_solver_comparison_markdown_table‘ and ‘generate_solver_comparison_percent_deviation_markdown_table‘, both implemented in the benchmark base class. The first one lists an objective value for each solver-instance pair. The second one lists average deviation from the lower bound and average execution time on all instances. We provide examples bellow.

Instance	GA forward 90	BRKGA rcpsp	BRKGA rcpsp forward
j3010_1	43	56	43
j3010_10	43	46	41*
j3010_2	58	65	58
j3010_3	63	77	63
j3010_4	64	72	64

Table 6.1: Example of ‘generate_solver_comparison_markdown_table‘ output

Solver	deviation (%)	time (s)
BRKGA rcpsp	4.6	5.1
BRKGA rcpsp forward	3.8	4.2
GA forward 90	4.6	10.0

Table 6.2: Example of ‘generate_solver_comparison_percent_deviation_markdown_table‘ output

Appendix II

In the rest of the appendix, we attach tables, algorithms, figures and others that we did not find place for in the body of this work, but we find them important to provide.

K Sample Visualizations of Problem Solutions

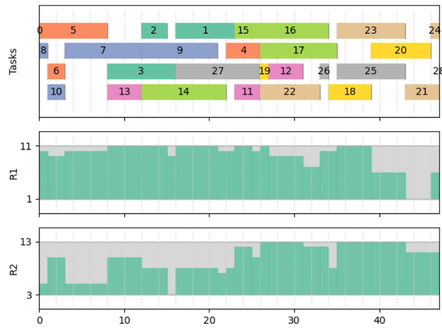


Figure 6.2: Solution of the cv25 RCPSP instance, containing 2 renewable resources

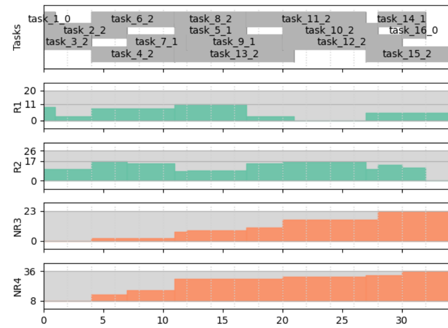


Figure 6.3: Solution of the c154_3 MM-RCPSP instance, containing 2 renewable and 2 non-renewable resources

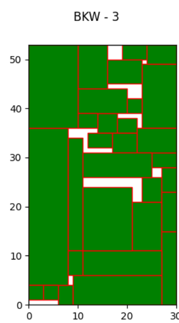


Figure 6.4: Solution of the BKW3 2D Strip Packing instance

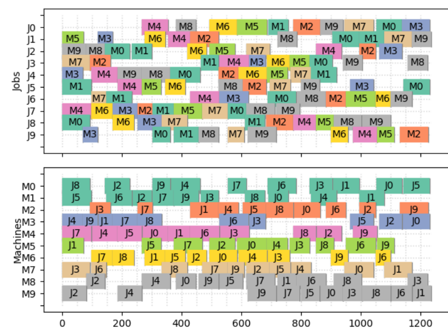


Figure 6.5: Solution of the abz5 Job Shop Problem instance, containing 10 jobs with 10 tasks to be done on 10 machines

L Complete table covering the 2D Strip Packing experiments

solver	deviation (%)	time (s)
best fit GA 200__1.0	19.9	191.8
best fit GA 200__1.0__60sec	20.6	61.8
best fit GA 30__1.0__60sec	21.2	60.2
CP Default Oriented Hybrid 60	44.6	55.1
CP Default Not Oriented Hybrid 60	44.9	55.0
CP Default Oriented Hybrid 15	45.6	13.8
CP Default Not Oriented Hybrid 15	53.1	13.8
naive GA 200__1.0	60.2	309.5
naive GA 30__1.0__60sec	68.3	60.0
CP Default Oriented 60	189.5	55.0
CP Default Oriented 15	204.0	13.8
CP Default Not Oriented 60	482.3	55.0
CP Default Not Oriented 15	491.1	13.8

Table 6.3: Full table covering 2D Strip Packing experiments

M Leveled 2D Strip Packing Fitness Function

```
def fitness_func(instance, x, out):
    """
    Place rectangles one after each other into levels, return total height
    """
    order = np.argsort(x)
    total_height, current_width, current_height = 0
    rectangles = [None] * instance.no_elements
    for i in order:
        if current_width + instance.rectangles[i]['width'] > instance.strip_width:
            total_height += current_height
            current_width = 0
            current_height = 0

        rectangles[i] = (current_width, total_height)
        current_width += instance.rectangles[i]['width']
        current_height = max(current_height, instance.rectangles[i]['height'])
    total_height += current_height
    out["F"] = total_height

    return out
```