Universitat de Girona
Escola Politècnica Superior

Bachelor's degree in Computer Engineering

DEGREE THESIS

Deep learning improvements for SAT-based solving the Resource-Constrained Project Scheduling Problem (RCPSP)

Author:
Bernat Comas i Machuca

Advisors:
Dr. Jordi Coll Caballero
Dr. Mateu Villaret Auselle

REPORT

Delivered on:
June 2024

Department :
Informatics, Applied Mathematics and Statistics

Project:	Degree Thesis
Document:	Report
Title:	Deep learning improvements for SAT-based solving the Resource-Constrained Project Scheduling Problem (RCPSP)
Author:	Bernat Comas i Machuca
Date:	June 2024

Studies:
Bachelor's degree in Computer Engineering
Universitat de Girona

Supervisor 1:
Dr. Jordi Coll Caballero
Universitat de Girona
Email: jordicoll@udg.edu
Web: Link

Supervisor 2:
Dr. Mateu Villaret Auselle
Universitat de Girona
Email: mateu.villaret@udg.edu
Web: Link

Acknowledgements

I want to express my gratitude to Jordi Coll, my supervisor, for his trust in me and unwavering support.

I also like to thank my supervisor Mateu Villaret, for consistently believing in my abilities and encouraging me to strive for excellence.

Thank you to the whole Logics and Artifical Intelligence group at the University of Girona for making me feel like home.

I also want to express my gratitude to my family and friends for everything they have done for me and for always being there.

And special thanks to my dad wherever you are for always being the light that points in the right direction.

Contents

List of Figures

List of Tables

Chapter 1

Introduction, motivation, purpose and project objectives

1.1 Introduction

The Resource-Constrained Project-Scheduling Problem (RCPSP) is a particular type of scheduling optimization problem that aims to minimize the makespan of a set of activities subject to precedence and resource constraints. This type of problem has a strong application background and is widely present in engineering and construction, software development, manufacturing, and logistics management among others.

In this project, we are going to encode this particular type of scheduling problem to Boolean Satisfiability (SAT) and solve it using a SAT solver. After that, we are going to research how the addition of a deep learning model to assist the solver during the propagation may improve the solving metrics.

Since many different RCPSP SAT encodings exist, during the first part of this project we are going to develop our own RCPSP encoding, specifically for the preemptive variant (PRCPSP), allowing pauses during the execution of activities. Nevertheless one can see that this project can be easily extended to work with any type of RCPSP encoding.

Deep learning is a subset of machine learning that includes algorithms used to model high-level abstractions into data. These models try to simulate the learning process of the human brain and are often known as neural networks. The development of deep learning models to solve SAT instances has received some attention in recent years [1]. However, the aim of the developed model will not be to replace a SAT solver. Instead, its goal will be to assist the solver during the propagation process, preserving its completeness while assisting in its decision-making capabilities, which has not been set out in literature before.

The model developed will be trained using RCPSP-related information about the activities and resources that participate in the problem. This information would otherwise not be considered when solving SAT instances. Therefore, we aim to discover how will this knowledge impact the solving process. To include this information we are going to modify a SAT solver to select the conflicts that use clauses that are the most likely to belong to the refutation tree of the problem. With this, we hope to

improve the solving process by finding earlier more “interesting” and “useful” conflicts.

This project was developed under the supervision of the Logic and Artificial Intelligence research group (LAI) at the University of Girona during 5 months.

1.2 Motivation and Purpose

My interests in both the fields of logic and artificial intelligence had been developing throughout the degree and finally materialized during the fourth year. Due to the reduced attention the topic of deep learning received during the degree, I invested time in reading several books as I decided I wanted to go further into this subject. Laying out the possibilities for this Degree Thesis, my main objective was to find a way to combine these two passions, and the opportunity finally emerged in a conversation with my tutors.

In the first semester of the fourth year, I discovered neural networks in the "Advanced Techniques of Artificial Intelligence" subject, and I implemented my first deep learning model. Ever since then, I have wanted to expand my knowledge of Graph Neural Networks (GNN), and this thesis was the perfect opportunity. During its development, I expect to learn about deep learning and form a basis for my future studies in the field, which will start with a Master’s degree in artificial intelligence in the following year.

Moreover, even before planning the thesis, I had always understood that I wanted to do my final thesis from a research perspective and try to center it around the fields of logic or artificial intelligence, doing something that had not been done before. The first thing I did was get in touch with the Logics and Artificial Intelligence research group and they provided me the tools and knowledge I needed to develop this thesis. Together, we came up with this idea which I instantly knew was the opportunity I had been waiting for.

The Logics and Artificial Intelligence (LAI) research group has been working in the field of encoding scheduling problems using logical constraints for many years. This thesis was fully in its area of interest and expertise, lining up with their current research line, Razonamiento y Aprendizaje (RAP), which has one of its focus in the specialization of SAT solvers for scheduling problems and the use of deep learning.

1.3 Project Objectives

This project’s objectives can be summed up into the following key points:

1. Create a new SAT encoding for the PRCPSP.

2. Create a second SAT encoding for the PRCPSP to solve the problem by reducing it to the RCPSP problem, and using some of the already existing encodings for the RCPSP. This reduction requires decomposing each of the original activities into a number of new activities, hence potentially increasing the size of the encodings.

3. Build a suitable GNN model to try to improve SAT-based PRCPSP resolution metrics and optimize its hyper-parameters.

4. Train and validate how well the aforementioned model performs, comparing different hyper-parameter selections.

5. Modify the solver to use the GNN to select a clause during conflict analysis.

6. Integrate the developed system into the LAI's group SMTAPI for constraint problem solving.

7. Test the performance of the two developed encodings. This comparison will show not only how well our custom made PRCPSP encoding performs using a particular solver in comparison to the dummy encoding, but also the size of the respective encodings, meaning how many variables and clauses each one of them will yield.

8. Test how the inclusion of RCPSP-specific information through the GNN may impact our solver by comparing the normal solving process to the deep learning enhanced solving process.

Although the main objective of current SAT solvers is to solve instances in the fastest time possible, we realize that querying a deep learning model is a process that may slow down the solving time due to technological limitations, and for this reason, our objective is to evaluate its performance through different metrics, not only the solving time.

Chapter 2

Feasibility assesment

This section describes the different aspects that make this project possible. Everything needed for this project can be categorized as either equipment (hardware), dependencies and tools (software), and rest. This rest is all contained in the Economical Feasibility section and includes the human costs and software costs. The legality assessment has been contemplated, but it has not been included as the current legislation does not intervene in our research project.

2.1 Equipment

This project has been developed in a computer with the following specifications:

- Processor: 12th Gen Intel Core i5-12400F
- Memory: 16.0 GB DDR4
- Graphic Card: NVIDIA GeForce RTX 3060
- Storage: SSD M.2

This competitive computer enabled me to work with basic graph neural networks without a lot of complications.

During the development of this project, I have also been using a cluster provided by the Logic and Artificial Intelligence research group at my university. This cluster has made it possible to solve the PRCPSP instances in a competitive time and more importantly, solve all instances with nodes that have a consistent performance in the sense that the results are comparable regardless of which node has been used to obtain them. It has 22 CPU cores running in parallel and 16 GB of RAM.

2.2 Dependencies and Tools

For the development of the SAT encodings of the PRCPSP I have used a C++ library developed by the Logic and Artificial Intelligence research group called SMTAPI. This library has enabled me to have a more advanced starting point, not having to start from scratch on the communications between the encoding and the solver, which have been developed following its architecture and as a new module.

In addition, I have used the solver "Maple COMSPS DRUP" from the 2016 SAT competition [2] based on the popular solver MiniSAT. The selection of this solver was done by my supervisor Jordi Coll, because his familiarity with MiniSAT-related solvers was vital for the subsequent modifications done and for other reasons explained in Chapter 7: Studies and Decisions. Still in C++ the library `Python.h` provided helpful communications between the Maple solver and the Python module, which is vital for the newly developed conflict selection.

To develop the deep learning model I have used the Python libraries PyTorch and PyTorch Geometric (PyG), as they provide helpful implementations of several deep learning methods that will be used throughout the project. Other Python libraries that are less important to the development process have also been selected to add visualization (tqdm, matplotlib) or other functionalities (os, time, numpy, pickle). We go further into why we selected these libraries and how they work in Chapter 7: Studies and Decisions.

The version control of both the Python deep learning model and the C++ library is being done using Git, and the planning of the project has been done using the tool Trello [3] (more information on why it is being used in Chapter 3: Methodology).

2.3 Economical Feasibility

This project is being developed over 5 months dedicating an average of 6 hours per day and excluding weekends. This results in 22 weeks, which means that the total time dedicated to this project will be approximately 660 hours. If a Junior AI engineer working makes 11,5€ per hour (2000€ per month according to the web Glassdoor [4]), it will mean the total wage would be around 7590€.

All programs and tools used in the thesis were free of use, so this aspect adds no other costs to the project. The version of the aforementioned tool Trello is free, so it will incur no additional costs.

The cluster has been provided free of charge by the Logical and Artificial Intelligence research group. This cluster cost around 23000€ in 2021. Looking for similar clusters on the internet, Google offers its Google Kubernetes Engine ([5]) with similar specifications (22 vCPUs and 16Gb of RAM) for about 1€/hour. We are going to be making use of the cluster for around 50 hours.

The price (in 2023) of the previously described computer used to develop the project was around 1300€. We are going to assume that the costs of keeping the computer on are negligible.

This results in the costs found in Table 2.1.

Category	Time in hours	Cost per hour	Cost (€)
Human costs	660	11.5	7590
Programs & Tools	-	0	0
Cluster	50	1	50
Computer	-	-	1300
Total	-	-	8940

TABLE 2.1: Table of costs of the thesis.

Chapter 3

Methodology

The methodology that has been used to develop the this thesis is the SCRUM methodology. This methodology is designed and typically used for teams of developers, so I have done the appropriate adaptations to meet the needs of my single-developer project.

3.1 The SCRUM Methodology

The SCRUM methodology is an agile development methodology. Each agile methodology introduce its own particularities, but they all respect the foundational principles on Agile methodology, known as the Manifesto for Agile Software Development [6]. This manifesto includes the following twelve principles.

- Customer satisfaction by early and continuous delivery of valuable software.
- Welcome changing requirements, even in late development.
- Deliver working software frequently (weeks rather than months).
- Close, daily cooperation between business people and developers.
- Projects are built around motivated individuals, who should be trusted.
- Face-to-face conversation is the best form of communication (co-location).
- Working software is the primary measure of progress.
- Sustainable development, able to maintain a constant pace.
- Continuous attention to technical excellence and good design.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- Best architectures, requirements, and designs emerge from self-organizing teams.
- Regularly, the team reflects on how to become more effective, and adjusts accordingly.

These principles can be easily applied to our needs and most of them suit this project perfectly, like sustainable development, self-organizing teams or delivering work frequently. But SCRUM goes further than this manifesto by adding several concepts

that help ground the twelve principles and offer practical solutions. In the SCRUM guide [7], the authors describe the SCRUM methodology as being proficient in five values: Commitment, Focus, Openness, Respect, and Courage. These underlying values emerge from the characteristics of the SCRUM methodology, which are described as follows.

In SCRUM, people in a team are divided into three roles: the developers, who actively participate in developing the product, the product owner who accounts for maximizing the value of the sprint team, and finally the *SCRUM master*, who is accountable for establishing the SCRUM as described in the SCRUM guide.

In SCRUM, the project is divided into sprints. These sprints are fixed-length events comprised of different stages. Firstly, the sprint planning stage is the moment when the people participating in the sprint, specify what will be the focus of the sprint, the work expected to be done and how will this work be divided into smaller sub-parts. Once the planning stage is completed we move to the programming stage. During this stage, daily stand-up meetings are established of about 15 minutes where the developers lay out their course of action for the following day. At the end of each programming stage, there are two very important team meetings. Firstly, the sprint review is the opportunity to present the sprint results and showcase what has been accomplished, often doing a demonstration of the product. Then there is the Sprint Retrospective, which focuses on identifying areas of improvement in the process and dynamics of the sprint and finally developing a plan to implement these improvements.

Other important concepts in the SCRUM methodology are the *Product Backlog* and the *Sprint Backlog*. The Product Backlog is the list of all the features and activities that need to be completed to finish the product. The Sprint Backlog on the other hand is the set of features and activities from the Product Backlog that will be completed during the current sprint. The Product Backlog may not have features as detailed as the ones in the Sprint Backlog, and it has constant modifications to adapt it to the needs of the project, whilst the Sprint Backlog should only receive minimal modifications.

To adapt this methodology to this solo-developing project, some changes have been made. The division of the sprint into four parts has been kept intact, as well as the Product Backlog and Sprint Backlog structure. The role distribution is the following: my supervisors are the product owners, I am the developer, and I am also the SCRUM master. This posed a challenge, as I would have to find time in every Sprint Retrospective to identify what could be improved in the methodology and make sure I am correctly following it, playing both the role of the developer and the SCRUM master.

During this thesis, I have been using the program Trello [3] to act as a Product Backlog and Sprint Backlog. This tool enables me to divide the total sprint work into smaller tasks, as well as to present them in a KANBAN-style grid, which proved to be very effective in organizing what needed to be done during the sprint planning. In addition, working on the thesis from the department's offices three times a week also enables me to have these *Daily Scrums*, or stand-up meetings with both my supervisors. Although these meetings are not held daily, they take place three times a week.

Chapter 4

Planning

This thesis will be developed between January and May 2024. It will be divided into three main sprints, lasting two to four weeks. The three sprints will have a part of the application assigned, but they may also include improvements or corrections of other parts.

The different stages of development of the thesis are listed below, including the development sprints and the planning stages. The colour next to each stage identifies them in Figure 4.1, which shows the weeks assigned to every one of them.

Preparation for the first sprint Before the first sprint, a review of the existing literature on the subject has to be done. The objective is to find papers that describe RCPSP and PRCPSP SAT encodings as well as papers that describe approaches to deep learning models that learn from SAT problems, familiarizing myself with the concepts and preparing myself for the thesis development.

First Sprint: PRCPSP Encoding The first sprint, will be dedicated to developing the PRCPSP encoding and solving it using a SAT solver. This will mean modifying the SMTAPI provided by my tutors to work with PRCPSP encodings, which previously only included the multi-mode resource-constrained project-scheduling problem (MRCPSP), as well as developing the actual encoding and modifying the SMTAPI to be able to add features for the GNN model to the variables of the encoding and create the corresponding feature file.

Preparation for the second sprint Planning the second sprint will mean defining the communication scheme between the library that solves PRCPSP instances using the newly defined encoding and the future GNN model, as well as designing its architecture and planning the training and validation workflows.

Second Sprint: GNN Model During the second sprint, the objectives will be to first build the scripts that transform an RCP instance into the files suitable to learn from using the developed GNN Dataset. This will be done through the SMTAPI, DRAT-trim and Maple Solver. Secondly, we are going to build the GNN model's architecture, training workflow and validation workflow. The selected hyperparameters during this sprint may be tweaked during the third sprint to optimize the model based on empirical results.

Third Sprint: Solver Modifications The third sprint will consist of modifying the existing SAT solver recommended by my tutor, to enhance its conflict selection capabilities with the option of querying a GNN model to ask which

January

M	T	W	T	F	S	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

(A) January

February

M	T	W	T	F	S	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29			

(B) February

March

M	T	W	T	F	S	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

(C) March

April

M	T	W	T	F	S	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

(D) April

May

M	T	W	T	F	S	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

(E) May

FIGURE 4.1: Calendar Grid of the Planning

conflict within a set of candidates should be used to prune the search tree. Once these modifications are built, the final step will be to extract the results by comparing the resolution metrics using the clause outputted by the solver and using the clause that caused the first conflict, as it is currently doing.

Results and memory This final part of the project will be dedicated to gathering the results of the project and presenting them in the memory.

The planning of the sprints can be seen in figure 4.1. The Sprint Reviews and Sprint Retrospectives for each of the sprints are scheduled on the next Monday after finishing it.

Chapter 5

Preliminary concepts and Framework

This chapter will describe the important concepts that should be known before going into this project.

This thesis has been developed by Bernat Comas i Machuca, with the mentorship of Jordi Coll Caballero and Mateu Villaret Auselle.

5.1 Scheduling Optimization

Scheduling problems have been and are still widely studied in the literature due to their ubiquity in industry and services [8]. These problems involve determining the execution periods of a set of activities, subject to various constraints. In this thesis, we focus on the Resource Constrained Project Scheduling Problem (RCPSP), which is the most studied problem involving resource constraints, and which is generalized by many other problems of practical interest.

The Resource-Constrained Project-Scheduling problem (RCPSP) is the problem of finding the minimal makespan that allows for the execution of a set of activities respecting given preemption rules and renewable resource capacities. The makespan of a set of jobs is the time between the start of execution of a group of activities and their finish time [8]. RCPSP problems aim to minimize the makespan of a set of activities (jobs) that can be programmed while preserving precedence constraints and renewable resource constraints. This means that a job can be programmed only when all its predecessors have been executed (precedence constraints), and in every instance, there is a maximum amount of a single resource (capacity) that can be used at once(resource constraints). Every activity has a specific demand for every one of the resources (which can also be 0), and resource constraints force the sum of resource demands of all activities executed in a single time instant to be non-greater than the capacity of that resource.

An RCPSP instance, as defined in Miquel Bofill et al. (2020) [9] can be formally defined by a tuple (V, p, E, R, B, b) where:

- $V = \{0, 1, \ldots, N, N+1\}$ is a set of activities. Activities 0 and $N+1$ are dummy activities introduced by convention, which represent the start and the end of the schedule, respectively. The set of non-dummy activities is defined by $A = \{1, \ldots, N\}$.

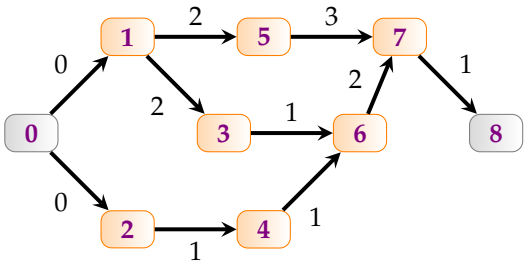
- $p \in \mathbb{N}^{N+2}$ is a vector of naturals, where p_i is the duration of activity i. For the dummy activities we have $p_0 = p_{N+1} = 0$, and $p_i > 0 \; \forall i \in A$.
- E is a set of pairs of activities, representing end-start precedence relations. Specifically, $(i, j) \in E$ iff the execution of activity i must precede that of activity j, i.e., activity i must finish before activity j starts. We will assume that we are given an activity-on-node precedence graph $G = (V, E)$ that contains no cycles, since otherwise the precedence relation would be inconsistent. By convention, there is a path from activity 0 to each other activity, and also a path from each activity to activity $N + 1$.
- $R = \{1, \ldots, v\}$ is a set of renewable resources.
- $B \in \mathbb{N}^v$ is a vector of naturals, where B_k is the available amount of each resource k.
- $b \in \mathbb{N}^{(N+2)\times v}$ is a matrix of naturals corresponding to the resource demands of activities, where $b_{i,k}$ represents the amount of resource k that activity i demands per time step during its execution, with $b_{0,k} = 0$, $b_{N+1,k} = 0$ and $b_{i,k} \geq 0$, $\forall k \in 1..v$, $\forall i \in 1..N$.

A schedule is a vector of naturals $S = (S_0, S_1, \ldots, S_N, S_{N+1})$ where S_i denotes the start time of activity i. We require w.l.o.g. that $S_0 = 0$. A solution of the RCPSP is a schedule S of minimal makespan S_{N+1} subject to:

- Precedence constraints: every activity must start after all its predecessors have finished.
- Renewable resource constraints: the capacity of a resource cannot be surpassed at any time, by the sum of the demands of the activities running at that time.

Figure 5.1 depicts an example of an RCPSP instance with 7 (non-dummy) activities and 2 resources, and Figure 5.2 shows an optimal solution for it.

Precedence graph:



Activity durations and demands (left), resource capacities (right):

	0	1	2	3	4	5	6	7	8
p_i	0	2	1	1	1	3	2	1	0
$b_{i,1}$	0	3	3	2	1	2	0	3	0
$b_{i,2}$	0	2	2	4	3	1	2	2	0

B_1	B_2
3	4

FIGURE 5.1: Example of an RCPSP instance with 7 (non-dummy) activities and 2 resources.

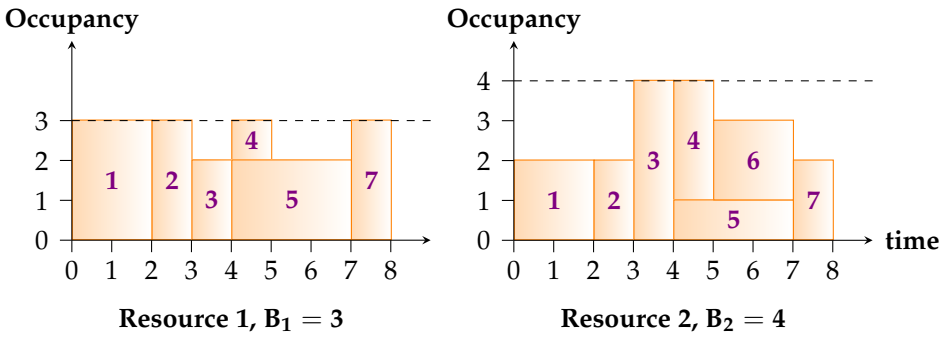


FIGURE 5.2: An optimal solution for the instance of Figure 5.1, with schedule $S = (0, 0, 2, 3, 4, 4, 5, 7, 8)$. The capacity of the resources is never exceeded, and the precedence relations are respected.

Other important concepts in RCPSP are Earliest start times (ES, the earliest an activity can start), Earliest closing times (EC, the earliest an activity can finish), Latest start times (LS, the latest an activity can start) and Latest closing times (LC, the latest an activity can finish). In addition, we also introduced extended precedences, as used in Bofill et al. (2020) [10] by computing the transitive closure of the given activity precedence graph G. We will denote by $G^* = (V, E^*)$ the extended precedence graph, where E^* is a set of weighted edges. There will be an edge $(i, j, l_{i,j}) \in E^*$ for every pair of activities (i, j) in V which are connected by a path starting at i and ending at j. The time lag $l_{i,j}$ will be a lower bound on the difference between the start times of i and j. This time lag $l_{i,j}$ will correspond to the weight of the critical path from i to j in G, i.e., the minimum weight (sum of durations) of any path going from i to j in G. Time lags can be easily computed with the Floyd-Warshall algorithm. All these metrics can be precomputed from the input data.

There are different approaches to finding the minimal makespan. For example, the up-bottom approach starts with an upper bound, meaning a makespan that is big enough that we are sure that all activities will be able to be scheduled, and decrement it until the instance encoded is unsatisfiable (meaning the makespan is too small to schedule all the activities). A commonly, but quite bad in terms of performance, used upper bound is the sum of the total duration of the activities.

Resource-constrained scheduling has many different variants, some of the most known are Multi-mode Resource-Constrained Project-Scheduling Problems (MRCPSP) [11] and Preemptive Resource-Constrained Project-Scheduling Problems (PRCPSP) [8]. In this project, we are going to work on Preemptive RCPSP.

PRCPSP is a variant of RCPSP that allows the preemption of activities. Activity preemption enables the solution to introduce pauses in the execution of activities and resume its execution at a later point in time. PRCPSP instances have the same format as RCPSP instances: they can be defined by means of the same tuple (V, p, E, R, B, b) described in Definition 8.1.1.

Preemptive Scheduling results in makespans at least as optimal as its non-preemptive counterparts (naturally, an RCPSP solution is also a PRCPSP solution, but not necessarily the other way around). Therefore, activity preemption often shortens the makespan compared to classical RCPSP, so it is an excellent alternative if the activities can be split and the cost of starting and stopping an activity is negligible. The

preemption discussed in this thesis results in activities being split only at ordinal times, as we can easily see that any decimal preemption could be represented as a non-decimal one by multiplying everything by the inverse of the fraction of time we are trying to preempt in.

RCPSP instances can be represented in different formats, the most widely used ones being SM files and RCP files. These are plaintext files containing the activities to be scheduled, their durations, resource usages and precedence rules.

5.2 Constraint solving and the Boolean Satisfiability problem (SAT)

The Boolean Satisfiability problem (SAT) is the paradigmatic NP-complete problem. As such it has drawn the attention of the research community in the last decade, specially for its aplicability in solving hard combinatorial problems. SAT is the problem of deciding the existence of an interpretation satisfying a propositional boolean formula. Typically SAT considers boolean formulas in Conjunctive Normal Form (CNF). A CNF is a conjunction of clauses, and each clause is a disjunction of literals. A literal is either a variable (positive literal) or its negation (negative literal). An interpretation in the boolean satisfiability context is an assignment of true (1) or false (0) to the variables of a formula. We say that an interpretation satisfies a formula if it satisfies all the clauses of the formula [12]. A clause is satisfied by an interpretation if at least one of its literals is satisfied. A positive literal x is satisfied by an interpretation if this interpretation assigns true to x. A negative literal $\neg x$ is satisfied by an interpretation if this interpretation assigns false to x.

The following is an example of a boolean formula in CNF.

$$(p \vee q) \wedge (q \vee \neg p) \tag{5.1}$$

The variables in formula (5.1) are p and q, and the literals appearing in this formula are p, q and $\neg p$. This boolean formula has two clauses separated by the conjunction operator. An interpretation that would satisfy this boolean formula would be p=0 and q=1, as every clause would have at least one literal which is evaluated to true. This is exemplified in formula (5.2) where the satisfied literals are green and the falsified ones are red.

$$(p \vee q) \wedge (q \vee \neg p) \tag{5.2}$$

There are two specially relevant inference rules to understand how a SAT solver works: the unit propagation rule and the resolution rule. The unit propagation rule states that, given an assignment, if all the literals of a clause are falsified except one, which is unassigned, this unassigned literal must be assigned true to satisfy the clause. The resolution rule derives a clause $A \vee B$ from the premises $x \vee A$, $\neg x \vee B$. Any clause derived with resolution (conclusion) can be added to the set of clauses of the original formula since it is logically implied by the premises.

The state-of-the-art algorithm to solve the SAT problem is Conflict-Driven Clause

Learning (CDCL) [13–15]. This algorithm combines the exploration of a search tree with the application of inference rules (mostly the resolution rule) to learn new clauses. More precisely, CDCL will do branching on the search tree by assigning unassigned literals (making *decisions*). The concept of decision level refers to the depth in the decision tree, i.e. the number of decisions made since the beginning of the search. After each decision, the partial assignment is extended with the necessary assignments inferred with unit propagation. This process is repeated until either all clauses are satisfied or a clause becomes unsatisfied (we detect a *conflict*). If a conflict is detected the resolution rule is used to learn a new clause explaining such conflict that defines a point to backtrack to in the search tree.

Typically the existent SAT solvers stop the search when they encounter the first conflict. However, if we ignore the found conflict and continue applying unit propagation we can find other conflicting clauses that may lead to better backtracking points thus speeding up the solver. This degree of freedom is what we will explore during this thesis.

Apart from decisions, modern SAT solvers can also do restarts, which means removing all the decisions (i.e. all branching points of the search tree) while preserving the learnt clauses.

The RCPSP is NP-hard, which means that it is at least as hard as any problem in NP, including SAT. Finding an optimal solution for this problem requires exploring a large solution space, and there's no known polynomial-time algorithm to solve it optimally. One of the most common approaches to solve the PRCPSP is to encode them into SAT [11]. This encoding, allows us to use efficient SAT solvers to find solutions to the original problem [16].

This approach is advantageous because SAT solvers have been extensively optimized and are efficient in many cases. However, while encoding PRCPSP into SAT allows us to leverage SAT solvers, it doesn't guarantee efficiency for all instances of the original problem. SAT solvers treat the encoded problem purely as a set of logical constraints and don't understand the particularities of the original problem. In this thesis, we are going to take into account the original problem to inform the SAT solver and try to improve its efficiency.

The standard file format to represent CNF formulas is DIMACS. An example of a DIMACS file used to store the formula (5.1) would take the following form:

```
p cnf 2 2
1 2 0
2 -1 0
```

The first line is the header. It starts with `p`, followed by the format, `cnf`, and the number of variables and clauses, in this case, both are 2. Then all the clauses are written separated by a zero and often also a line break.

There are other approaches to solving this type of problem such as Satisfiability Modulo Theories (SMT) solvers [17]. SMT solvers extend the capabilities of a SAT solver by incorporating theories beyond boolean logic, such as arithmetic theories, arrays or bit-vectors. In this thesis, we are not going to use SMT solvers, but it is important to be familiar with the concept as it might appear during the development.

5.3 Refutation trees and proving the unsatisfiability of a SAT formula

Refutation trees are directed graphs that indicate which of the original clauses from the formula participated in proving its unsatisfiability, as well as the clauses that have been learnt in the process [16]. It is important to note that there may be more than one possible refutation tree for a formula. An example of a refutation tree can be seen in Figure 5.3. Here, we can see that clauses 1, 2 and 3 participate in the refutation tree while clause 4 does not. This refutation tree proves that the boolean formula comprised of the clauses 1, 2, 3 and 4 as written on the left side that use variables a, b and c is unsatisfiable, meaning no interpretation satisfies the four clauses at the same time. Here, a new clause that did not belong to the original set of clauses appeared, a. This new clause was derived from clauses 1 and 2 using the resolution rule [18].

$$(a \vee b) \wedge (\neg b \vee a) \wedge (\neg a) \wedge (b \vee c)$$

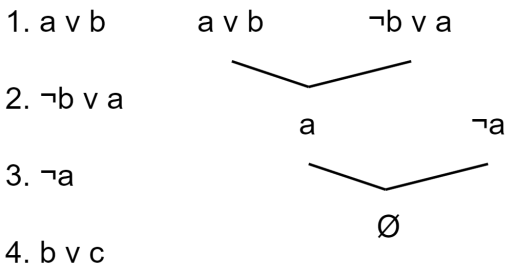


FIGURE 5.3: Example of refutation tree.

The CDCL algorithm generates a refutation tree in the process of determining the unsatisfiability of a boolean formula. In particular, after every conflict, we use the resolution rule to learn a clause which in turn can be used to learn new clauses in future conflicts. Unsatisfiability is determined when CDCL finds the empty clause.

Some SAT solvers can output a proof of unsatisfiability in a file detailing the process followed to prove the unsatisfiability of a given formula. This proof of unsatisfiability is not the refutation tree, as it contains all learnt clauses, including those not participating in the refutation. Different checkers validate the proof generated by SAT solvers. The most recent and widely used one is DRAT-trim [19, 20]. This checker not only validates that a formula in DIMACS is unsatisfiable but it can also output a trace check. This trace check contains a (sub-optimally) minimized subset of clauses which is sufficient to generate a refutation of a SAT formula. For this reason, we believe that the clauses that appear in this generated trace are very relevant for us.

5.4 Deep Learning concepts

Deep learning is a subset of machine learning that includes algorithms used to model high-level abstractions into data [21]. These models try to simulate the learning process of the human brain and are often known as neural networks (NN). Neural networks are comprised of multiple layers (hence the “deep” in deep learning), which extract features from the raw input.

Although many different deep learning architectures exist, in this project we are going to try to model our data into a Graph Neural Network (GNN). Graph Neural Networks are a particular deep learning architecture suitable for modeling data that can be represented as graphs [22]. These types of neural networks are used on a vastly different fields such as protein folding [23], social networks or computer vision, where each pixel in an image can be seen as a node connected to its surrounding pixels. Boolean formulas in Conjunctive Normal Form (CNF) can be naturally represented in a tripartite undirected graph. The first partition includes a node for each positive literal, the second partition a node for each negative literal, and the third partition includes a node for each clause. There is an edge between the two literals of the same variable, and for every clause there is an edge that connects it to each of its literals. For this reason, this type of neural network is the obvious choice. Figure 5.4 is an example of the tripartite graph.

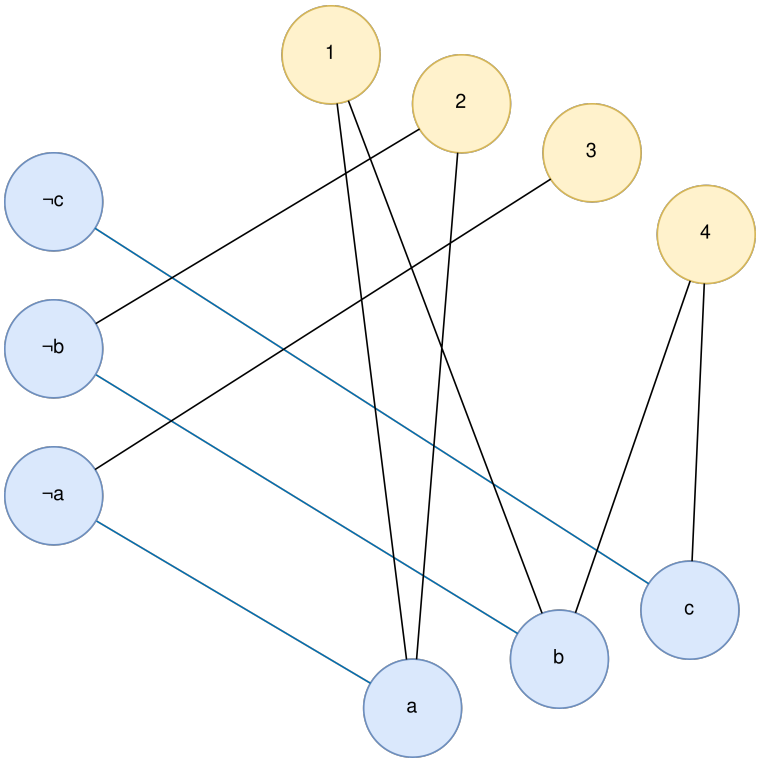


FIGURE 5.4: Graph representation of the boolean formula $(a \vee b) \wedge (\neg b \vee a) \wedge (\neg a) \wedge (b \vee c)$. In blue, the literal nodes and literal-to-negated edges, in yellow the clause nodes and in black the edges that connect each clause with its literals.

There are many different architectures that have been developed and are encompassed under the term GNN, like Graph Convolutional Networks (GCN) or Graph Attention Networks (GAT). GCNs generalize the concept of convolution from images to graphs [24]. A convolutional layer uses kernels that slide over the input data to perform a convolution operation. They are very useful because convolutional operations maintain the spatial structure which is vital in Graph structured input data.

GAT networks on the other hand implement attention mechanisms for feature learning on graphs [25]. Message passing is a very important concept in Graph Neural Networks because it enables information exchange and aggregation among nodes

in a graph [26]. This process is done through the aggregation of neighborhood information to update their own information. At each iteration of a message passing layer, a hidden embedding is updated based on information obtained from its neighborhood (Figure 5.5). The particular implementations of these structures are not in the scope of this project.

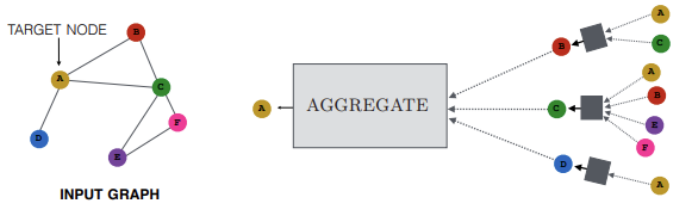


FIGURE 5.5: Illustration of the message passing process. Source: Omar Hussein, Medium [26]

When training a deep learning model, it is important to choose the optimal parameters that will enable the model to correctly map the input features to the labels or targets (independent variables), thus learning from the data [27]. The parameters that control the learning process in deep learning are known as hyperparameters (hyper meaning top-level parameters), and the model parameters are directly affected by them. Hyperparameters, unlike model parameters, are not modified during the training process and therefore are not present in the final model. Some model parameters include the train-test split, the optimization algorithm, the learning rate in the optimization algorithm, the activation functions, the number of hidden layers, the number of epochs, batch size or the choice of loss function among others.

The Binary Cross Entropy Loss (BCELoss [28] is a loss function that is going to be used in this project. A loss function is used to measure the difference between predicted probabilities and actual labels. The function of the BCELoss is the following:

$$L(y, p) = -(y \cdot \log(p) + (1 - y) \cdot \log(1 - p))$$

Where `y` represents the label and `p` represents the predicted probability. As a loss function, the BCELoss should yield 0 if the prediction is correct and 1 if the prediction is incorrect. This is why it uses the negative logarithm. As it can be seen in Figure 5.6, the entropy of a probability (obviously between 0 and 1) grows as the probability gets closer to 0.

For this reason, if the prediction and the label are both 0 or if they are both 1, the function will yield $-(1 \cdot \log(1))$ which is 0. On the other side, if they are very distinct, for example if the prediction is 1 and the label is 0, the loss function will be $-(1 * log(0))$, which is a very large number.

Another loss function used in this project is the MSELoss or Mean Squared Error loss, which has the following definition:

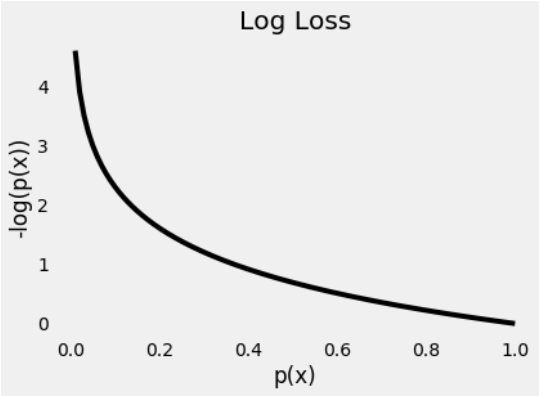


FIGURE 5.6: Evolution of the entropy with different probabilities. Image extracted from "Understanding binary cross-entropy / log loss: a visual explanation" by Daniel Godoy [28]

$$\text{MSE}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where y_i is the actual value (label), $\hat{y}_i$ is the predicted value and n is the number of samples. This loss function does exactly what its name indicates, computes the mean squared difference between the labels and predictions of all elements in a batch (group of training examples processed together). Figure 5.7 illustrates the evolution of the loss with respect to the difference between the true values (labels) and predicted values.

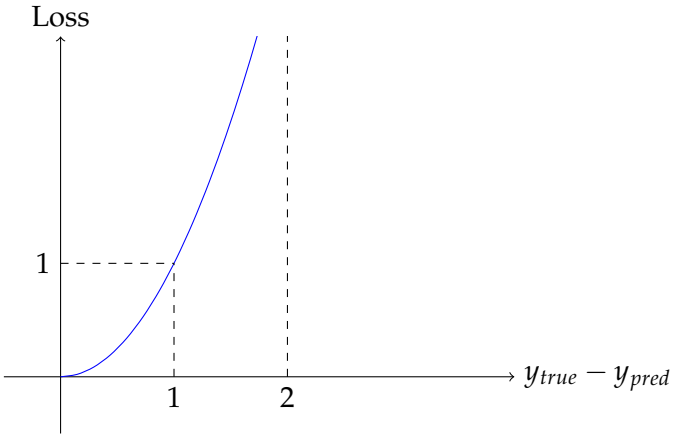


FIGURE 5.7: Evolution of the MSELoss in different inputs

Chapter 6

System Requirements

This chapter describes the requirements of the project. The coloured square appearing left of each requirement will indicate its level of importance, and are classified as follows:

■ - High Importance

■ - Medium Importance

6.1 Functional requirements

Functional requirements define the behaviours, features and functions that should be implemented to enable this project to fulfill its project objectives.

1. ■ The SMTAPI should take an RCP instance as an input and be able to output the DIMACS file which contains the Boolean Satisfiability encoding of the scheduling problem using our newly developed encoding and using the naive approximation that splits the activities.

2. ■ The SMTAPI should take an RCP instance as an input and be able to output the minimum makespan found using our selected solver and the Boolean Satisfiability encoding of the scheduling problem using our newly developed encoding and using the naive approximation that splits the activities.

3. ■ The SMTAPI should be able to label the variables of the encoded instance and output a feature file containing the features linked to each variable.

4. ■ The SMTAPI should be able to solve an instance assisted by repeated calls to our GNN model to select conflicting clauses.

5. ■ The SMTAPI should be able to validate if a solution for an instance of RCP with preempted activities is correct.

6. ■ The GNN should be able to take a DIMACS problem and its feature file and output the quality weights of each clause.

7. ■ The trained GNN models should be stored in a folder to be able to use them later.

8. A script should be created that develops a CSV file containing all the solutions of a specific dataset of RCP problems.

9. A script or group of scripts should be created that when executed submits all the RCP instances in a specified folder to find their makespan using the SMTAPI in the cluster.

6.2 Non-Functional requirements

Non-functional requirements are the requirements that specify how a system should behave rather than what it should do. The non-functional requirements of this project do not include any performance-related requirements as the objective of this thesis is to evaluate the performance of this implementation, nor security requirements, as it does not work with any private data or sensitive stored data.

1. The solving process should offer the possibility to use the GNN model or continue working as before.

2. The modifications of the SMTAPI and the solver should enable to easily add other encodings that work with the GNN model.

3. The data used to train the GNN should be publicly available data.

Chapter 7

Studies and Decisions

This chapter describes the studies and decisions made before and during the development of the project. It includes the libraries and external software used as well as why certain choices have been made during the development.

The environment where this work was developed is in the Windows Subsystem for Linux from Windows 11 in an Ubuntu distribution.

7.1 The SMTAPI

The SMTAPI is a library initially developed by my supervisor Jordi Coll in the LAI research group at the University of Girona. It offers a framework to generate SAT and SMT encodings and solve them using a backend solver. Moreover, the LAI group has already implemented a set of solvers and encodings for different problems using this library, including RCPSP and many of its variants. This framework divides the different parts of solving an RCPSP instance into a set of classes and functions that are explained in this section. It uses the g++ version 11.4.0.

7.1.1 Parser

The parser is the function that translates the `.RCP` file into an internal representation of an instance, the RCPSP or PRCPSP class. It receives a file as an input (usually the name of the file), and returns an instance of the class the file was parsed to. The parser depends both on the instance class we want to output and the format of the file we take in as an input (RCP, SM...).

For the purposes of this thesis, two new parsers will be developed. The first one takes in an RCP file and outputs an instance of class PRCPSP. The second one transforms the RCP file into an RCPSP problem but considering activities split at each time frame. The details of the two encodings will be explained in the following chapters.

The RCP format [29] was designed prioritizing the simplicity of the parsing process rather than human readability, and it lacks of complex syntactic structure. This means that implementing a parser does not require the use of common grammar recognition algorithms, but just processing a simple stream of characters.

7.1.2 Instance class

The instance class is the internal representation of an RCPSP instance. Its attributes include all the information provided by the RCP file (activities, resources, demands, capacities, predecessor relations...).

It may also contain functions and attributes that compute or store data that was not directly present in the RCP instance but has been calculated from this data and will help during the solving process.

7.1.3 Encoding Method

The encoding method contains functions that generate the formula (SMTFormula) of the RCP instance in SAT as well as the object representing the instance to be encoded.

Its three most important functionalities are the following:

- Encoding: The instance is translated into an equivalent object of type SMT-Formula, containing the boolean formula obtained from the encoding of the PRCPSP problem using the implemented method.
- Recover model: This function does the opposite process of the encoding function, receiving an encoded formula and the values the solver has assigned to each variable, and retrieving the results, that is, the values of each variable that participated in the encoding. This function translates the output of the SAT solver, that is 0 or 1 for each boolean variable, to their original meaning inside the problem, so it outputs the human-readable solution (the schedule of the activities). Evidently, this function is only called when the SAT solver has proven satisfiability.
- Narrow the bounds: This is an optional function. If we are optimizing a problem, it enables the optimizer to avoid having to encode the problem every time the upper bound is lowered by adding the needed clauses. These clauses depend on the encoding we are implementing, but their objective is to force a new makespan, narrowing the bounds (keeping the same lower bound and lowering the upper bound). This in return results in a better performance because instead of encoding the problem from scratch in every iteration, we simply add new constraints to the already existent encoding.

For the purposes of this thesis, a new encoding method will be developed for PRCPSP instances.

7.1.4 Encoder

The encoder is the part of the application that interacts directly with the solver. It includes the functions that call the solver and check if a certain makespan is correct. It contains the last encoded formula as well as solving related data, and it may also contain a solver instance if the encoding works directly with the solver. If it does not contain the solver instance, we usually write the boolean formula into a DIMACS file and query the solver through the command line.

It offers the following functionalities:

- Check if a makespan is correct: If the encoding has not been created yet it creates it, otherwise it narrows the bounds. Then it calls the solver to check if the current instance is satisfiable, that is, it exists an interpretation that evaluates the encoded formula to true. Finally, it asserts the solution.

- Asserting the solution: Asserting the solution means calling the solver with the previously encoded formula and retrieving the values of each variable.

There are many backend solvers to choose from, each one with its encoder implementation. In this thesis, we have adapted the MiniSAT encoder, as it interacts directly with MiniSAT related solvers.

7.1.5 Solver

The SAT solver is a vital part of this workflow because it is what outputs an interpretation for the formula or not depending on the satisfiability of the boolean formula it takes as an input. It interacts with the library through the encoder. The solver will be modified to work with the deep learning model developed, as well as to enable the output of the variable features.

7.1.6 Optimizer

An optimizer is an implementation of the logic needed to optimize an RCP instance (finding the lowest makespan). There are several implemented optimizers because there are different techniques to find the minimum makespan. Some of the implemented optimizers are the following:

- Bottom-Up optimizer: Starting with the lowest makespan, it is incremented until it finds a satisfiable instance, thus finding the lowest satisfiable makespan.

- Up-Bottom optimizer: Starting with the highest makespan, it is decremented until it finds an unsatisfiable instance, thus finding the highest unsatisfiable makespan.

In this project, the up-bottom optimizer was selected because in several works by the LAI group it had been seen that its results were superior to the rest [9, 30]. The workflow of the up-bottom optimizer is illustrated in the activity diagram in Figure 7.1 as well as the sequence diagram in Figure 7.2.

7.1.7 SMTFormula

The SMTFormula is the internal representation of a Boolean Satisfiability Formula. It stores the clauses and literals, and provides functions that directly encode some common scheduling constraints, such as Pseudo-Boolean constraints, Sorting constraints or Cardinality constraints.

7.1.8 Main

The entry point to the library are the main classes, custom made for each individual problem. They parse the program arguments, call the instance parsers, initialize the optimizers, select the solvers, and initiate the optimizing process. For the purpose of this thesis a new main will be created and an existing main implementation of the MRCPSP problem, which can also solve the RCPSP problem, will be used.

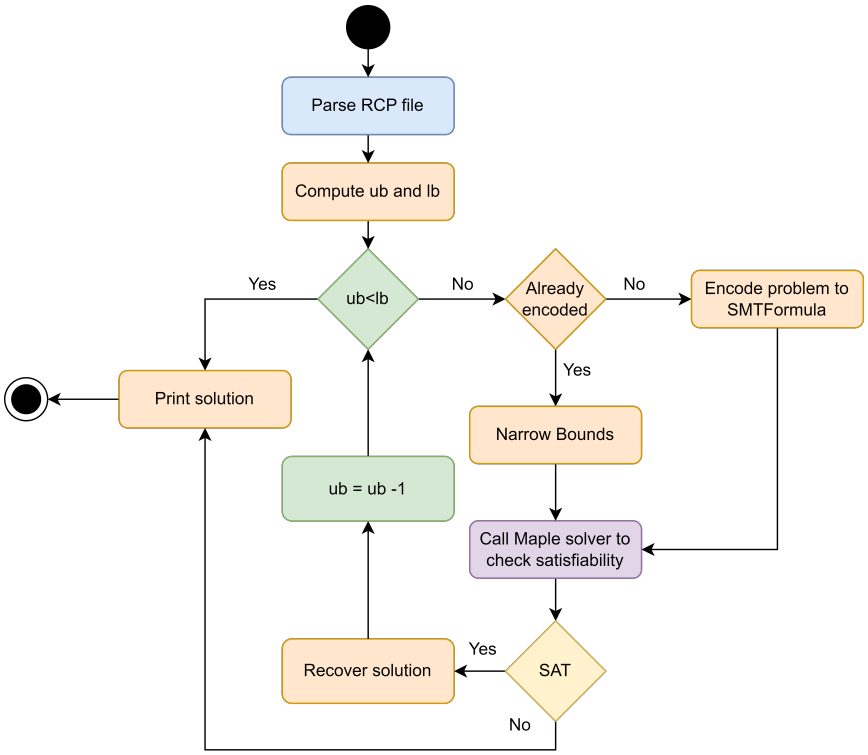


FIGURE 7.1: Activity diagram that represents the workflow inside the SMTAPI. The colour code of the image is the following: Encoding, Encoder, Solver, Optimizer, Main

Figure 7.2 represents a simplified version of the internal workflow of SMTAPI when a user queries it to solve an RCP instance.

7.2 Maple Solver

The Maple COMSPS DRUP is a solver extracted from the 2016 SAT competition, based on the popular SAT solver MiniSAT. The advantages this solver offers in comparison to other solvers we considered are the following:

- Ease of implementation: Compared to other solvers like KisSAT or Cadical, this is a software that is easy to modify and is ideal for our needs.
- Generation of a SAT proof: It was vital for our solver to emit proofs of unsatisfiability, because these proofs would then be used by the DRAT-trim to generate the trace check files we use to label the clauses and train the GNN.
- Generation of a DIMACS file: This solver can generate a DIMACS file containing the clauses inside its solving execution.
- SMTAPI: As SMTAPI already included an encoder to work directly with the MiniSAT solver, this specific version of the MiniSAT solver could be directly asked to solve a boolean formula without needing to wait for a DIMACS file to be written.
- Modern solver: Although it is a solver that came out some years ago, it still

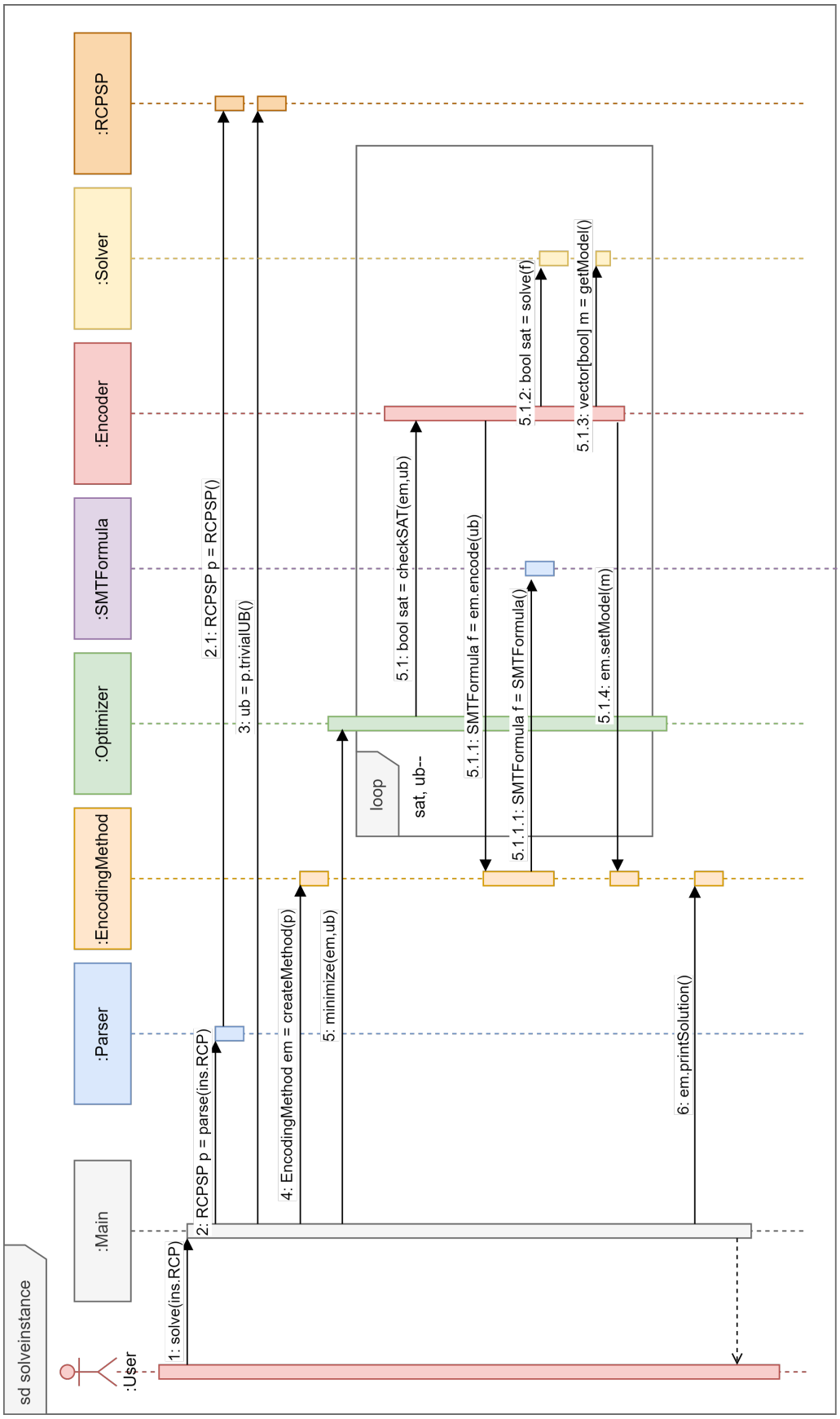


FIGURE 7.2: Simplified sequence diagram illustrating the workflow of the SMTAPI when solving an RCP instance. Note that the encode function in 5.1.1 might be substituted, if it exists, by the narrow bounds function that would modify the existing SMTFormula instead of creating a new one.

offers competitive results, and as our primary objective was not solving speed, this solver offered the perfect balance between ease of implementation and good performance.

- Known and recommended by my supervisor Jordi Coll because he had already worked with MiniSAT-related solvers.

In the remaining of this document we will refer to this SAT solver simply as "Maple", which is the common name for the set of solvers where Maple COMSPS DRUP belongs. We warn the reader not to confuse it with the popular mathematical software by the same name.

7.3 The Python C++ library

One of the challenges of this thesis is to achieve a fast communication between the solver and the GNN model. The approach chosen to fill this role is the Python C++ library [31]. This library not only offers the possibility to call Python scripts from C++, but it enables a direct communication and transmission of data between the SMTAPI and our Python scripts.

This is a key point in our architecture because having a slow communication between the SMTAPI and the GNN model will have a big impact in the solving time of our solver when querying the GNN model. Another option considered was to train the model using Python and then load it from C++ using the PyTorch C++ API [32] directly from our C++ code. This would have completely removed the need for communication between C++ and Python, but although this option would probably have sped up the querying process, we finally opted out of it for several reasons. Firstly, the library PyTorch Geometric, which is indispensable to build a graph model, has been seen to cause problems when being used from C++, as unlike the PyTorch library, PyG does not have a direct port to C++. In addition, the developers have a warning on the first page of the manual [32] that states that the API is in "beta" stability and changes might be made that affect the backend, and that is without even considering GNN models. For this reasons we have chosen to use the Python C++ library.

7.4 SLURM

SLURM (Simple Linux Utility for Resource Management) is an open-source workload manager for Linux clusters [33]. It is highly used in high-performance computing as it provides a framework for job scheduling (not to be confused with scheduling problems). In the context of this thesis, the SLURM workload manager is used in the cluster for testing the encodings and obtaining the results. This workload manager was already installed in the cluster provided by my supervisors.

The most common SLURM instructions are `squeue`, `scancel` and `sbatch`. The first instruction `squeue` prints all the current jobs in execution and waiting in the execution queue; `scancel` cancels a job or a group of jobs and is frequently used with the flag `-u user` to cancel all jobs submitted by a user; and `sbatch` submits jobs to the queue, that will eventually be scheduled. With these three instructions we can do the basic operations we need for this project, so we will not go any further into this sophisticated workload manager.

7.5 DRAT-trim

Recall that the purpose of using the GNN model is to select the best clause among all possible conflicting clauses in a dead end of the CDCL search process. Selecting a promising clause will mean defining a measure that tells us how good a clause is. We refer to this measure as the quality of a clause. The way we define this quality will directly affect the results as it will define how good a clause is and the values the neural network should try to predict. For this project, the measure chosen to evaluate the quality of a clause is its frequency of appearance in the trace check outputted by the DRAT-trim.

DRAT-trim is an external library that enables us to generate trace check files that contain a (sub-optimally) minimized subset of clauses which is sufficient to generate a refutation of a SAT formula. From the trace check file we can easily obtain the number of times the clause has participated in the refutation. Notice that we are supposing that a refutation tree will exist and therefore that the instance is unsatisfiable.

The CDCL algorithm, used to solve SAT, is in practice alternating the search of a solution with the construction of the refutation. Therefore, we work with the hypothesis that prioritizing in CDCL's refutations the use of clauses that would be selected by DRAT-trim will also reduce the size of the refutations generated by CDCL and hence, speed up the solving process. The trace check files are generated using the DIMACS file of an instance and the proof emitted by the solver.

We chose to use this specific satisfiability proof checking tool because it is the most widely used one, it is fast compared to its predecessors, and can emit clause dependency graphs that indicate the clauses that participate in the refutation tree of an unsatisfiable formula. To use DRAT-trim we simply compile it using `g++` (we are using version 11.4.0), and query it:

```
./drat-trim file.dimacs proof.proof -r trace.tc
```

Where `file.dimacs` and `proof.proof` are the encoding and proof files outputted by the solver. The option `-r` followed by a file name asks the tool to not only validate the correctness of the proof but also to output a trace check file. Here it will be named `trace.tc`.

7.6 GNN Architecture

The neural network architecture will take as an input a graph representation of a DIMACS instance as well as the features of each variable and will output the predicted quality of every one of the clauses. To fulfill this purpose we have two possible very distinct ways of building the model.

We can build a classification model that classifies the clauses as either participating in the refutation tree or not doing so. The downside of this option is that the labels would give no information about which of the clauses is better among the conflicts that are classified as participating in the refutation tree.

The second option was to build a regression model (a recommender) that sorts a given set of clauses (the ones that generate a conflict) from the one that is expected

to appear the most in the refutation tree to the one that is expected to appear the least. This option would have normalized labels so that the clause that appears the most in the refutation tree is given a value of 1 and the clauses that do not appear are given a value of 0.

We ultimately decided to build both models and assess their performance. As we are training two very different models: a classifier and a regressor, we will have to adjust the hyperparameters accordingly. We have chosen to use the BCELoss with the classifier, as it is a loss function typically used in problems where labels are either 0 or 1 (hence the name: Binary Cross Entropy), and the MSELoss with the regressor, as it is typically used when we have to predict a continuous value.

7.7 Python libraries

This section describes the Python libraries used to implement the Graph Neural Network (GNN) in Python. The version of Python used is Python 3.10.12, which was released in June 2023.

- PyTorch version 2.2.0 (January 2024) [34]. There are different libraries that offer deep learning implementations in Python, the most famous being PyTorch and Tensorflow Keras. We chose to use the PyTorch library because I had previous experience developing deep learning models that used it.

  This version was chosen because it is the most recent PyTorch version supported by PyTorch Geometric, as the newest version (PyTorch 2.3.0 release on April 2024) caused problems when using undirected heterogeneous graphs.

- PyTorch Geometric (Torch geometric) version 2.5.2 [35]. It is the most popular and complete library used to implement Graph Models using PyTorch, so PyTorch Geometric was the obvious choice. We are using the most recent version.

- Numpy version 1.26.4 [36]. The most popular library to work with large quantities of data and scientific computing and an essential companion for PyTorch. The version selected is the most recent version of the library.

- Pickle version 4.0 [37]. This serialization library will be very useful when parsing the instances several times to consult the deep learning model. It will enable us to serialize and save to a file the already parsed instance and reduce the time it takes to load the instance every time the model is queried thus improving the performance. An alternative to this workflow would be to create a Python server that keeps the instance loaded in memory while the solver queries different clauses, but the idea was discarded due to the already substantial dimensions of the project and the lack of any guarantee that it would improve its performance in any way.

- Matplotlib version 3.8.3 [38]. Displaying graphs during the training of a neural network is very useful and matplotlib is a very popular Python library that offers easy displaying array contents as graphs.

Other libraries used that either are not vital to the functionality of the application or are already included with any modern Python installation include the `os` library

(work with files and directories), `tqdm` library (visually pleasing loading bars) and `time` library (obtain current time).

Chapter 8

Analysis and Design of the system

This chapter contains an analysis of the needs of the project, followed by a design of each one of the modules involved in the thesis.

8.1 Analysis

This thesis has three very clearly differentiated parts that should be analysed separately as they are three distinct modules that communicate between them.

8.1.1 Part I: The PRCPSP encoding

The first part of the project will consist of developing both the PRCPSP custom-made encoding and the PRCPSP dummy encoding. The difference between these two encodings is that in the first one, the RCP instance is encoded into PRCPSP constraints, while in the second encoding, the RCP instance is reduced to an instance of RCPSP as follows.

Reduction 8.1.1. Given a PRCPSP instance (V, p, E, R, B, b), we generate a corresponding RCPSP instance as follows:

For every non-dummy activity $A_i \in V$ with duration d of the PRCPSP instance:

- We introduce a sequence of activities $A_{i,1}, \ldots, A_{i,d}$ of duration 1
- We add precedences between each two consecutive activities $A_{i,j}$ and $A_{i,j+1}$
- For every precedence $(A_i, A_j) \in E$, we add a new precedence $(A_{i,d}, A_{j,1})$
- Every new activity $A_{i,j}$ will have the same resource demands as the original A_i

The resources and capacities, as well as the dummy activities, remain the same in the generated RCPSP instance.

Having parsed the instances using this process will enable us to use RCPSP constraints to solve the problem and obtain its PRCPSP solutions. An example of this activity splitting process is illustrated in Figure 8.1, where an activity of duration $d=3$ is split into three activities.

This part will be designed following the same framework used by the SMTAPI. That means we will need to build two new parsers: one that directly encodes an RCP

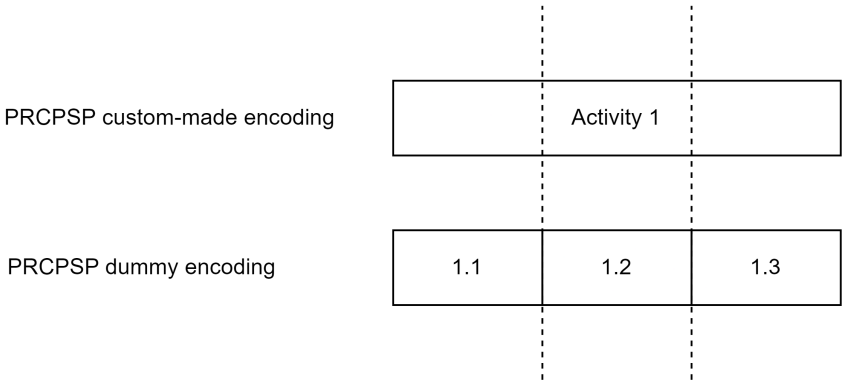


FIGURE 8.1: Activity preemption in the PRCPSP custom-made encoding vs the dummy encoding. In the PRCPSP custom-made encoding an activity can have more than one instant of duration (here it has 3 duration). In the PRCPSP dummy encoding, the parser will break it into three new activities as can be seen with Activity 1.1, Activity 1.2 and Activity 1.3.

file to an instance of PRCPSP and another one that does the activity splitting and parses the resulting sub-activities into an instance of MRCPSP with a single mode to be encoded as an RCPSP. We will also need to build a brand new Encoding of the PRCPSP as the SMTAPI did not have preemptive support yet. Then, we will have to modify the existing SMTAPI MiniSAT encoder to add the support for the Maple solver, so that it can output the features of the variables, create a DIMACS file using the same variable names and be queried from the encoder. The new solver will be added and a new main for the PRCPSP problem will be created as well as modifying the existing MRCPSP main to add a new solving argument that solves the specified RCP problem as a PRCPSP using the dummy encoding.

8.1.2 Part II: The GNN

The second part consists of generating our training instances from the RCP files, creating the GNN and validating it. This will mean finding a way to unify the DIMACS and trace check file in a new file format that contains all the clauses the two files contain as well as the number of times each clause appears in the trace check as clauses that caused the creation of a new clause.

Generating our own training instances is the process that can be seen in Figure 8.2. The white squares represent files that are involved in the process. The yellow rounded squares are programs implemented in C++. The blue ones are programs implemented in Python. Finally the green one is the tool DRAT-trim. The red arrows in the figure represent that the program where the arrow originates generates the file it points to. The black arrows simply indicate that the file is being used by the program the arrow points to. The blue links mean direct communication through C++ function calls.

To build the model we will need to find a way to define the DIMACS graph using structures inside the PyTorch Geometric library. Then, when we have the training instances we will need to build the training process and the architecture of the model. This means that we are going to have to make a choice on the different hyperparameters we want to use. After selecting the hyperparameters and having tested how

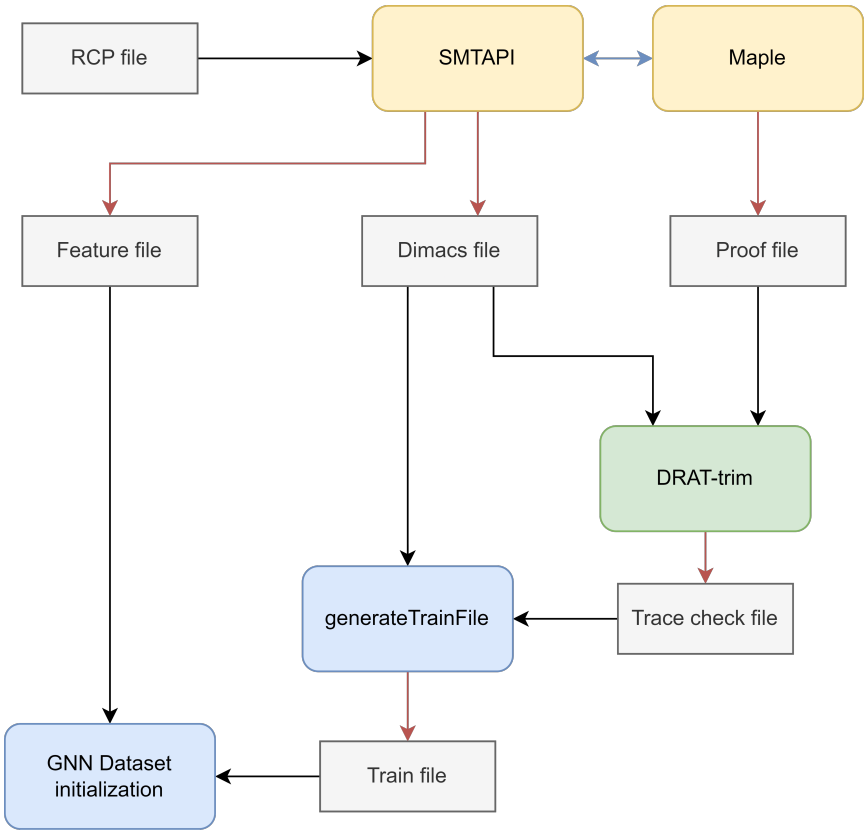


FIGURE 8.2: Pipeline of the instance generation process. The colour code of the elements is: DRAT-trim, C++ Programs, Python programs and files. The arrow code is the following: program creates file, program uses file,

different values behave, we are going to have to make a choice in the metrics we want to use to validate the model and test that it is correctly optimized. This process will have to be made for both approaches to the problem as explained in the previous chapter.

Once we are satisfied with the model's performance, we will save it using PyTorch's serialization module so that it can be loaded and used later on.

8.1.3 Part III: The MAPLE modifications

The final design will be built once the GNN is trained, and will consist of several modifications that have to be done to the solving workflow of the Maple solver. Apart from that, we will also have to develop a Python script that can be called from C++ using the Python C++ library and that queries the module with the existing conflicts and outputs their quality.

When modifying the way the Maple solver operates, our objective will no longer be to stop the propagation every time a conflict is found, but to have several conflicts to choose from and learn from the most promising one. This means changing the way the solver propagates (unit propagation is explained in Chapter 5).

The current workflow of the solver propagates all literals in a decision level until a conflict is found. Then it stops, learns from this conflict and continues its execution.

With these new modifications, all conflicts will be stored in an array, and once there are no remaining clauses that can propagate in the current decision level, we are going to query the solver for the most promising conflict through the Python C++ API, among the ones found and learn from this conflict.

It is vital to use unsatisfiable boolean formulas to learn from, as we need trace check files that come from proving the unsatisfiability of an instance to have a refutation tree. For this reason, we will first need to find the optimal makespan of all instances and try to solve them with an upper bound that we already know is unsatisfiable (meaning lower than the actual found best makespan), which we will use the makespan minus one.

Notice that training from unsatisfiable instances is coherent with the aim of speeding up the solver for two main reasons. On one hand, in an optimization process consisting of a sequence of satisfiability calls, as is our case, the hardest call is typically the one that certifies the optimality [39]. This call, which asks for a solution better than the optimal one, returns unsatisfiable. On the other hand, a CDCL-based solver performs local unsatisfiability proofs for every unsuccessful branch of the search tree, even if the original instance is satisfiable. In fact, one of the most successful branching heuristics, which is VSIDS [40], prioritizes the variables appearing in conflicts, motivated by the fact that solving an instance requires doing many unsatisfiability proofs.

8.2 Module Design

The workflow of the SMTAPI (C++ library) is described in the subsection 8.2.1. The GNN Dataset initialization is explained in the subsection 8.2.3. The communication between these two modules is explained in the subsection 8.2.2.

8.2.1 SMTAPI module

The first module in the project will be responsible for solving an RCP instance using our newly developed encodings. The structure of this module will be closely tied to the structure of the library SMTAPI explained in chapter 7. Using the up-bottom optimizer to check satisfiability will mean that our workflow will be the one represented in the activity diagram of Figure 8.3. The optimizer will obtain the pre-calculated upper and lower bounds, and if the lower bound is greater than the upper bound it will mean that we have found the minimum makespan, equal to the lower bound. For every iteration where this does not happen, the instance will be encoded (directly or narrowing the bounds with a decremented upper bound) and solved using the Maple solver. If the resulting instance is satisfiable we will continue lowering the upper bound. If the instance is unsatisfiable it means we have found the minimal makespan in the last satisfiable ub. Note that the upper bound is lowered in every iteration before the $ub < lb$ check.

The stopping reason will be either that the upper bound we are checking is lower than the pre-calculated lower bound, which means that we already know it is the optimum makespan, or the instance we have tried to solve is unsatisfiable, which naturally means that all encodings with a lower makespan will also be unsatisfiable.

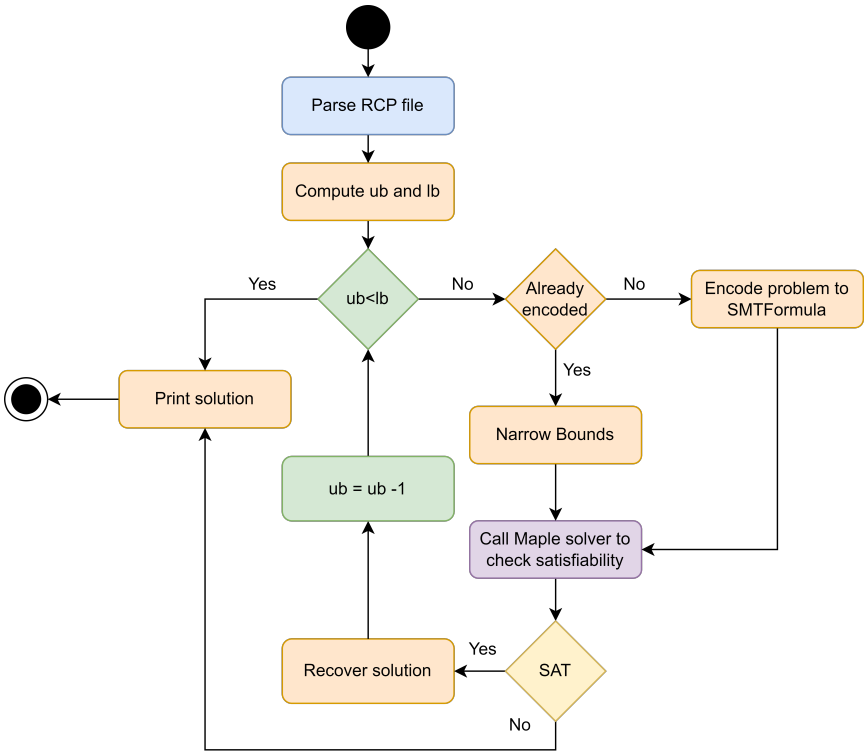


FIGURE 8.3: Activity diagram that represents the workflow of data inside the SMTAPI.

The other decision point in the activity diagram is the "Already Encoded" node. If we had already encoded the same instance in a higher makespan it means we don't have to do it again, but we can simply narrow the bounds, as we explained in the previous chapter, by adding the appropriate clauses that enforce the new upper bound for all the activities.

Both encodings will have the same workflow represented in Figure 8.3, and the difference will only be in the "Encode problem" node, as they will use different encodings, the "Narrow bounds" node, as they may have a different way to narrow them (they will probably not but they should be defined separately for good software practices), and a different parser. Thanks to the design of the SMTAPI this will be straightforward.

8.2.2 Converting the solving results to actual instances

Once we have the DIMACS file and the trace check file, we have to convert this into a single file that can be read by the GNN Module. This is where the `generateTrainFiles` module comes in. This module gets these two files as input and generates a `.train` file.

The `.train` file is a plaintext file designed for the purposes of this project that contains more information than a DIMACS file. This file includes a line for each clause in the original DIMACS file. Although the DIMACS file can have more than one clause in each line, the `.train` file contains only one clause in each line (the set of literals separated by a space), followed by a 0 and then the quality of the clause.

Creating this module that unifies these two files outside of the GNN module is a design decision that keeps the deep learning module cleaner.

8.2.3 GNN Module

The GNN Module is the module that is in charge of instantiating the graph instances and combining them in a dataset, training the Deep Learning model, validating it, saving the models to a folder, and loading them to be used from the SMTAPI.

The first part of the GNN module will be creating the GNN dataset. This starting point assumes that we have a folder that contains all the instances we want to train from. This means that for each instance we must have a `.train` file and a `.features` file. The `.features` file is a file containing one variable in each line, followed by its features separated by a single space. The `.train` file will follow the structure described in the previous section. Using the data from these two files, we are going to use PyTorch Geometric's HeteroData to implement the dataset, as we have different types of node types (variables and literals).

Training a Neural Network is a process that involves different steps, illustrated in Figure 8.4. Firstly, we have the raw data, which in the global process of this project, will be the RCP instance. Our feature selection process involves preparing the data files that will serve as the model input, and choosing which features we will use for each variable. This will result in the `.train` file and the `.features` file. All the instances we have to train will have to be split into a training set, a validation set and a test set. This is very important because otherwise we risk overfitting the model to the instances we used to train, thus capturing the noise and becoming too complex to generalize to unseen data. The training process will be done using the training set, in several epochs (complete passes through the whole training set), and using a particular loss function and optimization algorithm.

Once we have our model trained, we will validate it, modifying its hyperparameters and architecture until we are satisfied with its performance, and finally save it. Later on, we can test it with unseen data (test set) to assess its final performance. When we are confident with our model's performance, we will be able to use it by querying it from the Maple solver thus using new data to generate labels.

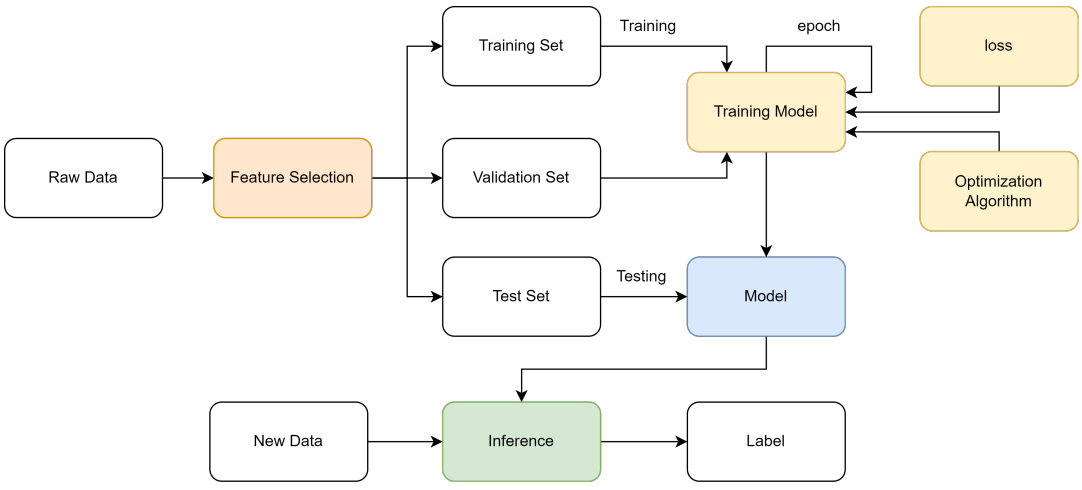


FIGURE 8.4: Scheme of the process of training and using a Neural Network.

Shaping this complex network will involve designing the architecture of the model, modeling the training workflow, and deciding the validation schemes to use.

Firstly, the model architecture. To build a Graph Neural Network we are going to use message passing layers. There exist many types of message passing layers that can be used with heterogeneous graphs. For the architecture of this project, we are going to use the SAGEConv layers from William L. Hamilton et al. (2018) [41] as well as the GraphConv layer from Christopher Morris et al. (2018) [42].

SAGEConv stands for Sample and Aggregate Convolutional. It uses the mean of the features of the neighbours and it aggregates them with the current node features as can be seen in Figure 8.5. SAGE layers are a type of message passing layer designed to be used in large graphs, and they have been proven to efficiently generalise to unseen data, which is what we are looking for. Moreover, this layer was already implemented in the Pytorch Geometric module we are using. In addition, using this type of layer has also implementation benefits, as they enable us to work with different-sized graphs, as they do not require to initialize their input size, which is perfect for our implementation.

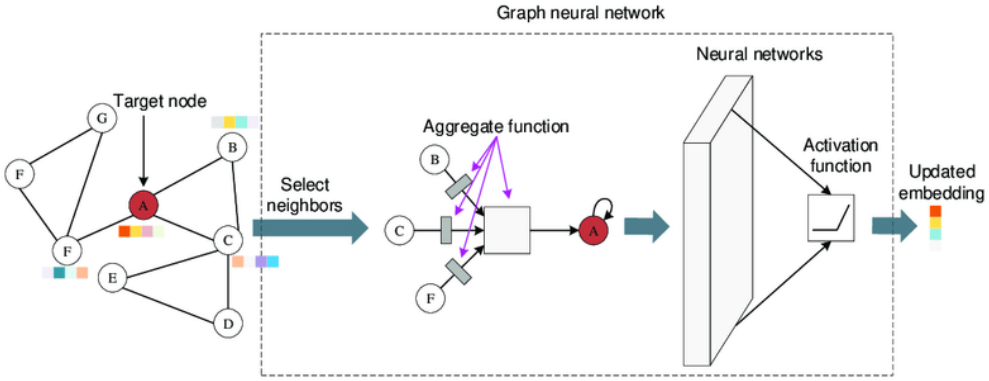


FIGURE 8.5: Diagram of a SAGEConv layer in a Graph Neural Network extracted from "Exploring SageConv: A Powerful Graph Neural Network Architecture" by Sahil Sheikh in Medium [43].

The GraphConv layers are a variant of convolutional neural networks that are designed to work with graphs. They have been chosen because they are a particular implementation of a message passing layer that although not being as efficient as SAGEConv when working with large graphs, in return they operate on the entire neighbourhood when doing the aggregation operations. They are most suited for problems where the entire graph is known, as is our case.

Apart from the convolutional layers, we use a linear layer to play the role of the last layer. This layer will have the same input size as the hidden layers, and it will output a single node, which will be our network's output.

There are also two activation functions: the ReLU function and the softmax function. The ReLU function or Rectifier function cancels any negative output by returning zero, and outputs the input value if it is positive. It is widely used in all types of neural networks as it helps overcome the vanishing gradient problem. This problem happens when the gradients of the loss function become very small when

they are propagated backwards during the training, which often happens with activation functions such as the sigmoid or tanh. The ReLU activation function can be expressed as follows:

$$\text{ReLU}(x) = \max(0, x)$$

The softmax function has the particularity of mapping any input to values between zero and one. Paired with the BCELoss function, which only accepts values between zero and one, it is widely used in many recommender systems.

The architecture of the Graph Neural Network will be the one that can be seen in Figure 8.6. Note that the edge information will go directly as an input to all the Graph convolutional layers as the edge information is needed to perform the neighbourhood operations. The feature information is updated on each pass through the different layers and activation functions. The input size of the first SAGEConv layer can vary, as previously explained, but the number of output layers of the SAGEConv layer, as well as the input and output of the second and third layer will be the defined number of hidden layers. As previously explained, the role of the last fully connected layer (or linear layer) will be to transform the number of layers outputted by the last ReLU into a single node, which will be the one that will indicate the predicted probabilities of a clause.

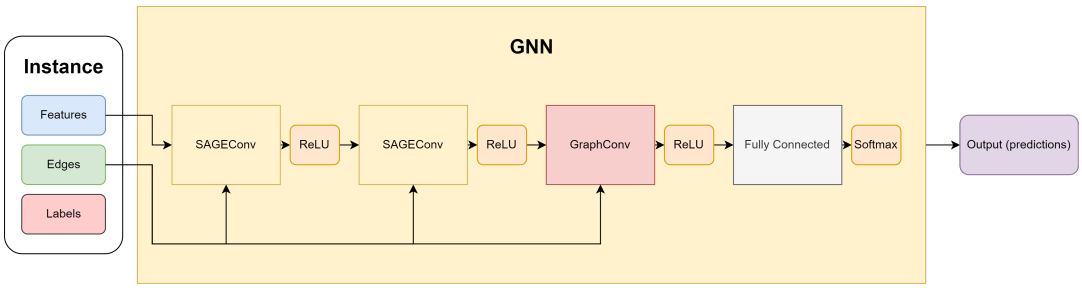


FIGURE 8.6: Diagram of the architecture of the GNN.

The training workflow will involve iterating for a specified number of epochs. During each epoch, the process will loop over all the training and testing batches to simultaneously train and validate the model using unseen data. Within each iteration of the training batches, each batch will be passed through the model, and the model's weights will be updated using the loss gradient.

8.2.4 GNN Validation

To be able to confidently use the ML model, we need to find a method to validate it by itself without using the SMTAPI. The ultimate goal of our predictor is to select the best one of the specified conflicts. This is where the measure Precision@K comes in. Precision@K measures the proportion of relevant items among the top k items recommended [44]. Although this is not a metric designed to train classifiers, we needed to find a metric that could be used in both a regressor and a classifier to be able to compare them. This being a metric used in recommenders (which is the ultimate goal of our model) is what made us decide to go with it.

$$Precision@K = \frac{|\text{Top K clauses predicted} \cup \text{Top K clauses labeled}|}{K}$$

Having chosen the metric we still have to decide how we are going to select the conflicts to choose from. This selection will be done at random, but it would be interesting to see how it would change if the selection was done by the solver.

Another measure that is often used together with Precision@K is Recall@K. While Precision@K focuses on determining how many of the recommended items are relevant compared to the number of recommended items, Recall@K compares it to the total of relevant items. For this reason, this metric will give us another insight into our model, specifically in the number of clauses predicted to have good quality that have not been selected by our GNN model.

$$Recall@K = \frac{|\text{Top K clauses predicted} \cup \text{Top K clauses labeled}|}{|\text{Relevant clauses}|}$$

8.2.5 Final Workflow

The final workflow of the SMTAPI is represented using an Activity diagram in Figure 8.7. The reason why we only generate the feature file when we encode a problem is that it is the only moment when we introduce new variables, so adding new clauses as the `Narrow bounds` function will never add or modify the existing variables. As we are only adding variable features it will not be necessary to modify the feature file when adding new clauses, so it can be generated only when a new encoding is created from scratch.

When consulting the GNN model, there are going to be three ways in which it will receive information. Firstly the DIMACS file represents the encoded instance. It is vital to create a Graph representation of the problem as it represents the relations between the literals and clauses and the structure of the instance. Secondly, the feature file that includes all the selected features of variables, meaning the RCPSP information we have and we will use to predict the best conflict. Finally, it will also receive the array of conflicts to choose from, which will be sent using the Python C++ library directly when invoking the Python script.

The output of the call to the model will be simply an integer that corresponds to the index of the clause with the highest quality among the conflicts it received.

To select which is the best conflict to choose from we will be calling a Python function stored in a Python file. This function will receive as a parameter the name of the instance (to know which files contain the DIMACS and the feature file), as well as the array of conflicts the solver wants the model to choose upon. The script should load the instance, load the model, consult the model with the instance, and select the clause that has the highest predicted value from the clauses present in the conflict array. Then, it will answer with the index this conflict occupied in the original array.

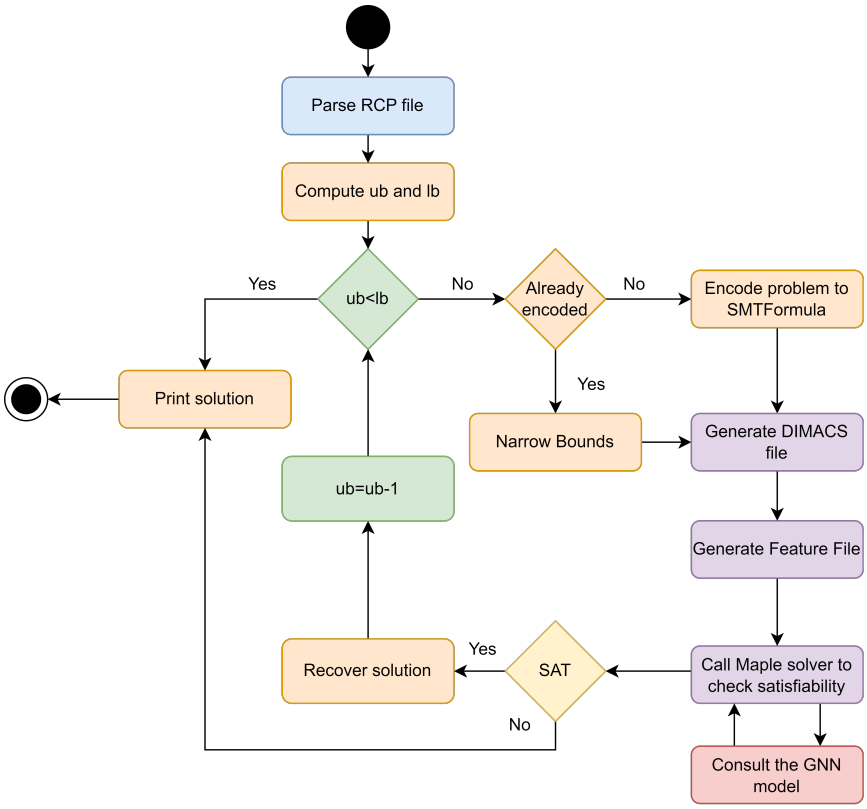


FIGURE 8.7: Activity diagram of the final workflow. The colour code of the image is the following: Encoding, Encoder, Solver, Optimizer, Main, GNN model

Chapter 9

Implementation and Deployment

9.1 The PRCPSP and MRCPSP instances

It is very important to understand that the only reason we are using the MRCPSP class is because RCPSP instances can be naturally saved inside MRCPSP instances, and it is what the code from the LAI group that we are using as a basis does. For this reason, we will not be creating a new class for RCPSP instances.

We define an instance of the PRCPSP using the following information:

- Number of activities
- Number of resources
- Demands: Array of Vectors, for each activity and resource, its demand.
- Successors: Array of Vectors, for each activity, what is the index of their successors.
- Duration: Array of integers, duration of each activity.
- Capacity: Array of integers, for each resource, its capacity.

Although an instance of PRCPSP can be defined only with these attributes, our PRCPSP instance has other attributes that can help with the resolution like the precalculated extended precedences.

To define an MRCPSP we have the same attributes as a PRCPSP instance with the following changes. Firstly, the number of demands is an array of arrays of vectors as we have to store a different demand for each activity, mode and resource. Secondly, the duration is an array of vectors, as we need to store the duration for each activity being executed in each mode. Finally, we also need an array to store the number of modes each activity has.

9.2 RCP to PRCPSP Parsers

To create the instances of PRCPSP problems and be able to encode them we need a parser that takes in an RCP file and outputs an instance. This section describes the created parsers.

```
32      4
12      13      4       12
0       0       0       0       0       3       2       3       4
8       4       0       0       0       3       6       11      15
4       10      0       0       0       3       7       8       13
6       0       0       0       3       3       5       9       10
3       3       0       0       0       1       20
8       0       0       0       8       1       30
5       4       0       0       0       1       27
9       0       1       0       0       3       12      19      27
2       6       0       0       0       1       14
7       0       0       0       1       2       16      25
9       0       5       0       0       2       20      26
2       0       7       0       0       1       14
6       4       0       0       0       2       17      18
3       0       8       0       0       1       17
9       3       0       0       0       1       25
10      0       0       0       5       2       21      22
6       0       0       0       8       1       22
5       0       0       0       7       2       20      22
3       0       1       0       0       2       24      29
7       0       10      0       0       2       23      25
2       0       0       0       6       1       28
7       2       0       0       0       1       23
2       3       0       0       0       1       24
3       0       9       0       0       1       30
3       4       0       0       0       1       30
7       0       0       4       0       1       31
8       0       0       0       7       1       28
3       0       8       0       0       1       31
7       0       7       0       0       1       32
2       0       7       0       0       1       32
2       0       0       2       0       1       32
0       0       0       0       0       0
```

FIGURE 9.1: Example of an RCP file that belongs to the first instance of the J30 dataset. The first line contains the number of activities followed by the number of resources. Then the capacity for each one of the resources. Next a line for each activity containing: one column for its duration, one column for its demand for every resource (ordered), the number of successors, and finally the index of each successor. Note that the first and last activity have duration d=0 and their demands for all resources are also 0.

9.2.1 The RCP parser

The objective of this parser is to read an RCP file and create an equivalent PRCPSP instance. A PRCPSP instance inside an RCP file is defined by the following parameters:

- Number of activities to schedule
- Number of resources available
- Capacity of each one of these resources (renewable, available at each time instant)
- Activities with their duration, demand for each resource, and successor activities.

All these parameters are specified in the RCP file, and the only purpose of the parser is to store them and act as a middleman between the RCP file and the encoding. An example of an RCPSP file to understand the structure appears in Figure 9.1.

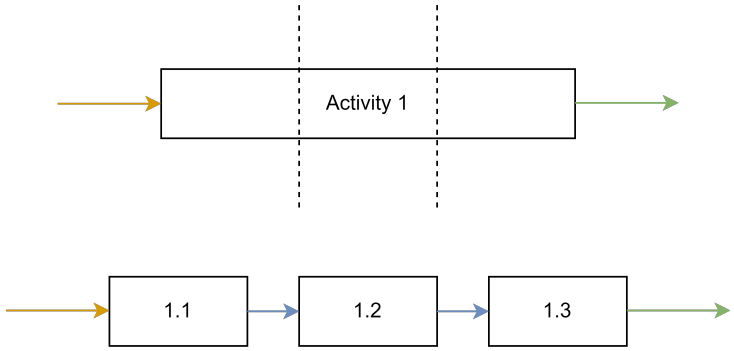


FIGURE 9.2: Visualisation of the precedence relations of the activities in the RCP to RCPSP as PRCPSP parser. The PRCPSP "Activity 1" is split into three RCPSP sub-activities 1.1, 1.2 and 1.3, of duration 1. The orange and green arrows are the precedences that already existed in the original PRCPSP activity. The blue arrows are precedences added to avoid sub-activity concurrence.

On the other hand, we have also created a parser that, instead of returning an instance of the PRCPSP class, returns an instance of the MRCPSP class. The way these two encodings store the data is very similar (view section 9.1), so the parser is exactly the same. It calls their interfaces to set the capacity of the resources, set the demands of the resources etc.

9.2.2 The RCP to RCPSP as PRCPSP parser

This parser is less intuitive than the previous one. The objective is to parse the RCP file into an instance of RCPSP which encodes the original PRCPSP problem (recall the reduction from PRCPSP to RCPSP explained in Reduction 8.1.1). This means that instead of creating the number of activities specified in the RCP file, we are going to create as many activities as the sum of the durations of the RCP specified activities plus two (these two being the first and the last activities that have duration 0). This way we will be able to solve the instance using an RCPSP encoding and get the solution as a PRCPSP problem, and we will be able to compare it to our other encoding and see how it performs. Figure 9.2 illustrates how an activity is split using this transformation.

To create this parser we are going to do it in two steps. In the first step, we are simply going to read the RCP file and store the attributes read in an array of vectors of successors, an array of vectors of demands and a vector of capacities, similarly as we did when parsing PRCPSP. In addition, we are also going to store the first and last sub-activity index that corresponds to every initial activity. This will help us add the precedences during the second step. The last activity index will be equal to the first activity index plus the duration (meaning that it will be a unit greater than the actual last sub-activity in all cases except when the duration is 0). We also keep a count of the number of activities we should be getting. To do so, we initialize a counter to 0 and add the duration if it is greater than zero or else we add 1.

The second step will consist of iterating from 0 to the number of activities specified by the RCP file (before splitting activities). For all sub-activities $A_{i,j}$ that compose each initial relevant activity A_i we set its duration d=1 and its demands to the demands of the original activity. If the activity was not relevant ($d = 0$) we add the

same activity with duration $d = 0$ and the same precedence rules. Then, for all the new sub-activities $A_{i,j}$ except the first one ($j > 1$), we add the previous one $A_{i,j-1}$ as its precedent. Finally, for every activity, if there existed a precedence relation $i \rightarrow s$ we add a new precedence relation $A_{i,d} \rightarrow A_{s,1}$. To do so, we will need to check if the first and last sub-activities are equal (to know if their duration was zero). If they are not, we will use the last sub-activity minus 0 as explained.

9.3 Encodings

In this section we present two different encodings, the PRCPSP custom-made encoding and the PRCPSP dummy encoding. Both encodings use the same viewpoint which consists of the following two kinds of variables:

- $s_{i,t}$: Starting time variables, true iff the activity i starts or has started its execution at time insant t. These variables are defined for every activity $i \in A$, and for every $t \in [ES(i), LS(i))$.

- $x_{i,t}$: Execution time variables, true iff the activity i is being executed at time period t. These variables are defined for every activity $i \in A$, and for every $t \in [ES(i), LC(i))$.

Note that start time variables refer to time instants whereas execution variables refer to periods, as illustrated in Figure 9.3.

Table 9.1 summarises the nomenclature of the input data and preprocessed data defined in Chapter 5, as well as the viewpoint variables of the encodings.

TABLE 9.1: Summary of nomenclature.

Parameters	
N	Number of non-dummy activities
$V = \{0, \ldots, N+1\}$	Set of activities
$A = \{1, \ldots, N\}$	Set of non-dummy activities
p_i	Duration of activity i
$G = (V, E)$	Precedence graph
$(i, j) \in E$	End-start prececence
$R = \{1, \ldots, v\}$	Set of resources
B_k	Capacity of resource k
$b_{i,k}$	Demand of activity i for resource k
UB	Upper bound on the makespan
$T = \{0, \ldots, UB-1\}$	Set of all times between 0 and UB-1
Pre-computed sets and functions	
$G^* = (V, E^*)$	Extended precedence graph
$(i, j, l_{i,j}) \in E^*$	Extended precedence with lag $l_{i,j}$
$ES(i)$ and $LS(i)$	Earliest and latest start time of activity i
$EC(i)$ and $LC(i)$	Earliest and latest close time of activity i
Variables used in formulations	
$s_{i,t}$	Activity i has started at time t
$x_{i,t}$	Activity i is running at time t

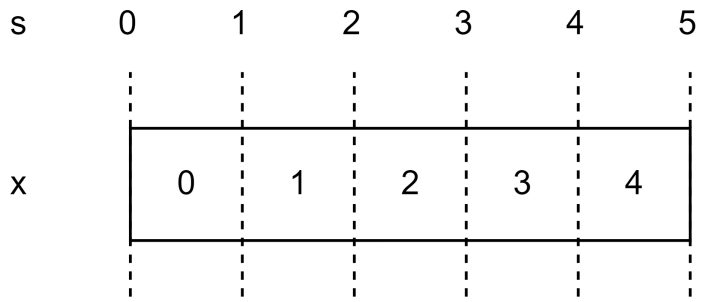


FIGURE 9.3: Comparison between the x variables and the s variables of a single activity of duration 5. Note that, conceptually, the sub-index of variables s_i identify possible start times, which correspond to time instants. In contrast, the first sub-index of variables $x_{i,t}$ identify running periods, being period i the one comprised between instants i and $i+1$.

9.3.1 PRCPSP custom-made encoding

The PRCPSP custom-made encoding is the encoding we have developed to directly encode PRCPSP instances into SAT considering that any activity can be preempted.

As we can see in the RCP file (Figure 9.1), the first and last activities of this encoding are dummy activities. They are activities with duration 0 that use no resources because they are used to facilitate the encoding process. They are always defined in the RCP files and their purpose is to act as the first activity to be executed (meaning that no activity can start before its beginning), and the last activity to be executed (no activity can be scheduled after it ends).

In the following sections we describe the constraints of our encodings, which are represented as clauses or implications (which are trivially convertible to clauses applying standard equivalences).

Starting times of the activities

To define the starting times of the activities we are going to use the execution time variables. If an activity is being executed it has already started:

$$\forall i \in A, \forall t \in [ES(i), LS(i)], \quad x_{i,t} \rightarrow s_{i,t} \tag{9.1}$$

This clause only forces starting times to be true when an activity is executed but not the other way around. For this purpose we introduce constraint (9.2): If in a specific time an activity has started but in the previous time it had not started, it means that it must be in execution in that instance.

$$\forall i \in A, \forall t \in [ES(i)+1, min(LS(i), LC(i)-1)], \quad s_{i,t} \wedge \neg s_{i,t-1} \rightarrow x_{i,t} \tag{9.2}$$

This clause will have to be different if the time we are considering is the earliest start time (ES) of the current activity, as variable $s_{i,t-1}$ will not exist, resulting in constraint (9.3).

Constraint (9.3) forces, on the first execution time of each activity ($t = ES(i)$) that if the activity has started it must be in execution in that time.

$$\forall i \in A, \forall t = ES(i), \quad s_{i,t} \rightarrow x_{i,t} \tag{9.3}$$

In addition, we have to ensure that once a variable that indicates that an activity has started executing is true, the rest of the variables belonging to this activity that refer to a later time are also true. This can be done by adding constraint (9.4).

$$\forall i \in A, \forall t \in [ES(i), LS(i) - 1], \quad s_{i,t} \rightarrow s_{i,t+1} \tag{9.4}$$

Ensuring its execution time is its duration

As we are working with preempted activities, our way to ensure that every activity i is being executed the number of time periods specified by its duration p_i will be to add the following cardinality constraint.

$$\forall i \in A, \quad \sum_{t=ES(i)}^{LS(i)-1} x_{i,t} = p_i \tag{9.5}$$

This constraint forces a set of variables to have exactly p_i variables that are 1 (i.e. true).

This kind of constraint is commonly known as an Exactly-K constraint, which is a type of cardinality constraint. Different techniques to encode Exactly-K constraints to CNF exist. The encoding chosen for this project is the one described in Roberto Asín et al. (2011) [45].

Precedence

Precedence constraints are added to force any precedence relation (i, j) to be respected, making sure that activity i can't be in execution after activity j starts. We can encode this relation using constraints (9.6) and (9.7).

If both an activity and its successor can be scheduled, if the successor has started, the predecessor can't be scheduled.

$$\forall (i, j) \in E, \forall t \in [max(ES(i), ES(j))..min(LC(i), LS(j))], \quad s_{j,t} \rightarrow \neg x_{i,t} \tag{9.6}$$

If the predecessor could be scheduled but the successor can no longer be scheduled, the predecessor must not be scheduled.

$$\forall (i, j) \in E, \forall t \in [LS(j)..LC(i)], \quad \neg x_{i,t} \tag{9.7}$$

Renewable resource constraints

The last indispensable set of clauses we must introduce are the Renewable Resource Constraint clauses. These clauses ensure that there is no instant of execution where the sum of resource requirements by all activities that are being executed exceeds the specified capacity of each resource.

$$\forall k \in R, \forall t \in T, \quad \sum_{\substack{i \in A \\ t \in [ES(i), LC(i)-1]}} b_{i,k} \cdot x_{i,t} \quad \leq B_k \tag{9.8}$$

The constraint described in (9.8) is a pseudo-boolean constraint. To encode this constraint we are going to use the SAT encoding based on Multi-Valued Decision Diagrams (MDDs) from Miquel Bofill et al. (2020,2022) [10, 46].

Extended precedences

To compute extended precedences we use a variation of the Floyd Warshall algorithm. This variation consists of modifying the algorithm to find the longest path between each pair of nodes instead of the shortest (we can do so because the precedence graph is a directed graph with no loops). We will therefore have to initialize the extended precedences to the minimum instead of the maximum.

To enforce these extended precedences we introduce constraints (9.9) and (9.10), that use the computed values.

For every extended precedence $(i, j, l_{i,j}) \in E^*$, if activity j has started its execution at t, activity i has started its execution at $t - l_{i,j}$.

$$\forall (i, j, l_{i,j}) \in E^*, \forall t \in [max(ES(i) - l_{i,j}, ES(j))..min(LC(i), LC(j))], \quad s_{j,t} \rightarrow s_{i,t-l_{i,j}} \tag{9.9}$$

For every extended precedence $(i, j, l_{i,j}) \in E^*$, if $t - l_{i,j} \leq ES(i)$, we have to make sure that activity j is not being executed before its predecessor.

$$\forall (i, j, l_{i,j}) \in E^*, \forall t \in [ES(j), ES(i) - 1], \quad \neg s_{j,t} \tag{9.10}$$

Narrowing the bounds

As explained in the previous chapter, narrowing the bounds means lowering the expected makespan of an instance without having to re-encode the whole instance from scratch. We know that the makespan of the instance is defined by the start time of the last activity of the problem, whom all other instances precede.

Therefore, we will introduce the following constraints to stop any activity from being executed in the time instants after the new upper bound.

The last activity should have started in the new upper bound.

$$s_{N+1,UB} \tag{9.11}$$

No activity should be in execution after the current upper bound (The times are between 0 and $UB - 1$, so UB is the first time after the upper bound). Here, $lastUB$ refers to the previous UB that existed before narrowing the bounds.

$$\forall i \in A, \forall t \in [UB, lastUB], \quad \neg x_{i,t} \tag{9.12}$$

9.3.2 PRCPSP dummy encoding

We refer to the PRCPSP dummy encoding as the result of, first, reducing a PRCPSP instance to RCPSP as explained in Reduction 8.1.1, and second, encoding the RCPSP instance to SAT.

The encoding of RCPSP shares the constraints from the custom-made PRCPSP encoding that do not express relations between variables $s_{i,t}$ and $x_{i,t}$. Namely, we reuse the constraints: start time variable consistency constraints (9.4); renewable resource constraints (9.8); extended precedence constraints (9.9) and (9.10); and narrow bounds constraints (9.11) and (9.12). The remaining constraints are defined as follows.

Starting times of the activities

As this is an RCPSP encoding, we are going to ignore that we know that all activities will have a duration of 1 or 0 and generalize the encoding for any duration.

If in a specific time t an activity has started but in $t - p_i$ time it had not started, it means that it must still be in execution. This clause is the non-preemptive counterpart to clause (9.2).

$$\forall i \in A, \forall t \in [ES(i) + p_i, min(LS(i), LC(i) - 1)], \quad s_{i,t} \wedge \neg s_{i,t-p_i} \rightarrow x_{i,t} \tag{9.13}$$

In a time t after the latest start time, if an activity had not started in $t - p_i$, it means that it will be in execution, as it can't have started later than $LS(i)$.

$$\forall i \in A, \forall t \in [max(ES(i) + p_i, LS(i)), LC(i) - 1], \quad \neg s_{i,t-p_i} \rightarrow x_{i,t} \tag{9.14}$$

For all times that are previous to $ES(i) + p_i$, if the activity has started it means that it is being executed. We need this clause because the constraint (9.13) is only added when the time is after $ES(i) + p_i$.

$$\forall i \in A, \forall t \in [ES(i), min(ES(i) + p_i, LS(i), LC(i) - 1)], \quad s_{i,t} \rightarrow x_{i,t} \tag{9.15}$$

In all times between the latest start time and the earliest start time plus the duration of an activity i (if there are any), we know that the activity has to be in execution, as

the activity must have started (it is after $LS(i)$) and it can't have finished (it is before $ES(i) + p_i$).

$$\forall i \in A, \forall t \in [LS(i), min(ES(i) + p_i, LC(i) - 1)], \quad x_{i,t} \tag{9.16}$$

In this encoding we don't need to add Exactly-K constraints as we did in the custom made encoding because not allowing the preemption of activities means that we can control that an activity is being in execution in a specified time frame simply by using constraints (9.13), (9.14), (9.15) and (9.16).

Precedence

A smart encoding for precedence relations $(i, j) \in E$ in RCPSP problems is to encode the constraint that, if activity j has started at time t, forces its predecessor i to have started at time $t - p_i$.

$$\begin{aligned} &\forall (i, j) \in E, \forall t \in [min(ES(i), ES(j) - p_i), max(LS(i), LS(j) - p_i)], \\ &s_{j,t+p_i} \rightarrow s_{i,t} \end{aligned} \tag{9.17}$$

Notice that in Constraint (9.17) there are some values of t for which variables $s_{i,t}$ or $s_{j,t}$ may not be defined. In particular if $t < ES(i)$ then $s_{i,t}$ is replaced by a false constant, and if $t > LS(i)$ then $s_{i,t}$ is replaced by a true constant. We do the same for $s_{j,t}$.

9.4 Recovering the PRCPSP solution from the RCPSP instance

Up to this point, we have described the process we use to parse the RCP file into an RCPSP instance by splitting the activities, as well as the RCPSP constraints we use to transform the instance into a boolean formula. Solving this boolean formula will result in an assignment for all the variables of the RCPSP encoding. From this variables we obtain a solution for the RCPSP instance. Therefore there is a final step that consists of translating this solution to the original PRCPSP problem.

The solution to the PRCPSP problem is defined by a vector of start times $starts_i$ and a matrix of execution periods $execution_{i,t}$. On the other hand an RCPSP solution is defined by a vector of start times $starts'_{i_j}$, knowing that every sub-activity i_j has a duration of exactly 1, where activities $i_1, \ldots, i_{p_i}$ are the sub-activities resulting of decomposing activity i in the reduction process 8.1.1. Then, we define:

$$starts_i = starts'_{i_1}$$

$$execution_{i,t} = \left(starts'_{i_1} \vee \cdots \vee starts'_{i_{p_i}}\right)$$

This will be done only before printing the optimal solution, as it is the only moment when we are interested in what the solution of the original problem is. That means

that the solution that will be printed by the SMTAPI will directly be the solution to the original RCPSP problem.

9.5 Validating the PRCPSP encoding

Since we are dealing with an NP problem, its validation can be done in polynomial time. To validate the correctness of the results of both encodings, a PRCPSP checker has also been developed. This checker receives the output of the optimizer and validates that the returned solution is correct.

Validating the correctness of a PRCPSP solution involves the following steps:

1. Parsing the DIMACS instance and storing it as a PRCPSP instance.

2. Parsing the output of the optimizer by filling a list of starting times of the activities ($starts_i$), ending times of the activities ($ends_i$) and matrix of execution of each activity ($x_{i,j}$).

3. Then we can start checking its correctness: firstly the renewable resource check will consist of verifying that the amount of a single resource used by all activities executed in an instant is not greater than the capacity of this resource. We iterate over every resource r and every time t in $[0, makespan]$, add the resources of the activities that are being in execution according to the matrix of execution, and check if the resulting value exceeds the capacity of r.

4. Next, the execution check: the number of times an activity is being executed must be equal to its duration. It means iterating over all activities and counting the times where the execution matrix indicates that the current activity is in execution.

5. Finally the precedence check: all precedence rules are respected. We iterate over all successor activities j of an activity i and check that if activity i has a duration $p_i > 0$, $starts_j > ends_i$, meaning that the predecessor has its last execution instant before the successor starts.

This validation scheme works with both the outputs of the PRCPSP dummy encoding and the custom-made encoding, so it means that we can create a script that executes an instance with both encodings, and checks their correctness after each execution. The name given to this script is `checkExec.sh` and its results can be seen in Figure 9.4. This script simply solves the same instance separately with the two proposed encodings and then calls SMTAPI's PRCPSP checker we have developed passing the name of the RCP file and the file where the solution is saved to for both solutions.

9.6 Testing both SAT-encodings in the cluster

The process of executing instances in the cluster is done through the queue manager SLURM. To ease working with the workload manager, several scripts were developed. Working using the cluster will mean that we will be able to solve several instances at the same time and independently. This is very useful but it comes with some challenges regarding concurrence. If we simply execute all the instances in a

```
solversjordi $./checkExec.sh 1 1
../instances/j30rcp/J301_1.RCP: OK in makespan 43
../instances/j30rcp/J301_1.RCP: OK in makespan 43
solversjordi $./checkExec.sh 2 6
../instances/j30rcp/J302_6.RCP: OK in makespan 47
../instances/j30rcp/J302_6.RCP: OK in makespan 47
solversjordi $./checkExec.sh 2 7
../instances/j30rcp/J302_7.RCP: OK in makespan 47
../instances/j30rcp/J302_7.RCP: OK in makespan 47
solversjordi $./checkExec.sh 2 8
../instances/j30rcp/J302_8.RCP: OK in makespan 54
../instances/j30rcp/J302_8.RCP: OK in makespan 54
solversjordi $./checkExec.sh 2 1
../instances/j30rcp/J302_1.RCP: OK in makespan 36
../instances/j30rcp/J302_1.RCP: OK in makespan 36
solversjordi $./checkExec.sh 7 2
../instances/j30rcp/J307_2.RCP: OK in makespan 42
../instances/j30rcp/J307_2.RCP: OK in makespan 42
solversjordi $./checkExec.sh 15 1
../instances/j30rcp/J3015_1.RCP:        OK in makespan 46
../instances/j30rcp/J3015_1.RCP:        OK in makespan 46
```

FIGURE 9.4: Example execution of the checkExec script.

folder using 22 parallel processes, all instances will be executed more than once. If we divide manually the set of instances, we may find that some of these divisions finish their execution far earlier than others, as some divisions may get more complex instances than others. For this reason we have developed a lock file scheme, that will lock an instance (meaning that no other process will be able to execute it) once a process takes hold of it and starts solving it.

The lock file scheme will consist of a folder named locks with the same structure as the folder where the results will be placed, but it will contain an empty folder for every instance that has started its execution. Checking if this folder exists would not be enough to prevent concurrency issues. Therefore, we use the atomic instruction `mkdir` and check if it succeeded, otherwise it means that the folder had already been created and the instance is already in execution.

To submit a set of jobs to the solver, four different scripts were used, `submitNJobs`, `runSolver`, `runSolverInstance` and `runOptimizerRedirect`.

9.6.1 The `submitNJobs` script

The first script used, `submitNJobs`, takes as an argument the number of jobs to submit ($NJOBS$), and creates $NJOBS$ tasks that will execute the file specified in the second argument forwarding the rest of arguments to this file being executed. This is the script responsible for calling sbatch to submit the jobs, and it does so using the following arguments:

- partition: exclusiu. Forces SLURM to assign the specified partition named "exclusiu" to the problem.
- time: 00-48:00:00. Sets a time limit for the total running time of the job allocation. In this case it is 48 hours, but it will most likely never be reached thanks

to the particular timeouts we add to each execution independently.

- out & error: We specify the particular files that should store the out and error log files of the request. These files are not where the output of the makespan optimization is sent because we redirect their output to their own file but rather where any SLURM or intermediate script-related output is sent as well as any output we want to have from the intermediate scripts.

- ntasks: Maximum number of tasks launched by the current job. It is left as default (1) because we execute `sbatch` as many times as nodes exist in the cluster.

- job-name: job$i. The name that will appear when viewing the current jobs queued and in execution, we set it as job followed by the number of the task we are launching (between 0 and $NJOBS$.

The sbatch instruction with the previously specified arguments will be executed $NJOBS$ times creating then as many jobs as we requested by parameter, which will be as many as the number of nodes in the cluster.

An example of a call to the `submitNJobs` script can be seen in Figure 9.5

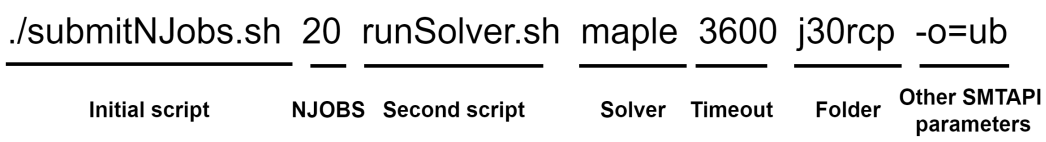


FIGURE 9.5: Breakdown of the different arguments of the call to submit jobs.

9.6.2 The `runSolver` script

The file passed in the `sbatch` execution is the script `runSolver.sh`. Therefore this script will be called $NJOBS$ times and will be the one responsible for enforcing the lock scheme.

The work done by this script is the following:

1. Make sure the folder where the execution results should be put exists and create it otherwise.

2. Prepare the arguments to be passed to the `runOptimizerRedirect.sh` script.

3. Iterates over all the files in the folder. If the lock file does not exist it creates it, otherwise it goes to the next iteration.

4. Calling `runOptimizerRedirect.sh` using the `time` utility in Linux to measure the execution time as well as the `timeout` utility to stop the execution after the specified timeout received by the parent script, which in the call represented in Figure 9.5 would be 3600 seconds.

5. Finally it passes the `runOptimizerRedirect.sh` script to the timeout utility with the following arguments in the specified order:

    - Out: File that will contain the output of the SMTAPI.

- Err: File that will contain the error output of the SMTAPI and also where the results of the `time` instruction will be saved due to a redirect.
- Solverbin: The solver to be used by the SMTAPI.
- Instance: The current instance we want to solve.

The rest of the parameters are the ones received by the parent script and some parameters that refer to the SMTAPI execution that fulfill generic purposes like deactivating the printing of non-optimal solutions.

9.6.3 The `runSolverInstance` script

This is a variant of the `runSolver` script which does exactly the same as the original script but in addition to all the arguments passed down to the solver it also adds the upper bound of the current instance. The objective of this script is to make the SMTAPI call that only proves the unsatisfiability of the first unsatisfiable instance. We have previously explained why we are specially interested in the solutions for this call in the section 7.5.

To get the makespan of an instance these scripts need all solutions for the instances in the current dataset to be present in a CSV file in a specific directory.

To create this CSV file with all the solutions a new script was developed, the `toCSV` script, that simply iterates over all the solutions outputted by the cluster and extracts the execution time and makespan found. The outputted CSV will contain the identifier of the instance, followed by the solving time and the optimal makespan, or -1 if the instance has not been solved in the specified timeout. For this reason, for any new dataset we want to solve whose makespans are unknown to us, we first have to launch its execution to find them and then the execution to solve these unsatisfiable instances.

9.6.4 The `runOptimizerRedirect` script

Finally the `runOptimizerRedirect.sh` script calls the corresponding SMTAPI main (depending on what encoding we want to use) to find the best makespan. It passes as parameters the instance and solver specified, and the rest of the parameters it was given by its parent script, the `runSolver.sh`.

When doing this call it also does a redirection of the output and error streams to the files specified by the parent as the `out` and `err` files, which will be called the instance name ended by `.out` and `.err` respectively.

9.7 Generating the train files and obtaining the quality of a clause

The measure selected to define the quality of a clause is the number of times it appears in the trace check file generated by DRAT-trim 7.5. To parse this file, we have created the module `generateTrainFile`, which using the DIMACS file and the trace check file generates a new type of file, the `.train` file.

The `.train` file will contain all clauses that have appeared either in the trace check file or the DIMACS file, as well as its index of appearance, which will be the normalized number of appearances as originating clause in respect to the clause that appears the most. An *originating clause* in our context is a clause that has participated in obtaining another clause according to the trace check file.

Parsing the DIMACS file is straightforward. We declare a dictionary that will have the clauses as the key and the number of occurrences in the trace check file as the value. We are going to call this the `appearances` dictionary. Naturally, as for now we are only parsing the DIMACS file, we initialize every entry at 0.

Parsing the trace check file is more elaborated. Every line of the file describes a clause. Firstly, we have its index assigned by the DRAT-trim, followed by the literals that make up the clause, a 0 to indicate that the clause is over, and then the index of every one of its originating clauses. Finally, a 0 indicates that the line is over. The trace check file might not contain all the clauses that appear in the DIMACS file (because they are not used during the trace check), but it might also contain clauses that did not appear in the DIMACS file (because they were learnt during resolution). A comparison between a line in a DIMACS file and a line in a trace check file can be seen in Figure 9.6.

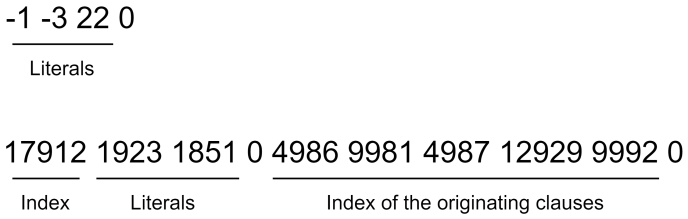


FIGURE 9.6: Line in a DIMACS file (top) in comparison to a line in a trace check file (bottom).

Next, we describe the implemented process of parsing a trace check file. A clause in this file might appear even before the literals that compose the clause do (as an originating clause). This would cause problems if we went with the same dictionary as the DIMACS file, so we are going to use two dictionaries. The first dictionary, the `literals` dictionary, will contain the clause literals indexed by the clause index in the trace check file. The second dictionary, the `occurrences` dictionary will store the number of occurrences of the clause also indexed by its trace check index. When we parse a line, the clause is added to the `literals` dictionary and if it was not already inserted we also add it to `occurrences` dictionary with 0 occurrences. Then, every originating clause is added 1 occurrence in the occurrences dictionary and initialized to 1 in the `occurrences` dictionary if it had not been already inserted.

When we finish filling these two arrays and parsing the trace check file, we will be sure that all indexes that are in the `literals` dictionary are also in the `occurrences` dictionary, and we can iterate over these two dictionaries for the same key and fill the `appearances` dictionary we filled when parsing the DIMACS file adding the clauses as key and the occurrences as value.

Finally, we normalize by the maximum number of occurrences and we can write the `.train` file. The normalization is done in this instant because it is part of the preprocessing of the data that will feed the neural network. Later, if we want to

have label 1 if they appear in the trace check and 0 otherwise we simply have to round up the values of the labels.

9.8 Dataset implementation

The dataset is created from the train and features files described in the previous chapter. These files are then parsed into what is called an instance. An instance is our internal representation of a train file and is the intermediate step before converting the data into tensors. A tensor is a multi-dimensional array used in various scientific computing fields, including physics, engineering, and computer science, to represent and manipulate data [47]. In the context of PyTorch, the main difference between a PyTorch tensor and a `numpy array` is that a tensor can run both in the CPU and the GPU, while a `numpy array` can only run on the CPU. For this reason, tensors are the data structure used to store deep learning data when it is passing through our GNN model. An instance in the context of this thesis is defined through the following attributes:

- Number of clauses.
- Number of literals.
- Array of clauses: An ordered list of all the clauses that form the instance. Every clause is represented by a `frozenset` of numbers, the Python representation for an immutable set of numbers. We chose this representation because a clause is a set of literals, which are represented as numbers. In addition, the `frozenset` can act as a key in a dictionary because of its immutability, and it is very useful to recover which literals appear in each clause once a clause has been selected.
- Array of features for each literal: The data the GNN trains with. Two-dimensional matrix containing the different features for each literal.
- Array of edges literal to literal: These edges appear between every literal and its negated counterpart in the form of an array of two position arrays. We use arrays knowing they will only have two positions because they will have to be transformed into tensors when we create the dataset. Note that as we explained PyG only has directed edges so every undirected edge will be transformed into two edges when we use the PyG transform `T.ToUndirected()(data)`.
- Array of edges literal to clause: These edges appear between every clause and its literals. Similarly to the other type of edges, they will be duplicated when we transform the instance to undirected, and they are an array containing arrays that only have two positions.
- Array of qualities: The label we are trying to predict. An occurrence for each clause and a value between 0 and 1.
- Dictionary of literals, which specifies for each literal what is its index in the features dictionary, which will be the same used to represent it in the arrays of edges.

This implementation, as explained in Chapter 5, uses a tripartite undirected graph. The library PyG implements undirected graphs by adding two directed edges for

each undirected original edge. These instances can be then stored inside an `HeteroData` class, which is the PyG implementation for heterogeneous graphs that can pass through a PyTorch Geometric neural network. This class stores the nodes and edges in dictionaries depending on their type. For every type of node, the tensor of features must be defined and not empty (because otherwise the node aggregation will cause an error). For this reason, we will add to the tensor of features of the clauses a one for every clause. For every edge type we have to define the edge index property, which is a tensor containing the index of two nodes that share the edge, which is equivalent to our arrays of edges. Finally, for any node type that has a label, we will have to define the `y` or label property, which will contain the expected label (array of qualities).

Since `HeteroData` will represent a graph, it will have nodes and edges. In our case they are divided into the following components:

- Literal nodes: A literal node is a node that represents a literal. The literal node has specific features. It is defined by the literal features as well as its index, that will appear in the edges.

- Clause nodes: A clause node is a node that represents a clause. A clause node does not have features but it will contain a 1 because of the implementation requirements. It is defined in the "literal to clause" edges by its index.

- Literal to literal edges: This is the equivalent to the array of edges literal to literal of the instance. It contains two entries for every literal to literal edge of the instance because these are directed.

- Literal to clause edges: This is the equivalent to the array of edges literal to clause of the instance. It contains two entries for every literal to clause edge of the instance because these are directed.

Figure 9.7 represents an example of `HeteroData` instance illustrating the different types of nodes and edges. Notice that every edge has two arrows representing that they are divided into two directed edges, one in each direction.

To store the different `HeteroData` instances, as we are working with a very large dataset, we are going to use PyTorch Geometric's Dataset class. This implementation of a dataset enables us to work in a specific workflow that loads the instances to memory one by one instead of loading the full dataset all at once. This is critical for the implementation as otherwise our computer wouldn't be able to train because of the size of the dataset.

We have to define the following operations to work with the PyTorch Geometric Dataset class:

- The init or constructor calls the parent constructor of the Dataset class and allows us to define attributes for the class. We are going to store the length of the dataset (the number of instances we load) as well as the folder we want to load them from and the root folder. The root folder is where we want to create the raw / processed file structure. This file structure will contain the raw and processed instances respectively to be loaded when they are needed one by one. This will allow us to not have to process the instances every time

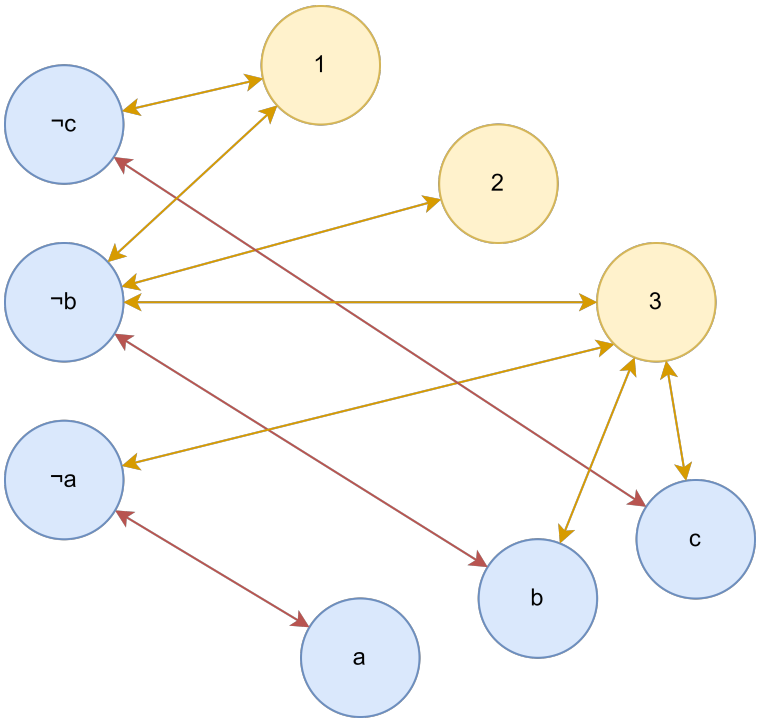


FIGURE 9.7: `HeteroData` representation of the boolean formula $(\neg c \vee \neg b) \wedge (\neg b) \vee (\neg b \vee \neg a \vee b \vee c)$.

we want to load an instance into memory.

When we load the instances from their original directory, they are stored in the raw directory. When we process the instances they will be serialized and stored in the processed directory.

- raw_file_names: This function has to return a filename that, if it exists, the loading of the instances will not be triggered.
- processed_file_names: Similarly to the previous function, if the filename returned exists in the corresponding directory does not trigger the processing of the instances.
- download: Loading the dataset. As our data is stored in a folder, this function loads the data and saves it into an individual data file for each instance to be used when needed.
- process: Do intermediate processing. We could call the functions of our instance that transform the loaded instances to `HeteroData` instances, but it would mean to load all files and add an extra step so we decided to do it directly in the download function.
- len: Returns the dataset length, in our case the attribute we declared.
- get index: We simply load the data file that contains the instance with the specified index inside the dataset.

Having implemented this Dataset class simplifies our workflow as we only have to give the instance of the dataset to the PyTorch geometric DataLoader and it will do

the batching for us. PyG offers different loaders [48] to divide the data into batches, but we selected the DataLoader because it merges the instances from a dataset into a mini-batch without performing any type of neighbour sampling or edge sampling, and loading the full instance in its assigned batch.

An important problem found during the training is that there are instances that result in an OutOfMemoryError if their train file is too big. This problem might be solved using neighbour sampling with mini-batches instead of training with full graphs but then the approach would change and it would not be on the scope of this project. It was decided that it is more important to have the full context of the graph, but it would be interesting to see how the process would change with other types of sampling. To solve this problem, we only consider instances that have a train file that is smaller than 20Mb, which excludes very few instances. Therefore, this will result in a smaller dataset than we would have liked.

9.9 Feature Selection and Data Preprocessing

During the feature selection process for our model, we started by creating a complete list of RCPSP data for each literal. Our goal was to gather useful information about literals that provided information that could not be extracted solely from the DIMACS file. Once we had gathered all the relevant RCPSP information, we removed any information that was redundant.

We define a literal by the characteristics of its associated variable. A variable can be a starting time variable, an execution time variable or an auxiliary variable introduced in the encoding of the cardinality or pseudo-boolean constraints. Every variable (and its corresponding literals) classified in the first two categories has an activity i and a time t associated. The rest will have dummy values for the categories that use the activity and time. Knowing this we defined the following features:

1. Is execution: 1 if the literal belongs to an execution time variable (see 9.3.1), 0 otherwise.

2. Is start: 1 if the literal belongs to a starting time variable (see 9.3.1), 0 otherwise.

3. Time t of the literal in relation to the upper bound (normalized between 0 and 1).

$$time\_ub = \frac{t}{UB} \tag{9.18}$$

4. Length of the literal activity i in relation to the longest activity.

$$length\_max = \frac{p_i}{\max_{j \in A}(p_j)} \tag{9.19}$$

5. Number of successors of the literal activity i in relation to the activity with the maximum number of successors.

$$succs\_max = \frac{|\{(i,j) \in E\}|}{\max_{a \in A} \left(|\{(a,j) \in E\}|\right)} \tag{9.20}$$

6. Minimum resource demand of the literal activity i in relation to the capacity of that resource.

$$r\_min = \min_{k \in R}(b_{i,k}/B_k) \tag{9.21}$$

7. Maximum resource demand of the literal activity i in relation to the capacity of that resource.

$$r\_max = \max_{k \in R}(b_{i,k}/B_k) \tag{9.22}$$

In the process, some features were discarded such as the polarity of the literals (positive or negative). The reason for this omission is that the polarity of the literals might add noise and it is not an RCPSP feature.

As explained in their descriptions, each feature has been normalized. This normalization process helps to homogenize the scale and distribution of the features of the literals across the dataset. Ensuring similar ranges for each feature will result in a more stable and efficient model training.

The variables that are not directly related to an activity and a specific time (the auxiliary variables of the pseudo-boolean and cardinality constraints) do not have specific values for some of the features. In this case, we are going to start by using `-1` as it is a value that will never occur by itself. We have tried other strategies like using `0`, `1` or using the mean of the rest of the instances, but we have seen no improvement.

9.10 Building the Model

Building the model consists of implementing the training workflow and the structure and layers of the model, as we have already defined the Dataset, and in the following section we construct the validation process.

The implementation of the architecture using PyTorch and PyTorch Geometric is straightforward. All the types of layers we considered are implemented so we have to import them and create the class that will form our network. To implement this class we have to implement on one side the `init` function, which instantiates the different layers we will use, and then the `forward` function that specifies the process the data goes through when passing through the network. Notice that the ReLU function is instantiated once and used twice. We can do that because it is an activation function and not a layer and therefore does not have weights.

The instantiation of layers in PyTorch works the following way: in the `init` function the first argument is its input size and the second argument is its output size. Specifying the SAGEConv to have input size `(-1,-1)` will make the layer adapt to the input size in both dimensions, as they are both `-1`. The activation functions do

not receive any parameters as they do element-wise operations, meaning they are applied to each value individually. In the forward function all layers receive the input or inputs as arguments and return their output. Notice that variable x changes its value every time the data passes through a layer or activation function as we had seen in the scheme of the architecture contrary to the edge index.

```
import torch
from torch.nn import Linear, ReLU, Softmax
from torch_geometric.nn import SAGEConv, GraphConv

class ClauseRec(torch.nn.Module):
    def __init__(self, hidden_channels):
        super(ClauseRec, self).__init__()
        self.conv1 = SAGEConv((-1,-1), hidden_channels)
        self.conv2 = SAGEConv((-1,-1), hidden_channels)
        self.conv3 = GraphConv(hidden_channels,hidden_channels)
        self.linear = Linear(hidden_channels, 1)
        self.relu = ReLU(inplace=True)
        self.softmax = Softmax(dim=1)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = self.relu(x)
        x = self.conv2(x, edge_index)
        x = self.relu(x)
        x = self.conv3(x, edge_index)
        x = self.relu(x)
        x = self.linear(x)
        return self.softmax(x)
```

Once we have designed the architecture of the GNN we have to implement the training process. Firstly we will select the hyperparameters. The loss function chosen to train the recommender is BCELoss (explained in Chapter 5). It is mostly used in classification tasks, which is why this will be the selected loss function when we create the classifier model. For the regression model we will use MSELoss or Mean Squared Error loss, which computes the mean squared error between the label and the model prediction.

The number of epochs we will use is 100 epochs, as we have experimentally seen that the results stabilize after this many epochs. The chosen optimizer will be the Adam optimizer [49]. This is a stochastic gradient descent optimizer that is widely used in recommender systems and has been proven to work well. We are going to use PyTorch Geometric's DataLoader to divide both the train and test sets into batches of 4 instances each. Then, in each epoch, we are going to traverse all the batches and compute the gradient of the loss using the `loss.backward()` function, and finally update the weights of the model using `optimizer.step()`.

After training each epoch we are going to traverse the validation batches and check how the model performs in the test set by using our validation function explained in the following section.

9.11 GNN Validation

The method to validate the model will be to iterate over the validation instances (not seen during the training process), and compute their Precision@K and Recall@K metrics for a specified K and a randomly selected set of conflicts.

To implement the Precision@K metric in Python, we start by selecting N conflicts at random and querying the model with the test instance. The next step will be to select the top K conflicts both according to the label and to the prediction. To do that we use list comprehension to sort the clause array according to the labels and predictions: `predictions_sorted = [x for _, x in sorted(zip(quals_pred, conflicts_pred))]` returns the list of predictions sorted, and then extracting the first K elements is straightforward.

Implementing the Recall@K metric is very similar to implementing Precision@K. We calculate the number of relevant items in the first K recommended items in the same way, and then instead of dividing it by the K, we divide it by the total number of relevant items. We will consider a relevant item an item that has a probability of more than 0.7.

To select the elements that appear in both arrays we can use the intersection function in Python, and measure the length of the resulting array. Finally we simply have to divide it by K and we have our Precision@K score.

This metric will be calculated iteratively and the presented value will be the mean of `num_iterations` iterations. The value for `num_iterations` will be 100 because we have experimentally shown that the metric stabilizes after this number of iterations. We will validate the model with a K=3 and use 10 conflicts as it is a good indicator of how the model is performing, but we will try other values for K when doing the validation of the final GNN model.

9.12 MAPLE - GNN Model communications

The modifications made in the solver can be classified into two categories: allowing the solver to continue searching for conflicts once a conflict has been found and directly communicating with the C++ model.

The original solver used stopped its execution once it found a conflict. Our approach is to continue looking for conflicts in the current decision level and query the model for the best clause to learn from after finishing. Firstly, we are going to create a flag that allows us to turn on/off this new approach. This means adding a new solving argument, which we are going to call `use-gnn`, that toggles the use of the model querying.

The next step is to modify the `propagate` method. Now, instead of returning the found conflict, we are going to store all the conflicts we find into a new vector of conflicts. Once the propagate method finishes, it will mean that all the conflicts in the current decision level have been explored.

The next step is to pass this vector of conflicts and query the model to choose among them. To call the model we will use the Python C++ library. The communication

workflow between the SAT solver and the GNN model consists of successive calls to the Python C++ library.

Firstly, the function calls `Py_Initialize` to initialize the interpreter. The next step is to load the module named `select_clause` using `PyImport_import`, which is the name of the Python script we are going to run. Then we retrieve the function `call` from the module loaded, which will be the function we are calling. This function receives the list of conflicts we are interested in as a parameter. Because the function has a single parameter, we use `PyTuple_New(1)` to initialize the parameter tuple the function is going to receive.

The next step is to prepare the list of lists of literals, which is straightforward using `PyList_New` and `PyList_SetItem` to create a list of lists of PyObjects (which are basically a string representing the literal). After adding this list to the tuple, we can call the function using `PyObject_CallObject(pFunc, pArgs)`, and store the return value of this function in a `PyObject`.

The output of the Python script is simply a long indicating the index the selected clause occupies in the vector of conflicts it has received. Finally, we call the function `Py_Finalize` to delete the interpreter and be able to create it again in the next call.

During all the code, there is a consistent checking for errors: checking if the objects are being correctly initialized, finalizing the Python execution and making sure a correct conflict is always returned. This error checking is not shown in Figure 9.8 because it would add a lot of noise to the diagram.

This C++ program receives the array `vec_conflicts`, which is an array that stores all the references to the clauses that caused a conflict. To access the clauses themselves we use the array `ca`, which is what the solver calls a `ClauseAllocator`. It stores all the clauses and is indexed by its reference. The function `obtain_literal_representation()` from the diagram simply receives a literal as a parameter and checks its sign and variable number to output the DIMACS representation of the variable by doing the following:

```
std::string literal = (sign(c[j]) ? "-" : "") +
    std::to_string(map[var(c[j])]+1);
```

Where `c[j]` is the internal representation of the literal that occupies the position j of clause c. The function sign returns true if the literal is negative, the var function returns the variable the passed literal comes from, and the map stores the number each literal was mapped to the last time the DIMACS file was outputted.

An important step for the communication between the solver and the Python module is to make sure that the variables that appear in the DIMACS file are the same that appear in the feature file, and in the same order. The MAPLE solver maps all the variables that are outputted in the DIMACS file to a subset of variables between 0 and n, where n is the number of variables after removing all variables that have already been assigned to false or that have been discarded by the solver through its inner processes. This is vital because we have to make sure to only output the features of the variables that participate in the DIMACS file and do it in the correct

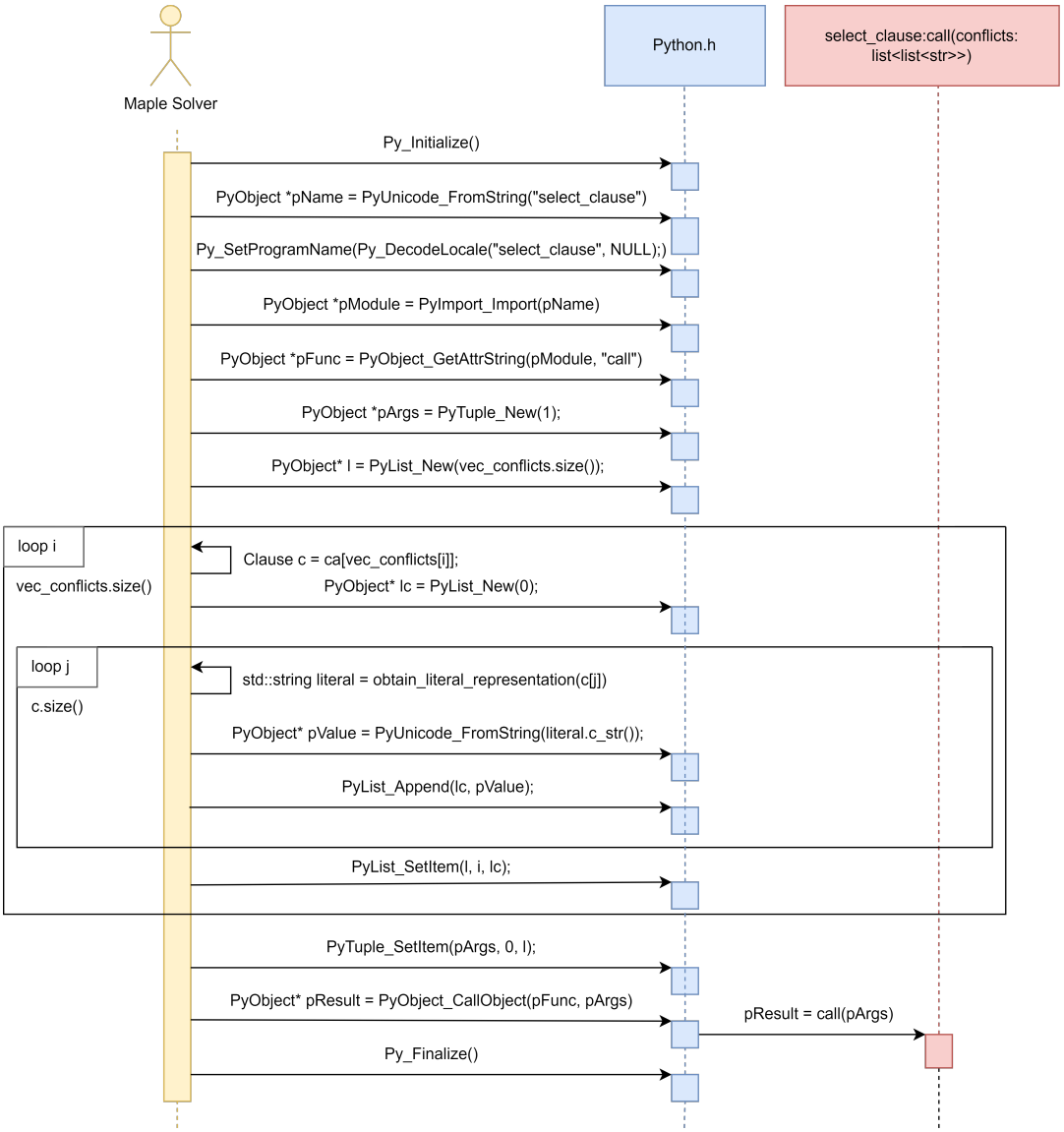


FIGURE 9.8: Sequence diagram of the calls the Maple Solver does to the Python C++ API and ultimately the GNN model.

order. For this reason, we have also modified the solver to save this variable mapping. When we output our feature file (right after the DIMACS file), the solver will load the mapping done the last time a DIMACS file was created, and will print only the new mapped indexes of all variables that appeared in that file. This is the reason why the indexes on the feature files may end up unordered, but this does not affect us.

Chapter 10

Results

In this chapter, we are going to go over the different experiments that we conducted. They can be classified into three categories. Firstly we have the comparison between the PRCPSP custom made encoding and the PRCPSP dummy encoding, both in time and in size of the encoding. Then, we compare the two developed GNN models, the Classifier and the Regressor in the metrics we defined, the Precision@K and the Recall@K. Finally we compare how the selected model performs when being queried by the SAT solver when solving PRCPSP custom made encoded instances through different solving metrics.

10.1 PRCPSP Encoding Comparison

To compare the two encodings developed we have decided to run all the instances of the datasets J30, J60, J90 and J120 and Pack. The "J" datasets are in increasing order of difficulty. The number after the "J" indicates how many activities to schedule they have without counting the first and last activities. This means that all instances in the J30 dataset will have 32 activities, in the J60 dataset they will have 62 activities etc. They also have a similar number of instances, the J30, J60 and J120 having 480 instances each, and the J120 having 600 activities. Finally we chose to add the Pack dataset to have more instance diversity.

The "J" datasets were first proposed in Kolisch, R. and A. Sprecher (1996) [29], and have since become widely used in the field of scheduling thanks to its great variety of instances. The Pack instances were presented in Jaques Carlier et al. (2003) [50].

To compare the results of these two datasets we will check how long it takes for the SAT solver in the cluster to recognize the unsatisfiability of the first unsatisfiable instance (i.e. the makespan minus one). To do this we will simply launch all the instances of the datasets first to obtain a CSV with their makespan, and later we will, as explained in the previous chapter, call the `runSolverInstance` script through the `submitNJobs` script to look for unsatisfiability in the time equal to the makespan. As the optimizer assumes the upper bound you give is correct, it directly encodes the problem for the upper bound minus one, thus encoding the first unsatisfiable instance.

For each instance we are interested in, on one hand, the number of instances that are solved with a specific timeout, and on the other hand, the size of the encoding (the number of variables and clauses of the generated DIMACS file).

The execution results for the custom-made encoding on a timeout of 10 minutes can be seen in Table 10.1, and the execution results for the dummy encoding on a timeout of 10 minutes can be seen in Table 10.2. In these tables, "#ins" is the number of instances of the specific Dataset, Solved is the number of instances that could be solved (proved minimum) with the specific timeout, "<1s" is the instances that have been solved in less than a second, "Vars" is the mean number of variables of all instances in the dataset that have produced a DIMACS file, and "Clauses" is the mean number of clauses of the instances of the Dataset that have produced a DIMACS file. Finally, the last column is the mean time of all instances in seconds.

Dataset	#ins	Solved	<1s	Vars	Clauses	Mean (s)
J30	480	423	369	6644	18673	9.65
J60	480	386	368	17088	47579	10.98
J90	480	378	163	73661	189383	10.67
J120	600	238	227	41168	112923	9.95
Pack	55	27	26	6991	19055	6.59

TABLE 10.1: Execution results for the custom-made encoding for all the datasets with a timeout of 10 minutes. #ins is the number of instances. <1s is the number of instances solved in less than one second. Mean(s) is the mean solving time in seconds.

Dataset	#ins	Solved	<1s	Vars	Clauses	Mean (s)
J30	480	433	379	7866	36873	4.68
J60	480	386	371	19702	93803	2.31
J90	480	378	210	81805	335918	6.71
J120	600	241	216	47409	247016	2.85
Pack	55	27	26	7866	27919	0.62

TABLE 10.2: Execution results for the dummy encoding for all the datasets with a timeout of 10 minutes. #ins is the number of instances. <1s is the number of instances solved in less than one second. Mean(s) is the mean solving time in seconds.

10.2 GNN Test results

Testing our model was done through the validation workflow described in Chapter 9. As we explained, the metrics used to compute the performance of our model are the Precision@K and the Recall@K. The following tables illustrates the results for different K using as training hyperparameters 64 neurons in each hidden layer, 100 epochs, the Adam gradient descent optimizer, the BCELoss when using the classification model and the MSELoss when using the regression model. Note that when using the classification model, the labels were either 0 or 1, while when using the regression model, the labels could take any value between 0 and 1.

The reason for choosing 64 neurons (or nodes) as the size of the hidden layers is that it is a fairly complicated task but we have a small dataset, and adding too many neurons might result in overfitting. That is why we have experimentally selected 64. All the models have been trained with instances from all the datasets we worked in this thesis, J30, J60, J90, J120 and Pack. We have used 80% of the dataset to train, 10% to validate and 10% to test the model from a total of 198. These instances are the ones that did not have a trivial makespan (greater than the computed lower bound), were

solved in 3600 seconds or less by the custom made encoding, and resulted in a train file of less than 20Mb. The tests have been obtained using a variable K, selecting 10 random conflicts and with 5000 iterations for each measure.

	Classifier		Regressor	
K	**Precision@K**	**Recall@K**	**Precision@K**	**Recall@K**
1	0.8590	0.0859	0.8835	0.0884
3	0.8120	0.2436	0.8122	0.2437
5	0.8684	0.4342	0.8098	0.4049

TABLE 10.3: Test results for the GNN model

10.3 Solving results

To compute the solving results we are going to use the test instances, that have not participated in the training of our GNN model and constitute 10% of the available instances. This set includes randomly selected instances from the five datasets we have used during this thesis (J30, J60, J90, J120 and Pack). We are going to compare how many conflicts, propagations, decisions and restarts the solver goes through as well as the total time (T) and time spent in the solving process, without considering the time spent in the Python calls(Solving T(s)). The GNN model we used was the Regressor model, as we have experimentally proved it has better validation results 11.2.

	Decisions	**Conflicts**	**Propagations**	**Restarts**	**T(s)**	**S.T(s)**
No GNN	2559.47	1311.58	384235.42	6.42	0.22	0.22
GNN	2720.89	1351.89	1375354.63	6.37	1077.68	2.98

TABLE 10.4: Stats of the solver using the GNN and without doing so. The stats are the mean of all test instances. T(s) is the total solving time of the instance. S.T(s) is the solving time without counting the calls to the solver.

While the solver without using the GNN could solve all the test instances, the solver using the GNN option managed to solve 95% of the test instances.

Chapter 11

Conclusions

11.1 Encoding comparison

The results we have obtained are very interesting in that the two encodings developed perform very differently. We clearly see that the dummy model obtains consistently a better result in terms of instances solved, but as the instances get more complicated, the difference shrinks. We can also observe that although the number of instances solved in the most complicated dataset (j120 containing 120 activities per instance) is still better in the dummy encoding, the custom encoding manages to solve more instances in less than a second than the dummy encoding. That being said, even in this dataset the custom encoding still has many instances where it performs far better than the dummy encoding, resulting in a mean solving time for the solved instances greater than the one of the custom encoding. Figure 11.1 illustrates the performance in the J120 dataset (the most complicated dataset). In this figure we can see that in the easy instances (solved in less than a second) the custom-made encoding is always faster.

In conclusion, the evidence suggests that the dummy encoding is clearly better in solving time in smaller instances (few activities), but in bigger instances the custom made encoding is clearly better in easier instances and slightly worse in harder ones.

When we look at the size of the encoding, meaning the number of variables and clauses, the results turn the other way around. As we can see in Figure 11.2 and Figure 11.3, the number of variables and clauses increases with the number of activities, but we see that the custom encoding results in much more compact encodings, of approximately half the size of the dummy encoding. This will be very beneficial for us when training the GNN as using more compact encodings will result in a faster training time and the model being able to solve bigger instances before running out of memory.

The reasons behind this big disparity can be attributed to three factors. Firstly, the number of start and execution variables should be much bigger in the dummy encoding. Having split the activities will cause in many more of these variables. That is caused by the fact that logically, splitting all activities will cause each fragment of duration 1 to have the same time window as the original activity minus the duration of the activity minus one. On the other hand, the exactly K constraints of the custom made encoding also play a big factor but in increasing the sizes of the custom made encoding, as it creates a big number of clauses and variables that will not be present in the dummy encoding. This is caused by the fact that the Exactly-K

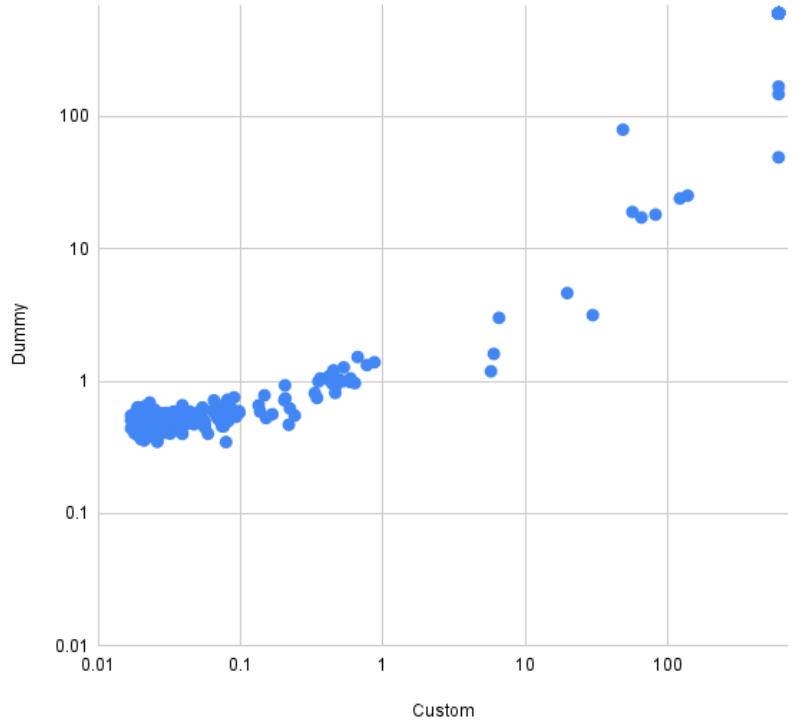


FIGURE 11.1: Scatter plot where each solved J120 instance is represented by its solving time in seconds in the custom encoding (horizontal axis) and dummy encoding (vertical axis).

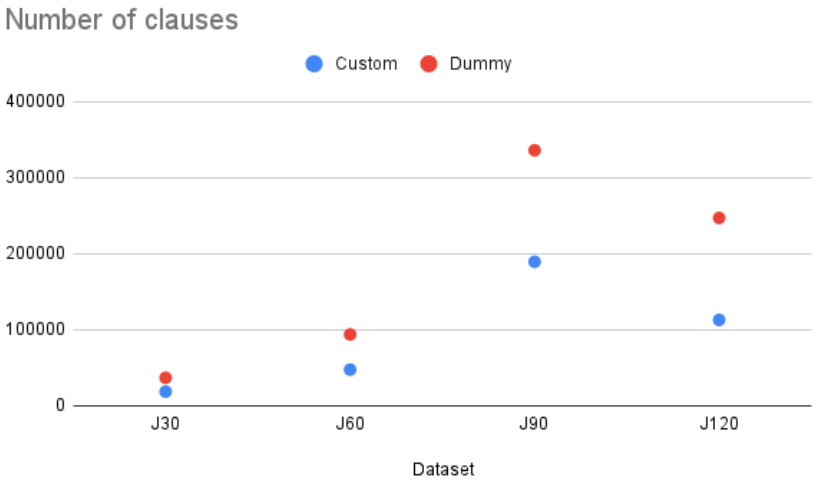


FIGURE 11.2: Comparison between the mean number of clauses generated by the dummy encoding and the custom made encoding in each dataset.

constraint is created using a large number of sorters and mergers, and each one of these introduces a substantial amount of clauses and auxiliary variables. But what really makes this big difference are the pseudo-boolean constraints. Although these constraints are added in both encodings, the dummy encoding having many more activities causes this pseudo-boolean encoding to grow substantially, resulting in the

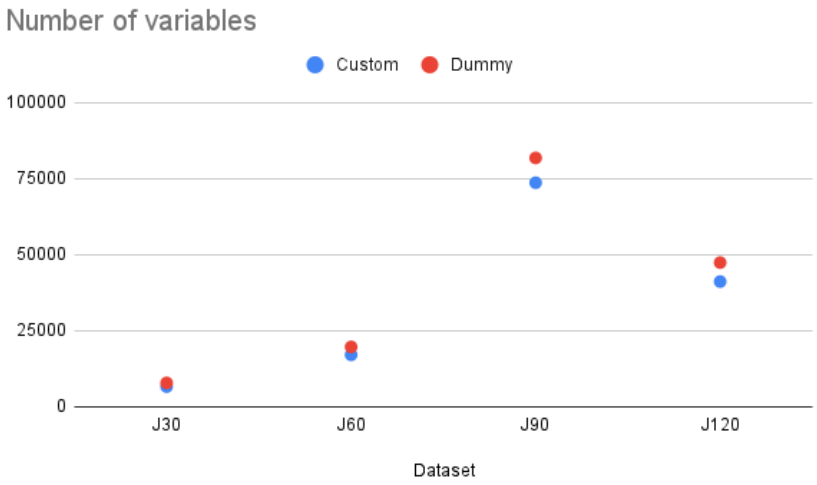


FIGURE 11.3: Comparison between the mean number of variables generated by the dummy encoding and the custom made encoding in each dataset.

differences we have seen experimentally.

11.2 Deep Learning model Validation

The validation of the GNN expresses how good our model is when selecting random conflicts using the metrics Precision@K and Recall@K. From Table 10.3, from which the Precision@K is summed up in Figure 11.4 we can extract several conclusions.

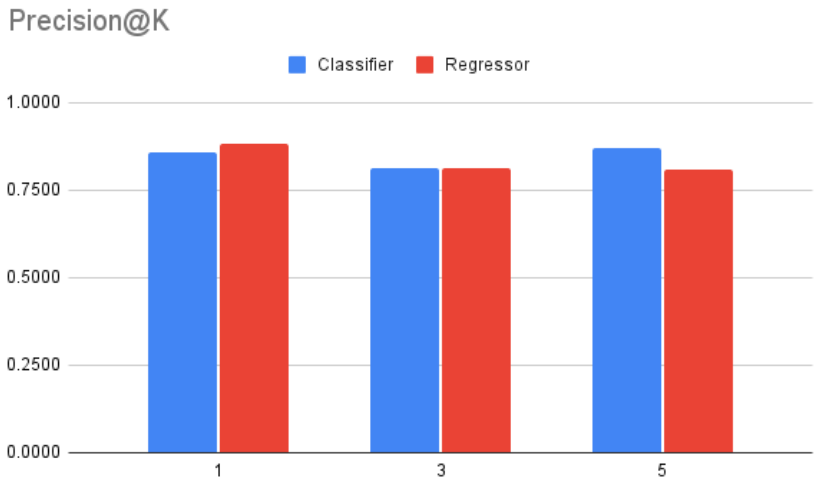


FIGURE 11.4: Bar chart representation of table 10.3. The horizontal axis contains the different K values and the vertical axis the Precision@K metric.

Firstly, the general Precision@K results found are considerably high, specially considering the amount of training data we have used, meaning that clauses being labeled relevant are consistently correctly being outputted as relevant.

Both the Classifier and the Regressor perform similarly, we can see that at K=5 the Classifier has considerably better results, but at K=1 the Regressor is the best. To select the better model among these two we have to take into account the type of problem we are trying to solve. That is why, as we are training a model where we will only use the best conflict found among the selected ones, we give more importance to the results of the K=1. If we check the formula of the Precision@K using K=1, we can see that for this value, what Precision@1 is describing is whether the best predicted conflict was the best one (1) or not (0).

The Recall@K is very low for K=1 and directly proportional to K. That is what is to be expected as we may have many relevant items, but the numerator of the Recall@K formula can only be as high as K. Considering we have selected 10 conflicts, this results for K=1 mean that there are many items considered relevant (predicted probability>0.7) that are not being selected. Because our model will be recommending only a single conflict, these results do not offer much insight into how well this model will perform in a real-world scenario.

Although in both models the output of the GNN model is a value in the range between 0 and 1 (which we round when using it as a classifier), the labels of the Classifier model are either 0 or 1. That makes the metric Precision@K pick among all the clauses that have a 1 as their label, and select the best K clauses randomly among the ones that participated in the regression tree. This clearly has an impact on the metrics of the Classifier.

Comparing both models for $K = 1$, we finally decide that we are going to use the Regressor to obtain the solving results, but both trained models are provided with the code and can be easily used to predict clauses from the SMTAPI.

11.3 Evaluation of the usage of the Deep Learning model in the SMTAPI

In Table 10.4 the solving resolution metrics for the execution of the test instances with and without querying the GNN model are compared. The first thing to notice is that not all instances could be solved using the GNN in our timeout (3600 seconds). This was predictable when looking at the actual resolution mean time of the instances. The time when not using the GNN model is 0.1% of the time using it. That being said, when looking at the solving time without the Python calls the resulting times are much more competitive. The difference we can still observe can be attributed to the fact that when we are using the GNN model we are searching all the conflicts instead of returning the first one we find. It is unrealistic to think that we can query the model at zero cost as making model calls will always come with a cost, but using other methods to do so will probably reduce the cost notably.

The plot in Figure 11.5 illustrates the relation between the number of decisions performed by the solver when using the GNN and when taking the first conflict we find. We can see that there are no outliers and the relation seems to be very stable (drawing almost a perfect line). That means that there are no instances where the usage of the GNN changes drastically the performance of the solver. That being said when looking at individual problems we see some instances whose solving metrics are better when using the GNN model. This might indicate that this technique still has room for improvement and can outperform a solver that does not use the deep

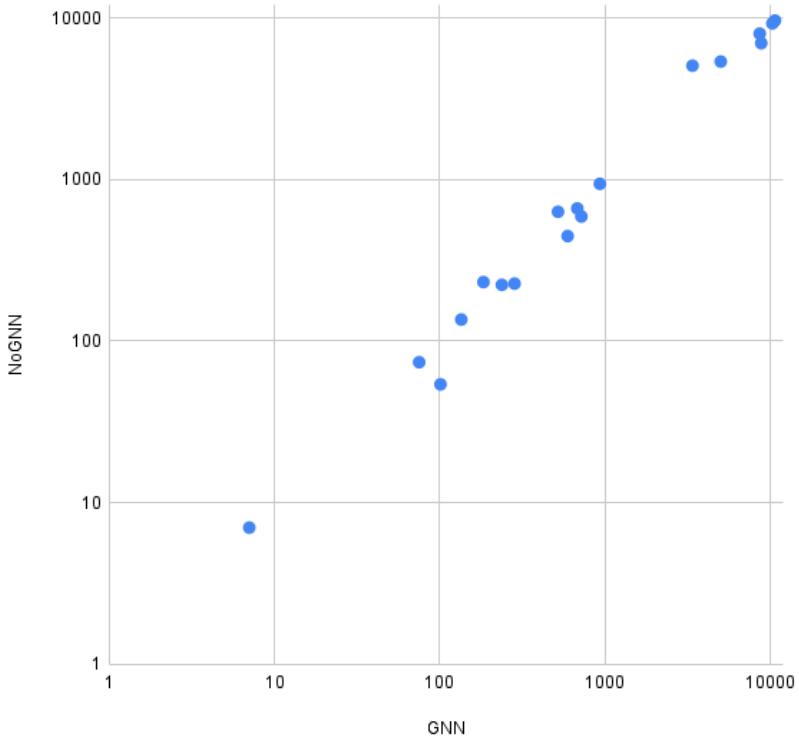


FIGURE 11.5: Scatter plot representing the number of decisions performed by the solver in each instance using the GNN and not doing so.

learning enhancements we built for bigger instances where the guidance of the GNN model has a bigger impact.

Looking at the metrics we can observe a pattern in the size of the instances we are solving. Due to the elevated costs of using the GNN, the only solved instances are the ones that can be quickly resolved (the mean time of solving instances is 0.22), meaning they were either trivial instances or the GNN model couldn't do much to improve its performance.

11.4 General conclusions, Requirement fulfillment and digressions

The software developed during this thesis offers a clear line of work in RCPSP improvements using deep learning thanks to its modularity, which enables future derived work to easily change its components for example using other solvers, other quality metrics or other neural network architectures. The structure we have created lays the groundwork to build more capable deep learning models that go a step further and outperform a classical SAT solver.

Having used the provided cluster to do experiments on deep learning and SAT solvers for the first time, it has flawlessly kept up with all the solver experiments.

We have been able to fulfill all requirements as expected. Our objectives have been

reached, and both the SMTAPI and the GNN can do everything the functional requirements laid out and the non-functional requirements have also been completely fulfilled. We have identified the key issues to work on, if we want to achieve an actual improvement in the solver's performance using GNN models.

On the other hand, the deep learning model could probably have gotten better results if it was trained with a greater number of instances. Although the set of instances used in this project contains a lot of information, it is a notably reduced set to train a GNN the size of ours. A team with more computation power, time and instances can probably achieve better results. That being said, even if we had better hardware, there would always be instances that we couldn't solve and hence use in the training set, because of the size of the encoding, if we want to load the full instance to predict all clauses. Ideally, the way to solve this would be to load only the graph neighbourhood of the conflicts we are comparing, and this way be able to solve greater instances, but as we previously explained, this would decrease the information we have from the instance and is not in the scope of this project.

Thanks to a very well-structured planning there were not many digressions from the original plan. The most important one was that firstly we had decided to create a regressor model and we finally decided to create both a regressor and a classifier with the same architecture but different hyperparameters to see how they would compare to each other.

Chapter 12

Future Work

There still remain different areas to be explored around the idea developed in this project.

Regarding the clause quality, it would be interesting to test other definitions for clause quality that differ from the one we used (based on the number of appearances in the DRAT-trim checking process), to see how they perform in the prediction of the best clause to propagate from. It would also be interesting to see how the results would change when adding clause features that give insights into the solving process of the SAT solver, like the decision level where a clause was learnt. This type of features may be useful when selecting the clause and they have not been explored in this thesis due to the performance backlash that would result from having to write a new DIMACS file every time the model is queried and read all over again.

Another good comparison might be to try other neural network architectures to see how they perform whether they are classical neural networks (without graph representation) or other more advanced architectures that exist or may come out during the following years. In addition, finding a model with more explainability would give us an insight into the relation between CDCL solving and RCPSP encodings. This knowledge would then contribute to the development of better encodings. Still in the subject of the GNN training, due to our limited access to resources and time we have trained our model with a reduced number of instances, but we could check how would the model perform if we increased the number of instances the model is trained from. The most efficient way to increase the number of instances would be to develop an instance generator, but one should be very careful in the implementation of a generator of instances for a deep learning model because the model might end up being biased toward the types of instances the generator has produced and perform very poorly when using real-world data, thus becoming overfitted.

In the subject of PRCPSP SAT encodings, one could also develop new approaches, whether they are alternative viewpoints or other encodings for the cardinality and pseudo-boolean constraints. Then, they could be compared to the presented PRCPSP encodings and finally check the model performance using this new encoding. For example, classical RCPSP encodings could be tested, but also other types of scheduling problems, such as MRCPSP. It would also be interesting to see if this approach could be extrapolated into other fields apart from scheduling, where information regarding the original problem is passed to a neural network architecture to guide the SAT solver in its resolution.

The time it takes to load the full instance every time the model is checked is also an important factor in the performance of the model, so it would be worth to explore other options that keep the instance loaded. One option would be to do it directly from C++, which, as explained, we discarded because differently from the library PyTorch, the library PyG does not have a C++ version. If in the future there is an alternative, in the form of another library or a direct encoding in C++, it could significantly improve the solving time.

Another option to have data persistence among the different solver queries would be to create a Python server that loads the new instance on the first call and keeps it loaded in memory during the subsequent calls. Here, the time between calls would be a determining factor for the performance of the model, as well as how are the learnt clauses treated (if they are included or not and how is this done). That being said, this option would give no guarantee of improvement as making a call to the server every time we want to solve an instance would also be detrimental to the performance of the solver.

Alphabetical Index

Bibliography

[1] Daniel Selsam et al. "Learning a SAT Solver from Single-Bit Supervision". In: *CoRR* abs/1802.03685 (2018). arXiv: 1802.03685. URL: http://arxiv.org/abs/1802.03685.

[2] Marijn Heule, Markus Iser Matti Järvisalo, and Martin Suda. *The International SAT Competition Web Page*. 2016. URL: https://satcompetition.github.io/.

[3] Atlassian. *Trello*. 2023. URL: https://trello.com/en.

[4] Glassdoor. *Sueldos para el puesto de AI Engineer en España*. 2024. URL: https://www.glassdoor.es/Salaries/spain-ai-engineer-salary-SRCH_IL.0,5_IN219_KO6,17.htm?countryRedirect=true.

[5] Google. *Google Kubernetes Engine pricing*. 2024. URL: https://cloud.google.com/kubernetes-engine/pricing.

[6] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: https://agilemanifesto.org/.

[7] Ken Schwaber and Jeff Sutherland. *The 2020 Scrum Guide*. 2020. URL: https://scrumguides.org/scrum-guide.html.

[8] Jacek Blazewicz et al. *Handbook on Scheduling*. Springer Cham, 2019. URL: https://doi.org/10.1007/978-3-319-99849-7.

[9] Miquel Bofill et al. "SMT encodings for Resource-Constrained Project Scheduling Problems". In: *Comput. Ind. Eng.* 149 (2020), p. 106777. DOI: 10.1016/J.CIE.2020.106777. URL: https://doi.org/10.1016/j.cie.2020.106777.

[10] Miquel Bofill et al. "An MDD-based SAT encoding for pseudo-Boolean constraints with at-most-one relations". In: *Artif. Intell. Rev.* 53.7 (2020), pp. 5157–5188. DOI: 10.1007/S10462-020-09817-6. URL: https://doi.org/10.1007/s10462-020-09817-6.

[11] José Coelho and Mario Vanhoucke. "Multi-mode resource-constrained project scheduling using RCPSP and SAT solvers". In: *European Journal of Operational Research* 213.1 (2011), pp. 73–82. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2011.03.019. URL: https://www.sciencedirect.com/science/article/pii/S037722171100230X.

[12] Lintao Zhang and Sharad Malik. "The Quest for Efficient Boolean Satisfiability Solvers". In: *Computer Aided Verification*. Ed. by Ed Brinksma and Kim Guldstrand Larsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 17–36. ISBN: 978-3-540-45657-5.

[13] J.P. Marques Silva and K.A. Sakallah. "GRASP-A new search algorithm for satisfiability". In: *Proceedings of International Conference on Computer Aided Design*. 1996, pp. 220–227. DOI: 10.1109/ICCAD.1996.569607.

[14] J.P. Marques-Silva and K.A. Sakallah. "GRASP: a search algorithm for propositional satisfiability". In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521. DOI: 10.1109/12.769433.

[15] Roberto J. Bayardo and Robert C. Schrag. "Using CSP Look-Back Techniques to Solve Real-World SAT Instances". In: *AAAI/IAAI*. 1997. URL: https://api.semanticscholar.org/CorpusID:1302756.

[16] Clark W. Barrett et al. "Handbook of Satisfiability". In: *Handbook of Satisfiability*. 2021. URL: https://api.semanticscholar.org/CorpusID:12067495.

[17] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[18] Irina Rish and Rina Dechter. "Resolution versus Search: Two Strategies for SAT". In: *Journal of Automated Reasoning* 24 (Dec. 2000). DOI: 10.1023/A:1006303512524.

[19] Marijn Heule. *Drat-Trim repository in Github*. 2023. URL: https://github.com/marijnheule/drat-trim.

[20] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. "DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs". In: *International Conference on Theory and Applications of Satisfiability Testing*. 2014. URL: https://api.semanticscholar.org/CorpusID:2064344.

[21] S. Weidman. *Deep Learning from Scratch: Building with Python from First Principles*. O'Reilly Media, 2019. ISBN: 9781492041382. URL: https://books.google.es/books?id=MrWuDwAAQBAJ.

[22] Zonghan Wu et al. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (2021), pp. 4–24. DOI: 10.1109/TNNLS.2020.2978386.

[23] J. et al. Jumper. "Highly accurate protein structure prediction with AlphaFold." In: *Nature* 596 (2021), pp. 583–589. DOI: 10.1038/s41586-021-03819-2. URL: https://www.nature.com/articles/s41586-021-03819-2.

[24] Si Zhang et al. "Graph convolutional networks: a comprehensive review". In: *Computational Social Networks* 6 (2019). URL: https://api.semanticscholar.org/CorpusID:207960027.

[25] Petar Veličković et al. *Graph Attention Networks*. 2018. arXiv: 1710.10903 [stat.ML].

[26] Medium Omar Hussein. *Graph Neural Networks Series | Part 4 |The GNNs, Message Passing and Over-smoothing*. 2023. URL: https://medium.com/the-modern-scientist/graph-neural-networks-series-part-4-the-gnns-message-passing-over-smoothing-e77ffee523cc.

[27] Towards Data Science Kizito Nyuytiymbiy. *Parameters and Hyperparameters in Machine Learning and Deep Learning*. 2023. URL: https://towardsdatascience.com/parameters-and-hyperparameters-aa609601a9ac.

[28] Medium (Towards Data Science) Daniel Godoy. *Understanding binary cross-entropy / log loss: a visual explanation*. 2018. URL: https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a.

[29] Rainer Kolisch and Arno Sprecher. "PSPLIB - A project scheduling problem library: OR Software - ORSEP Operations Research Software Exchange Program". In: *European Journal of Operational Research* 96.1 (1997), pp. 205–216. ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(96)00170-1. URL: https://www.sciencedirect.com/science/article/pii/S0377221796001701.

[30] Jordi Coll. "Scheduling through logic-based tools". PhD thesis. University of Girona, Spain, 2019. URL: http://hdl.handle.net/10803/667963.

[31] Code Project Matt Scarpino. *Interfacing C++ and Python with the Python API*. 2023. URL: https://www.codeproject.com/Articles/5365450/Interfacing-Cplusplus-and-Python-with-the-Python-A.

[32] The Linux Foundation. *PyTorch C++ API*. 2024. URL: https://pytorch.org/cppdocs.

[33] SLURM. *SLURM Workload manager: Documentation*. 2024. URL: /https://slurm.schedmd.com/.

[34] PyTorch Contributors. *PyTorch documentation*. 2023. URL: https://pytorch.org/docs/stable/index.html.

[35] PyG Team. *PyG (PyTorch Geometric) Documentation*. 2024. URL: https://pytorch-geometric.readthedocs.io/en/latest/.

[36] NumPy Developers. *NumPy documentation, Version: 1.26*. 2022. URL: https://numpy.org/doc/1.26/.

[37] Python Software Foundation. *pickle — Python object serialization*. 2024. URL: https://docs.python.org/3/library/pickle.html.

[38] The Matplotlib development team. *Matplotlib 3.8.4 documentation*. 2024. URL: https://matplotlib.org/stable/index.html.

[39] Ian P Gent and Toby Walsh. "The SAT phase transition". In: *ECAI*. Vol. 94. PITMAN. 1994, pp. 105–109.

[40] Matthew W Moskewicz et al. "Chaff: Engineering an efficient SAT solver". In: *Proceedings of the 38th annual Design Automation Conference*. 2001, pp. 530–535.

[41] William L. Hamilton, Rex Ying, and Jure Leskovec. *Inductive Representation Learning on Large Graphs*. 2018. arXiv: 1706.02216 [cs.SI]. URL: https://arxiv.org/abs/1706.02216.

[42] Christopher Morris et al. *Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks*. 2021. arXiv: 1810.02244 [cs.LG].

[43] Medium Sahil Sheikh. *Exploring SageConv: A Powerful Graph Neural Network Architecture*. 2023. URL: https://medium.com/@sheikh.sahil12299/exploring-sageconv-a-powerful-graph-neural-network-architecture-44b7974b1fe0.

[44] Recall and Precision at k for Recommender Systems. *Maher Malaeb — Medium*. 2017. URL: https://medium.com/@m_n_malaeb/recall-and-precision-at-k-for-recommender-systems-618483226c54.

[45] Roberto Asín et al. "Cardinality Networks: a theoretical and empirical study". In: *Constraints An Int. J.* 16.2 (2011), pp. 195–221. DOI: 10.1007/S10601-010-9105-0. URL: https://doi.org/10.1007/s10601-010-9105-0.

[46] Miquel Bofill et al. "SAT encodings for Pseudo-Boolean constraints together with at-most-one constraints". In: *Artif. Intell.* 302 (2022), p. 103604. DOI: 10.1016/J.ARTINT.2021.103604. URL: https://doi.org/10.1016/j.artint.2021.103604.

[47] Maxime Labonne. *What is a Tensor in Machine Learning? - Towards Data Science*. 2022. URL: https://towardsdatascience.com/what-is-a-tensor-in-deep-learning-6dedd95d6507.

[48] PyG Team. *torch_geometric.loader, PyG Documentation*. 2024. URL: https://pytorch-geometric.readthedocs.io/en/latest/modules/loader.html.

[49] Jason Brownlee. *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*. 2021. URL: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/.

[50] Jacques Carlier and Emmanuel Néron. "On linear lower bounds for the resource constrained project scheduling problem". In: *European Journal of Operational Research* 149.2 (2003), pp. 314–324.

Appendix A

Installation and User Manual

A.1 Setting up the SMTAPI

This section describes the steps used to set up the environment of the SMTAPI to solve RCP instances using the developed encodings and the GNN in Ubuntu distributions.

1. Make sure you have the correct version of C++ installed. The version used in the project is `gcc 11.4.10`. You check your current version by using:

```
gcc --version
```

2. Having already installed Python, install PyTorch and PyTorch Geometric if you did not already have them. You should also install its dependencies. You can do so by using the following commands:

```
pip3 install torch
pip3 install torch_geometric
# Dependencies
pip3 install torch_scatter torch_sparse
```

3. Download the libraries from GitHub. You can do it using the following command:

```
git clone https://github.com/Bernat-C/ClauseRec
```

4. Update the `PYTHONPATH` to include the folder "ClauseRecommender/src". You can do it using the following command. Make sure to change "user" to the name of your user.

```
export PYTHONPATH='/home/user/ClauseRec/ClauseRecommender/src'
```

5. Update the `CPATH` to include the folder where the desired Python installation you want to use is located. If you are using the Python version specified in

Chapter 5 and you used the default instalation in linux it should be in the folder "/usr/include/python3.10". The command to update the `CPATH` is the following:

```
export CPATH=":/usr/include/python3.10"
```

Note: If you are using a different Python version you should also update the `Makefile` of the `solversjordi` folder, specifically the line `LFLAGS += -lpython3.10` to match the Python library version you want to use.

A.2 Using the SMTAPI

Once the installation process is complete, we have to make the executables. This is done by using the following commands, depending on which parts of the SMTAPI you want to use. To use this commands you must be inside the directory solversjordi, as otherwise they will not work. They have to be used just once.

To compile the custom made encoding and the deep learning model, use:

```
make DEBUG=0 -j 16 prcpsp
```

And then, to solve an instance you can use the following command. Make sure to substitute `instancepath.RCP` for the name of your RCP instance. The explanation of the different options is found using the help command. Here, you can use the option `use-gnn=1` to make use of the GNN model and `u=number` to set the initial upper bound of the current instance to a specific number.

```
./bin/release/prcpsp2dimacs instancepath.RCP -s=maple --use-assumptions=0
    --print-nonoptimal=0 -o=ub
```

If you want to display the different solving options for the PRCPSP instance, you can use the following command after having compiled the file.

```
./bin/release/prcpsp2dimacs --help
```

To compile the dummy encoding, use:

```
make DEBUG=0 -j 16 mrcpsp2smt
```

To test the solving of an instance you can use the following command. Make sure to change `instancepath.RCP` for the name of your RCP instance. The parameters `prcpsp-dummy=1` and `encoding=satorder` are what activates the dummy PRCPSP solving.

```
./bin/release/mrcpsp2smt instancepath.RCP -s=maple --use-assumptions=0 --
    prcpsp-dummy=1 --encoding=satorder --print-nonoptimal=0 -o=ub
```

Similarly to the PRCPSP command, if you want to display the different solving options for the PRCPSP instance, you can use the following command after having compiled the file.

```
./bin/release/mrcpsp2smt --help
```

To compile the checker, use:

```
make DEBUG=0 -j 16 checkprcpsp
```

To check the solution of an RCP instance (it should have been firstly saved to a file) you can use the following command, where `instancepath.RCP` is the path of the original RCP instance and `solution.out` is the name of the file that contains the solution.

```
./bin/release/checkprcpsp -V=1 instancepath.RCP solution.out
```