UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

Bruno Machado Pacheco

Deep-learning-based Primal Heuristics for MILP: Supervised Solution-prediction Models

Florianópolis
2024

Bruno Machado Pacheco

Deep-learning-based Primal Heuristics for MILP: Supervised Solution-prediction Models

Dissertação submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina para a obtenção do título de Mestre em Engenharia de Automação e Sistemas

Orientador: Prof. Eduardo Camponogara, Ph.D.
Co-orientador: Prof. Laio Oriel Seman, Ph.D.

Florianópolis
2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Pacheco, Bruno Machado
Deep-learning-based Primal Heuristics for MILP: Supervised Solution-prediction Models / Bruno Machado Pacheco ; orientador, Eduardo Camponogara, coorientador, Laio Oriel Seman, 2024.
86 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Engenharia de Automação e Sistemas, Florianópolis, 2024.

Inclui referências.

1. Engenharia de Automação e Sistemas. 2. Integer programming. 3. Deep learning. 4. Graph Neural Networks. 5. Matheuristics. I. Camponogara, Eduardo. II. Seman, Laio Oriel. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Automação e Sistemas. IV. Título.

Bruno Machado Pacheco

Deep-learning-based Primal Heuristics for MILP: Supervised Solution-prediction Models

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Danilo Silva, Dr.
Universidade Federal de Santa Catarina

Prof. Teobaldo Leite Bulhões Júnior, Dr.
Universidade Federal da Paraíba

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestre em Engenharia de Automação e Sistemas.

Documento assinado digitalmente
Julio Elias Normey Rico
Data: 16/08/2024 15:08:57-0300
CPF: ***.840.859-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Julio Elias Normey Rico, Dr.
Coordenador do Programa

Documento assinado digitalmente
Eduardo Camponogara
Data: 16/08/2024 15:50:16-0300
CPF: ***.988.180-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Eduardo Camponogara, Dr.
Orientador

Florianópolis, 13 de Agosto de 2024.

AGRADECIMENTOS

Esta dissertação, enquanto requisito conclusivo do mestrado em engenharia de automação e sistemas, representa dois anos de dedicação ao estudo e à pesquisa. Tal qual em meu trabalho de conclusão do curso de engenharia de controle e automação, mas com ainda mais veemência, quaisquer louros que eu possa ter colhido são justamente devidos aos mestres, àqueles que me ensinaram ao longo desta jornada na academia. Aqui, agradeço especialmente ao professor Eduardo Camponogara, meu orientador, que além de um excelente representante desse grupo, também me apoiou e forneceu suporte em múltiplos âmbitos, assim tornando fértil o que eu considero ter sido um período de muito amadurecimento.

Não seria justo deixar de agradecer também a todos os meus colegas do GOS (grupo de pesquisa em otimização de sistemas). Exemplarmente, agradeço ao professor Laio Oriel Seman, meu co-orientador, pelas tantas ideias e desafios propostos, como também pelo pioneirismo em nosso grupo no que se refere à linha de pesquisa na qual minha dissertação se situa.

Finalmente, agradeço também aos meus, que justificam e motivam tanto a minha dedicação, quanto o meu descanso. Em particular, sou grato àqueles que estiveram mais próximos - meus primos, meus sogros e, especialmente, minha companheira -, a quem eu credito a instauração do meu sentimento de pertencimento nesta cidade.

ABSTRACT

Mixed-Integer Linear Programming (MILP) is a crucial tool for solving complex decision-making problems due to its ability to model combinatorial optimization tasks and arbitrarily approximate nonlinear features. Deep-learning-based primal heuristics offer a promising solution for efficiently addressing MILP problems. Focusing on supervised solution prediction models, this dissertation investigates the design, training, and integration of deep learning models into primal heuristics using the Offline Nanosatellite Task Scheduling (ONTS) problem as a test case. Key findings are drawn on model architecture, loss functions, data acquisition, and meta-heuristic. On top of that, the proposed learning-based heuristic approaches were able to provide, on one hand, a 35% reduction in the expected time to find a feasible solution to the ONTS problem, and on another, a 43% expected gain in the normalized quality of the heuristic solutions. These results highlight the potential of deep learning approaches to enhance the adaptability and efficiency of optimization solutions, with future research needed to further explore Graph Neural Network (GNN) generalization and improve data generation techniques.

Keywords: MILP, Matheuristics, Deep Learning, Learning-based Heuristics, Graph Neural Networks, Nanosatellite Task Scheduling.

RESUMO

A programação linear inteira mista (*Mixed-Integer Linear Programming*, MILP) é crucial no auxílio à tomada de decisão em cenários complexos devido à sua capacidade de modelar problemas de otimização combinatória e aproximar dinâmicas não-lineares. Heurísticas baseadas em modelos de aprendizagem profunda (*deep learning*) oferecem uma solução promissora para resolver problemas MILP eficientemente. Tendo foco em modelos supervisionados para predição de soluções, esta dissertação investiga o projeto, o treinamento e a integração de modelos de aprendizagem profunda em heurísticas primais, usando o agendamento *offline* de tarefas em nanossatélites (*Offline Nanosatellite Task Scheduling*, ONTS) como um caso de teste. As principais conclusões deste trabalho se referem à arquitetura dos modelos, às funções de perda, à aquisição de dados e à meta-heurísticas. Além disso, as heurísticas baseadas em aprendizagem propostas para o ONTS foram capazes de reduzir, em média, 35% do tempo necessário para encontrar uma solução factível, e um ganho médio de 43% na qualidade das soluções encontradas. Esses resultados destacam o potencial da aprendizagem profunda em gerar heurísticas adaptáveis e eficientes para problemas de otimização, direcionando pesquisas futuras para a investigação da capacidade de generalização de redes neurais baseadas em grafos e de técnicas para geração de dados sintéticos.

Palavras-chaves: MILP, Matheuristics, Deep Learning, Learning-based Heuristics, Graph Neural Networks, Nanosatellite Task Scheduling.

RESUMO EXPANDIDO

Introdução

Esta dissertação explora a aplicação de heurísticas primais baseadas em modelos de aprendizagem profunda à programação linear inteira mista (*Mixed-Integer Linear Programming*, MILP). MILP é uma ferramenta chave da pesquisa operacional devido a sua capacidade de modelar problemas combinatórios e de aproximar, com precisão arbitrária, dinâmicas não-lineares. Além disso, existência de *softwares* bem-estabelecidos para resolver problemas de MILP facilita a sua aplicação fácil e a torna confiável.

Encontrar soluções ótimas para problemas de MILP de forma eficiente é um desafio devido ao crescimento exponencial do espaço de busca em função do número de variáveis inteiras. Como consequência, heurísticas primais se tornam valiosas como uma forma tratável de encontrar soluções de boa qualidade em contextos de recursos limitados. Entretanto, projetar uma heurística primal efetiva é uma tarefa que requer um grande esforço de engenharia pois deve ser feita sob medida para o problema alvo. Recentemente, técnicas de aprendizagem profunda foram propostas para criar heurísticas especializadas de forma automática, explorando padrões existentes nos dados do problema alvo.

Objetivos

O principal objetivo desta dissertação é estudar e avaliar heurísticas primais para problemas de MILP baseadas em modelos de predição de solução treinados com supervisão. Este objetivo é subdividido em três:

- Analisar a literatura de aprendizado supervisionado para modelos de predição de solução de problemas de MILP, incluindo arquiteturas, algoritmos de aprendizagem e heurísticas primais baseadas em aprendizagem;
- Implementar heurísticas primais baseadas em aprendizagem para uma aplicação realista, incluindo as técnicas mais promissoras encontradas na literatura; e
- Avaliar as técnicas para heurísticas baseadas em aprendizagem com respeito a performance empírica na aplicação selecionada e as garantias teóricas fornecidas por cada uma delas.

Metodologia

Este trabalho estudou e avaliou diversas técnicas encontradas na literatura para os mais distintos componentes de heurísticas primais baseadas em modelos de solução de predição para problemas de MILP. Em relação à arquitetura dos modelos de predição de solução, foram investigados o uso de redes neurais baseadas em grafos (*Graph Neural Networks*, GNNs) com convoluções baseadas no operador SAGE e o compartilhamento de parâmetros da rede entre as suas convoluções. Duas técnicas distintas de treinamento foram implementadas e avaliadas: a primeira utilizando uma solução (quasi-)ótima, e a segunda utilizando múltiplas soluções para cada instância disponível do problema de otimização. Técnicas para aquisição de dados também foram analisadas, em particular no contexto da ausência de dados históricos. Três diferentes arquiteturas baseadas em modelos de solução de predição para a construção de heurísticas primais foram avaliadas a partir de dois objetivos distintos, mas complementares: encontrar soluções factíveis no menor tempo possível, e encontrar a melhor solução em um tempo limitado.

Os experimentos foram projetados para avaliar a efetividade das heurísticas propostas em um problem realista: o agendamento *offline* de tarefas em nanossatélites (*Offline Nanosatellite Task Scheduling*, ONTS). O cenário analisado do problema em questão é o agendamento durante a operação do satélite em órbita, que requer a solução de múltiplas instâncias do problema de MILP em uma janela de tempo limitada.

Resultados e Discussão

Os experimentos no problema de ONTS indicaram as melhores configurações para modelos de predição de solução. Em particular, eles apontam uma superioridade do operador SAGE (HAMILTON; YING; LESKOVEC, 2017) em relação à convolução original, proposta por Kipf and Welling (2017), além de um ganho de desempenho ao compartilhar os parâmetros entre as convoluções. Os melhores modelos de predição de solução foram treinados utilizando múltiplas soluções por instância do problema como supervisão.

A construção de heurísticas primais com os modelos treinados se mostrou mais efetiva quando se dava através da fixação de variáveis binárias através da predição dos modelos (*early-fixing*). Essa estratégia resultou em heurísticas que reduziram, em média, 35% do tempo necessário para encontrar uma solução factível, e aumentaram em 43% a qualidade da solução encontrada dado um tempo limitado de 2 minutos.

A aquisição de dados se mostrou um desafio devido a ausência de dados históricos e ao alto custo para encontrar soluções para as instâncias sintéticas do problema. De toda forma, a capacidade de generalização dos modelos de predição de solução construídos com GNNs permitiu o treinamento com instâncias mais fáceis (e, portanto, menos custosas) do que aquelas utilizadas para avaliação.

Considerações Finais

Esta dissertação demonstra que heurísticas primais baseadas em aprendizagem profunda são promissoras frente aos desafios da MILP. Os resultados contribuem para a área de pesquisa de aprendizado de máquina para otimização combinatória ao oferecer uma análise das técnicas mais relevantes encontradas na literatura e uma comparação empírica e não-enviesada em uma aplicação representativa. Além disso, esta dissertação aponta para uma investigação mais aprofundada sobre os limites da capacidade de generalização das GNNs em problemas de MILP, técnicas de geração de dados sintéticos para modelos de predição de solução, e um refinamento da eficiência desses mesmos modelos em relação aos dados necessário e o desempenho esperado.

Palavras-chaves: MILP, Matheuristics, Deep Learning, Learning-based Heuristics, Graph Neural Networks, Nanosatellite Task Scheduling.

CONTENTS

Introduction 12
Objectives . . . 13

I BACKGROUND 15

1 INTEGER PROGRAMMING . . . 16
1.1 Integer and Combinatorial Optimization . . . 16
1.2 Mixed-Integer Linear Programs . . . 17
1.3 Solving MILP Problems . . . 17
1.3.1 The Branch-and-Bound Algorithm . . . 18
1.3.2 Heuristics . . . 21
1.3.2.1 Matheuristics . . . 22

2 DEEP LEARNING . . . 23
2.1 Supervised Learning . . . 23
2.1.1 Supervised learning algorithm . . . 23
2.1.2 Generalization and overfitting . . . 24
2.1.3 Hyperparameter tuning . . . 26
2.2 Deep Neural Networks . . . 27
2.2.1 Gradient-based learning . . . 27
2.2.2 Graph Neural Networks . . . 29

II MATERIALS AND METHODS 32

3 SOLUTION PREDICTION MODELS FOR MILP PROBLEMS . . . 33
3.1 Embedding Optimization Problems . . . 33
3.1.1 Feature Engineering . . . 34
3.1.2 Graph Embedding . . . 35
3.2 Training Under Supervision . . . 36
3.2.1 Multiple Targets . . . 37
3.3 Learning-based Heuristics . . . 37
3.3.1 Warm-starting MILP Solvers . . . 38
3.3.2 Early-fixing Variable Assignments . . . 39
3.3.3 Trust-region . . . 39

4 OFFLINE NANOSATELLITE TASK SCHEDULING 41
4.1 Problem Statement 41
4.2 MILP Formulation 43

5 EVALUATION OF PRIMAL HEURISTICS 46

III EXPERIMENTS AND RESULTS 48

6 EXPERIMENTS 49
6.1 Data 49
6.1.1 Instance space: the FloripaSat I mission 49
6.1.2 Data acquisition 50
6.2 Solution Prediction Model 52
6.2.1 Instance embedding 52
6.2.2 Architecture 53
6.3 Training 55
6.3.1 Hyperparameter Tuning 55
6.3.2 Final solution prediction models 57
6.4 Learning-based heuristics 58
6.4.1 Tuning 58
6.4.2 Evaluation 59

7 DISCUSSION 62
7.1 Solution prediction models 62
7.2 Matheuristics 63
7.3 Data acquisition and generalization 63

Conclusion 65

BIBLIOGRAPHY 67

INTRODUCTION

Integer programming stands as a cornerstone in addressing complex decision-making problems, offering a powerful framework for modeling combinatorial optimization problems (WOLSEY, 2020). In particular, Mixed Integer Linear Programming (MILP) significance stems from its ability to represent combinatorial optimization tasks with linear objectives and constraints. Although apparently limiting, the linear requirements allow it to model many problems (NEMHAUSER; WOLSEY, 1988) and arbitrarily approximate most problems, e.g., through piecewise-linear approximations of the nonlinearities (CAMPONOGARA; NAZARI, 2015). On top of that, due to the linearity of the constraints and objective functions, we have reliable algorithms to solve MILP problems. In fact, due to the modeling capacity and the robustness of the software solutions, MILP has become the workhorse of combinatorial optimization (BENGIO; LODI; PROUVOST, 2021).

Nonetheless, solving instances of MILP problems efficiently remains a formidable challenge, motivating the development of heuristic solutions. The combinatorial nature of MILP implies that algorithms with optimality guarantees have intractable running times, as the search space expands exponentially with the number of integer variables. Primal heuristics, which aim to quickly finding high-quality feasible solutions to MILP problems, play a crucial role in enhancing the efficiency of optimization algorithms. Traditional primal heuristics are often rule-based and designed to exploit structures of a given MILP problem. As a consequence, they lack adaptability, struggling to generalize across diverse problem instances. As the landscape of optimization problems continues to evolve, there is a growing need for intelligent and flexible heuristics that can adapt to the intricacies of different MILP instances.

Recently, deep learning techniques have been successfully applied to MILP, resulting in effective heuristics (NAIR et al., 2021; GASSE et al., 2022; LARSEN et al., 2022; KHALIL; MORRIS; LODI, 2022; HAN et al., 2023). In contrast to handcrafted heuristics, which rely on expert knowledge to exploit theoretical structures of problem formulations, deep learning-based heuristics identify the hidden patterns of problem instances from data. This data-driven approach relies on the assumption that problem instances are drawn from underlying distributions and, thus, share characteristics not evident in the mathematical formulation. Such an assumption often holds for practical situations in which problems must be solved repeatedly, and the parameters that define the instances are random variables with unknown distributions (BENGIO; LODI; PROUVOST, 2021).

The research area of deep learning applications to MILP has seen a burst of publications in the past years, as seen in Fig. 1, with plenty of novel methods being proposed. The comparisons, however, are still limited, which hinders the effective application of the approaches available in

the literature. In this context, this master's dissertation aims to contribute to developing learning-based heuristics for MILP problems by evaluating existing techniques in novel applications. More specifically, this work focuses on deep learning models trained with supervision to predict solutions to MILP problems.

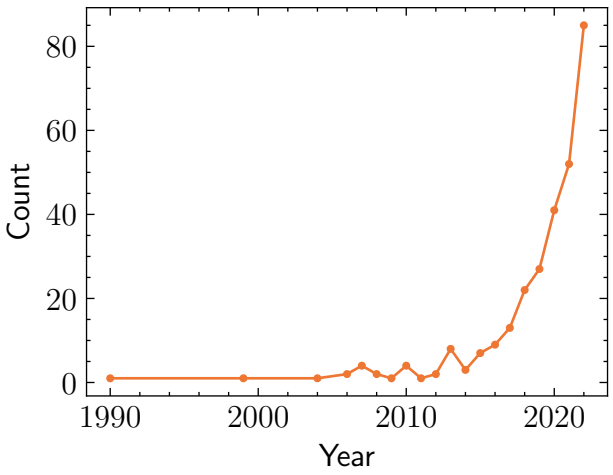


Figure 1 – Number of scientific publications (journal and conference papers) in English language containing the terms "learning" and "MILP" in their title, abstract or keywords. Source: SCOPUS

Objectives

In the topic of this dissertation, three research questions are of fundamental importance in the development of heuristic approaches:

Q1 How to design deep learning models to provide candidate solutions for instances of an MILP problem?

Q2 Which supervised learning techniques are most effective for training solution prediction models for primal heuristics? And

Q3 How to incorporate solution prediction models in primal heuristics?

Targeting these questions, this dissertation aims *to study and evaluate primal heuristics for MILP problems based on solution prediction models trained with supervised learning techniques*. This goal is broken down into three objectives:

- Analyze the literature on supervised learning solution prediction models for MILP problems, including model architectures, supervised learning algorithms, and learning-based primal heuristics;
- Apply learning-based primal heuristics for a realistic application based on the most promising techniques found in the literature; and

- Evaluate the techniques for learning-based heuristics with respect to the empirical performance in a realistic application and the theoretical guarantees provided by each.

The achievement of these objectives will result in the two major contributions of this work to the research community.

Part I

Background

1 INTEGER PROGRAMMING

Integer Programming (IP), a subset of mathematical programming, addresses optimization problems where decision variables are required to take on integer values. Specifically, mixed-integer linear programming (MILP) extends this concept by encompassing the assumption that for each possible discrete decision, a (continuous) linear program has to be solved. The complexity of MILP problems often necessitates sophisticated solution methods to find optimal or near-optimal solutions. This chapter provides an overview of MILP problem-solving techniques, ranging from exact methods like the branch-and-bound algorithm to approaches to provide approximate solutions, such as heuristics and matheuristics. Through these methodologies, the groundwork is laid for the subsequent discussion on deep learning-based primal heuristics, which aim to enhance the efficiency of MILP problem solving.

1.1 INTEGER AND COMBINATORIAL OPTIMIZATION

A solution for an integer and combinatorial optimization problem is the maximum or minimum value of a multivariate function that respects a series of inequality and equality constraints and integrality restrictions on some or all variables (NEMHAUSER; WOLSEY, 1999). It is not difficult to see that integer and combinatorial optimization encompasses a wide range of problems of practical utility. Examples include train scheduling, airline crew scheduling, production planning, electricity generation planning, and cutting problems (WOLSEY, 1998).

Mathematical programming is a language naturally suitable to formulate integer and combinatorial optimization problems, for example, in the form

$$\begin{aligned} \min_{\boldsymbol{y}} \quad & f(\boldsymbol{y}) \\ \text{s.t.} \quad & \boldsymbol{g}(\boldsymbol{y}) \leq \mathbf{0} \\ & \boldsymbol{y} \in \mathbb{Z}^n \times \mathbb{R}^p, \end{aligned} \tag{IP}$$

where $\boldsymbol{y}$ are the *decision variables*, of which $y_1, \ldots, y_n$ are integer variables and $y_{n+1}, \ldots, y_{n+p}$ are continuous variables. Furthermore, $\boldsymbol{g} : \mathbb{Z}^n \times \mathbb{R}^p \longrightarrow \mathbb{R}^m$, and $\mathbf{0}$ is a null vector of dimension m. Note that maximizing a function is equivalent to minimizing its negative, and an equality constraint can be represented by two inequalities, which renders (IP) a complete formulation.

For an integer program formulated as in (IP), the set

$$Y = \{\boldsymbol{y} \in \mathbb{Z}^n \times \mathbb{R}^p : \boldsymbol{g}(\boldsymbol{y}) \leq \mathbf{0}\}$$

is named the *feasible region* of the problem, and a vector $\boldsymbol{y} \in Y$ is a *feasible solution*. A feasible solution $\boldsymbol{y}^* \in Y$ is *optimal* if, and only if, there is no other feasible solution with a lower value of the *objective function* $f : \mathbb{Z}^n \times \mathbb{R}^p \longrightarrow \mathbb{R}$, i.e., $\boldsymbol{y}^*$ is optimal $\iff f(\boldsymbol{y}^*) \leq f(\boldsymbol{y}), \forall \boldsymbol{y} \in Y$.

Note that even if a problem is feasible ($Y \neq \varnothing$), it may not have an optimal solution, e.g., if the feasible region is unbounded and the objective function has no global minimum. Furthermore, if an optimal solution exists, it may no be unique.

Beyond the practical applications of integer programming, its computational complexity renders it an important theoretical model. It is easy to see that integer programming is an NP-hard problem (NEMHAUSER; WOLSEY, 1999). In fact, one of Karp's 21 NP-complete problems (KARP, 1972) is a special case of integer programming with no objective function (constraint satisfaction problem) and solely binary variables.

1.2 MIXED-INTEGER LINEAR PROGRAMS

MILP is a subset of IP in which the objective and the constraints are all linear functions and the problem requires integer and continuous variables. Formally, an MILP can be formulated as

$$\begin{aligned} \min_{\boldsymbol{y}} \quad & \boldsymbol{c}^T \boldsymbol{y} \\ \text{s.t.} \quad & A\boldsymbol{y} \leq \boldsymbol{b} \\ & \boldsymbol{y} \in \mathbb{Z}^n \times \mathbb{R}^p, \end{aligned} \tag{MILP}$$

where $A \in \mathbb{R}^{m \times (n+p)}$ is the constraint matrix, $\boldsymbol{b} \in \mathbb{R}^m$ is the right-hand side vector, and $\boldsymbol{c} \in \mathbb{R}^{n+p}$ is the cost vector. An *instance* of an MILP problem is specified by a tuple $(\boldsymbol{c}, \boldsymbol{b}, A, n)$.

The significance of this class of problems has already been recognized by Dantzig (1960). Many well-known problems can be formulated through MILP, such as the Traveling Salesperson Problem (TSP) and the map coloring problem. Furthermore, continuous nonlinear functions can be approximated to arbitrary quality by piecewise linear functions, which admit an MILP formulation (CAMPONOGARA; NAZARI, 2015). In other words, MILP is a powerful tool for approximating optimization problems with continuous nonlinearities.

1.3 SOLVING MILP PROBLEMS

Although MILP offers powerful models for a wide range of problems, solving such problems is unarguably hard. In fact, the NP-complete problem formulated by Karp (1972) only contains linear terms, which renders it a special case of MILP and, thus, assuming P≠NP, classifies MILP problems as NP-hard. However, despite the intractable nature, there are efficient and reliable algorithms and software solutions for the computation of optimal and approximate solutions to MILP problems (BENGIO; LODI; PROUVOST, 2021). Furthermore, the applications of MILP often require high-quality solutions in a limited time, which motivate the development of heuristic approaches, i.e., approaches that trade optimality (or feasibility) guarantees for a tractable running time.

1.3.1 The Branch-and-Bound Algorithm

The branch-and-bound algorithm follows a divide-and-conquer approach. An MILP problem is divided into smaller, easier problems, and the solution to these problems is combined such that a solution to the original problem is found (WOLSEY, 1998).

An MILP problem is divided by decomposing its feasible region. Given a problem P as in (MILP) with feasible region $Y = \{\boldsymbol{y} \in \mathbb{Z}^n \times \mathbb{R}^p : A\boldsymbol{y} \leq \boldsymbol{b}\}$, a decomposition of its feasible region $Y_1, \ldots, Y_K$ is such that $Y = Y_1 \cup \cdots \cup Y_K$. In this context, a subproblem is

$$\begin{aligned} P^{(k)} : \min_{\boldsymbol{y}} \quad & \boldsymbol{c}^T\boldsymbol{y} \\ \text{s.t.} \quad & \boldsymbol{y} \in Y_k, \end{aligned} \tag{1.1}$$

for which $\boldsymbol{y}^k$ is the optimal solution and $z^k = \boldsymbol{c}^T\boldsymbol{y}^k$ is the optimal cost. If the k-th subproblem is infeasible, it is assumed that $z^k = \infty$. If $k^* = \arg\min_k z^k$, then z^{k^*} is the optimal value of the MILP problem P and $\boldsymbol{y}^{k^*}$ is its optimal solution.

A decomposition is useful for a divide-and-conquer approach if the resulting subproblems are significantly easier to solve than the original problem. One way to achieve this is by decomposing the feasible region on the values for the integer variables. If this decomposition strategy is performed recursively until all integer variables have only one feasible assignment in each Y_k, then each $P^{(k)}$ is an LP problem, which allows us to compute, for each k, the optimal solution in polynomial time. For example, consider an MILP problem with 3 binary variables, i.e., with $Y \subseteq \{0,1\}^3 \times \mathbb{R}^p$. By recursively decomposing Y on the possible assignments for each binary variable, the tree structure of Fig. 2 could be assembled.

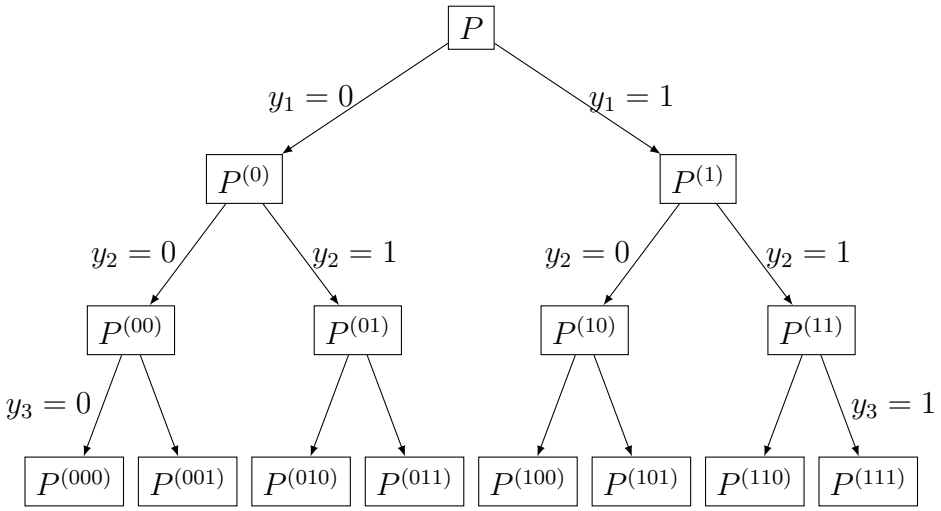


Figure 2 – Complete decomposition of an MILP problem on its 3 binary variables. Nodes are annotated with the associated subproblems. Assuming that the problem has no other integer variables, the subproblems at the leaf nodes are LP problems.

Unfortunately, completely decomposing an MILP problem through the integer variable assignments is only possible for very small, bounded problems. If the problem is unbounded in the

integer variables, the complete decomposition would result in an infinite recursion. Furthermore, the number of leaf nodes grows exponentially with the size of the problem[1].

The branch-and-bound algorithm follows an implicit approach that uses upper and lower bounds to avoid indefinitely dividing subsets from the decomposition. Let P be an MILP problem formulated as in (MILP) and $Y_1, \ldots, Y_K$ be a decomposition of the feasible region Y. If $\underline{y}^k$ and $\overline{y}^k$ are lower and upper bounds to z^k (optimum cost of $P^{(k)}$), for every $k = 1, \ldots, K$, then, it is true that

$$\min_k \underline{y}^k \leq z \leq \min_k \overline{y}^k,$$

where z is the optimum cost of P. In other words, we can compute bounds of the root problem as $\underline{y} \leftarrow \min_k \underline{y}^k$ and $\overline{y} \leftarrow \min_k \overline{y}^k$. Finally, if $\underline{y}^k \geq \overline{y}$ for a given k, then the optimal solution of $P^{(k)}$ will not be an optimal solution of P, because it is guaranteed that another subproblem has a better feasible solution. In other words, even if the optimal solution of $P^{(k)}$ is unknown, it is possible to disregard Y_k in the decomposition, i.e., Y_k does not need to be further subdivided.

For example, let P be an MILP problem and $Y = Y_1 \cup Y_2$ be a decomposition such that $Y_1 = \{\boldsymbol{y} \in Y : y_1 \leq 2\}$ and $Y_2 = \{\boldsymbol{y} \in Y : y_1 \geq 3\}$. Suppose that the LP relaxations[2] of $P^{(1)}$ and $P^{(2)}$ were solved to optimality, giving lower bounds $\underline{y}^1 = 20$ and $\underline{y}^2 = 15$, and that a feasible solution to P is known in Y_2 such that the upper bound $\overline{y}^2 = 17$ is known. Fig. 3 illustrates the decomposition along with respective bounds in the form of a tree. Because the lower bound of $P^{(1)}$ is greater than the original problem's upper bound (as $\overline{y} = \min_k \overline{y}^k$), the optimal solution is definitely not in Y_1, so this set is ignored in the decomposition and not further refined.

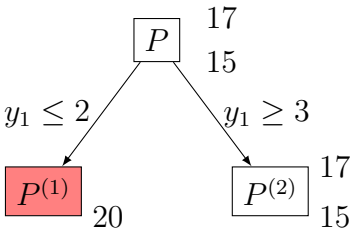


Figure 3 – Example of pruning the decomposition of an MILP problem based on known bounds. The lower (resp. upper) bounds of each (sub)problem are annotated at the bottom (resp. top) right of each node. The node associated to set Y_1 is painted red to indicate it is pruned based on the bounds of the root problem, which are updated based on the bounds from $P^{(2)}$.

Because of the usual tree representation, the operation of disregarding a set in the decomposition is often called *pruning*. Three basic rules can be listed that lead to pruning of the branch associated to set Y_k:

[1] In this context, the size of the problem is measured as the number of binary variables that the equivalent reformulation as a binary MILP would contain. The number of leaf nodes grows exponentially (with base 2) with respect to the number of such binary variables.

[2] The linear programming problem obtained by ignoring the integrality constraints of an MILP is called its *LP relaxation*.

Optimality Subproblem $P^{(k)}$ was solved to optimality (e.g., through its LP relaxation having an optimal solution that respects the integrality constraints);

Bound The associated lower bound is higher than the upper bound of the root problem ($\underline{y}^k \geq \overline{y}$);

Infeasibility $Y_k = \varnothing$.

The two key components of a branch-and-bound algorithm are the pruning system based on *bounds*, as discussed above, and the rules for subdividing the sets in the decomposition, or *branching*. A simple strategy for branching is to choose an integer variable that has taken a fractional value in the optimal solution to the LP relaxation and split the problem on this factional value. For example, let Y_k be a set in the decomposition of an MILP problem, and $P^{(k)}$ be the associated subproblem. Let $\widetilde{\boldsymbol{y}}^k$ be the solution to the LP relaxation of $P^{(k)}$, and suppose that the integer variable y_3 takes value 3.67 in $\widetilde{\boldsymbol{y}}^k$. Following the proposed branching strategy on y_3, the sets Y_{k_1} and Y_{k_2} would be created such that

$$Y_{k_1} = \{\boldsymbol{y} \in Y_k : y_3 \leq 3\}\,,\; Y_{k_2} = \{\boldsymbol{y} \in Y_k : y_3 \geq 4\}\,,$$

and the set Y_k would then be replaced in the decomposition by these two new sets. Fig. 4 illustrates this example.

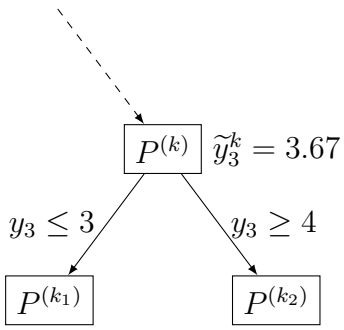


Figure 4 – Example of branching on a given set Y_k which is part of the decomposition of an MILP problem. Only the relevant part of the tree is illustrated (indicated by the dashed arrow). The optimum value of y_3 (the selected integer variable for branching) in the LP relaxation of $P^{(k)}$ is annotated next to the appropriated node.

Note that by branching on the fractional value of an integer variable in the optimal solution to the LP relaxation, the optimal solution of the LP relaxation becomes infeasible in the LP relaxations of the new sets[3]. Therefore, after branching, the best lower bound will necessarily increase. Using the example above to make this result tangible, it is possible to state that $\widetilde{\boldsymbol{y}}^k$ is infeasible for the LP relaxations of both $P^{(k_1)}$ and $P^{(k_2)}$. Therefore, $\max\{\underline{y}^{k_1}, \underline{y}^{k_2}\} > \underline{y}^k$.

There are many more intricate details to the construction of a branch-and-bound algorithm, such as the strategy for choosing a set in the decomposition to refine, the amount of information from each node of the tree to be stored, efficient reoptimization, and computing

[3] This is true unless there are degeneracies such as multiple solutions in the LP relaxation

the bounds using the dual. The reader is pointed to Wolsey (1998) and Vanderbei (1998) for advanced topics and detailed examples.

1.3.2 Heuristics

Although branch-and-bound provides an efficient algorithmic approach to solve an MILP problem, there is no guarantee that it will find a feasible (let alone an optimal) solution considering the NP-hardness of the problem. The development of heuristic or approximation algorithms is justified in many ways, as can be seen in Wolsey (1998), Chapter 12. The major one, for this work, is from practical applications with significant running time cost. In simple terms, such applications require good solutions quickly, rather than optimal solutions in an unknown time-horizon.

Heuristic algorithms are an umbrella term that encompass several different algorithms with different characteristics. Overall, distinguishing features of heuristics are on the feasibility and optimality of the solutions returned. For feasibility, it is important to know if there are feasibility guarantees on the heuristic solution or, at least, an expectation on how often the heuristic will return infeasible solutions. Similarly, it is relevant to understand whether the solutions provided are guaranteed to be within a limited distance (in terms of the objective function) of the optimal solutions, or if there is an expected value for such distance. Such heuristics with quality guarantees are referred to as approximation algorithms in the technical literature

Common examples of heuristic algorithms are greedy algorithms and local search approaches (NEMHAUSER; WOLSEY, 1999; WOLSEY, 1998). Greedy heuristics construct the solution incrementally by selecting, at each step, the alternative that best improves the objective function. Take the Symmetric TSP (STSP) as an example. A greedy heuristic for such problem could be to add to the solution the edge with the smallest cost. Note that this greedy heuristic will always return a feasible solution, although it is not expected that the solution will be close to optimal.

Local search algorithms take a feasible solution and try to improve the objective function by performing only limited changes. Given a complete tour (feasible solution) for the STSP, a small deviation can be achieved by removing two non-consecutive edges that are part of the solution and adding two *different* edges that reconnect the two disjoint paths. This way, a new tour will be achieved that differs from the original by two edges. In other words, such local search is limited to a 2-neighborhood of the original tour.

A general heuristic (or metaheuristic) algorithm for MILP problems is *diving* (FISCHETTI; LODI, 2011). Diving methods solve the LP relaxation, fix integer variables that assume fractional values, and update the LP relaxation. This process is repeated until there are no more integer variables to be fixed. It is often called diving because it is equivalent to quickly

navigating to a leaf node in a branch-and-bound tree.

Recently, heuristics based on machine learning models have been proposed (BENGIO; LODI; PROUVOST, 2021). This will be discussed in Section 3.3.

1.3.2.1 Matheuristics

Matheuristics is the name given to heuristic algorithms that use a mathematical programming model at its core. In fact, the diving algorithm introduced above can be seen as a form of matheuristic. In this section, however, the focus is on algorithms that actively optimize mathematical programming models. Relaxation induced neighborhood search (RINS) (MANIEZZO; BOSCHETTI; STÜTZLE, 2021) exemplifies the distinction intended in this chapter.

The RINS algorithm aims to improve a feasible solution by solving a smaller MILP problem. A neighborhood around a feasible solution is defined by fixing the values of some integer variables. The search on this neighborhood is performed by adding the variable fixing constraints to the original problem, effectively reducing its feasible region. Let P be an MILP problem as in (MILP), $\boldsymbol{y}^h$ be a feasible solution, and $\widetilde{\boldsymbol{y}}$ the solution to the LP relaxation of P. Let $J = \{j = 1, \ldots, n : y_j^h = \widetilde{y}_j\}$ be the index set of integer variables to fix. Then, the sub-MILP problem

$$\begin{aligned} \min_{\boldsymbol{y}} \quad & \boldsymbol{c}^T \boldsymbol{y} \\ \text{s.t.} \quad & A\boldsymbol{y} \leq \boldsymbol{b} \\ & y_j = y_j^h, \forall j \in J \\ & \boldsymbol{y} \in \mathbb{Z}^n \times \mathbb{R}^p \end{aligned}$$

can be solved, e.g., using branch-and-bound. Note that all integer variables that assume the same value in the feasible solution and in the solution to the LP relaxation are fixed, thus, reducing the search space, but abdicating from optimality guarantees.

As the RINS example above illustrates, a mathematical programming model plays a central role in a matheuristic (FISCHETTI; FISCHETTI, 2016). Because of their flexibility, matheuristics are also used within branch-and-bound algorithms, e.g., to find tighter bounds and accelerate the time to reach the first feasible solution (FISCHETTI; FISCHETTI, 2016; MANIEZZO; BOSCHETTI; STÜTZLE, 2021).

2 DEEP LEARNING

The advent of deep learning has revolutionized several fields, offering unprecedented capabilities in pattern recognition, decision-making, and problem-solving (GOODFELLOW; BENGIO; COURVILLE, 2016). In the realm of combinatorial optimization, particularly MILP, traditional methods often encounter computational bottlenecks when solving large-scale instances. However, recent strides in deep learning have opened up exciting avenues for developing effective primal heuristics. This chapter delves into the background of deep learning, focusing on the fundamental principles of supervised learning and neural network architectures. The understanding of such concepts is essential for exploring the design and implementation of deep-learning-based solution prediction models tailored to MILP instances.

2.1 SUPERVISED LEARNING

Supervised learning can be seen as the problem of finding a function that best associates inputs x to outputs y given a *training set* with finitely many examples of such inputs and outputs (GOODFELLOW; BENGIO; COURVILLE, 2016). Although the machine learning (ML) area has attracted plenty of attention in the last decade, this learning problem is not new. In fact, the core concepts were already established in the 1960s and 1970s (VAPNIK, 2000). This section's approach to supervised learning will be that of statistical learning theory, based on Vapnik (2000), but with a more modern notation, derived from pattern recognition (BISHOP, 2006; HASTIE; TIBSHIRANI; FRIEDMAN, 2009).

2.1.1 Supervised learning algorithm

Supervised learning will be defined here as a problem of estimating a function that minimizes the risk. Let $\mathcal{X}$ and $\mathcal{Y}$ be the input and output space, and $\mathcal{P}$ be a joint probability distribution [1] over $\mathcal{X} \times \mathcal{Y}$. To evaluate a function $f : \mathcal{X} \longrightarrow \mathcal{Y}$ with respect to its capacity to perform the correct association between an input $x \in \mathcal{X}$ and an output $y \in \mathcal{Y}$, one measure's the discrepancy, or the *loss*, $\ell(y, f(x))$. In this context, the *risk* associated to f over the joint distribution $\mathcal{P}$ is the expected value of the loss function, or

$$R(\mathcal{P}, f) = \int_{\mathcal{X}\times\mathcal{Y}} \ell(y, f(x))\mathcal{P}(x, y).$$

The machine learning paradigm is that the joint probability function is fixed but unknown, and one only has access to a finite number of samples (VAPNIK, 2000). This idea is best represented through a *dataset*, which is a finite set $\mathcal{D}$ of independent and identically distributed

[1] Or a joint probability density function, in the case of continuous spaces.

(i.i.d.) samples drawn according to $\mathcal{P}$. In this context, the *empirical risk* associated to a function f over a dataset $\mathcal{D}$ is

$$R_{\text{emp}}(\mathcal{D}, f) = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} \ell(y, f(x)).$$

In this work, it is considered that the function to be estimated is chosen from a *parametric model*[2], which is a family of functions $f_\theta : \mathcal{X} \longrightarrow \mathcal{Y}$ defined over a parameter space $\Theta \ni \theta$. The problem then becomes that of finding a parameter vector that minimizes the empirical risk from a dataset $\mathcal{D}$, or *fitting* a model to a dataset, denoted

$$\min_{\theta\in\Theta} \mathcal{L}(\theta), \tag{2.1}$$

where

$$\mathcal{L}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} \ell(y, f_\theta(x)) \tag{2.2}$$

is an alternate, commonly found notation for the empirical risk, which often carries the name *cost function* (MURPHY, 2013; GOODFELLOW; BENGIO; COURVILLE, 2016). Both "empirical risk" and "cost function" will be used interchangeably throughout this dissertation, although the former has a stronger connection to learning theory, while the latter is more related to practical matters.

A supervised learning algorithm is essentially an algorithm to optimize a cost function given a model and a dataset. In fact, it is possible to define a supervised learning algorithm as a simple recipe: "combine a specification of a dataset, a cost function, an optimization procedure and a model" (GOODFELLOW; BENGIO; COURVILLE, 2016). Many different algorithms exist that are suitable for different components of this recipe. For example, there are algorithms which are tailored for the synthesis of decision-tree models (BREIMAN et al., 2017). The least squares algorithm (used in the example of Section 2.1.2) can be seen as a supervised learning algorithm for polynomial models and the squared error loss function.

Although the loss function can be seen as part of the specification of the problem, such "ideal" function may not be suitable for the algorithm. For example, a *surrogate* loss function is necessary when the original loss function is not differentiable and the learning will be performed through gradient-based optimization, as will be discussed in Sec. 2.2.1. More generally, the cost function can be seen as a design choice, as the goal of learning is to achieve *generalization*, as presented in the following.

2.1.2 Generalization and overfitting

Blindly minimizing the empirical risk can lead to a problem named *overfitting*. Overfitting happens when the empirical risk does not reflect the true risk, that is, when a function is *fit* for the dataset, but not for the underlying data distribution.

[2] This is in contrast to non-parametric models, whose functions' parameters are defined in terms of the dataset (MURPHY, 2013).

This concept is best understood through an example. Suppose $\mathcal{X}, \mathcal{Y} \subseteq \mathbb{R}$ and that a dataset $\mathcal{D}$ with 40 i.i.d. samples is obtained such as illustrated in Fig. 5. It is desired to find a function that minimizes the risk measured through the squared error loss

$$\ell(y, f(x)) = (y - f(x))^2.$$

The models $f^{(1)}, f^{(2)}, f^{(3)}$ are polynomials of degree 1, 3, and 15, whose parameters are the weights of the polynomials. These models are adjusted using least squares algorithm, resulting in parameter vectors $\theta_1^*, \theta_2^*, \theta_3^*$ that achieve a global minimum of the empirical risk with their respective models. The performance of these models is illustrated in Fig. 5.

Note that model $f^{(3)}$ achieves the lowest empirical risk on $\mathcal{D}$. However, $f^{(2)}$ is the one that seems to best associate inputs to outputs, considering a visual intuition of the underlying data distribution. One could say that $f^{(3)}$ is *too complex* for the underlying distribution, which leads to it being more tightly adjusted to the noise present in the dataset rather than on the underlying data distribution (MURPHY, 2013). On the other hand of the spectrum is the $f^{(1)}$ model, which is not complex enough to model the desired behavior. While $f^{(3)}$ is overfitting the data, $f^{(1)}$ is underfitting it.

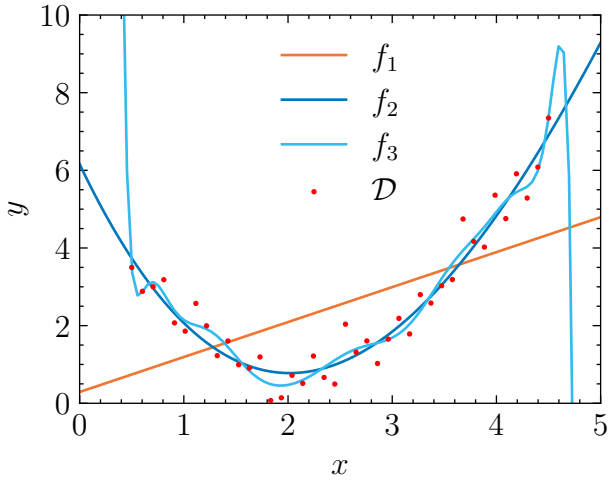


Figure 5 – Example of overfitting. The three models shown ($f^{(1)}, f^{(2)}, f^{(3)}$) are families of polynomials of degree 1, 3 and 15, respectively. The optimal parameter vectors θ_1^*, θ_2^* and θ_3^* were optimized to the dataset $\mathcal{D}$ using a the least squares algorithm. The empirical risks are $\mathcal{L}(\theta_1^*) = 79.67$, $\mathcal{L}(\theta_2^*) = 7.64$, and $\mathcal{L}(\theta_3^*) = 5.48$.

The level of overfitting that a function f presents could be determined through the *generalization gap*, which is the difference between the empirical risk and the true risk $R(\mathcal{P}, f) - R_{\text{emp}}(\mathcal{D}, f)$ (MURPHY, 2013). More specifically, a function can be said to overfit the dataset if it has a high generalization gap. As the underlying distribution $\mathcal{P}$ is not known, the generalization gap is approximated by *splitting* the dataset $\mathcal{D}$ into a training set $\mathcal{D}_{\text{train}}$ and a test set $\mathcal{D}_{\text{test}}$. The idea is that the parameters of the model are adjusted according to the empirical risk of $\mathcal{D}_{\text{train}}$, while $\mathcal{D}_{\text{test}}$ is reserved for estimating the true risk, also called *generalization error*. This way,

the resulting function cannot be overfitted to $\mathcal{D}_{\text{test}}$, which is kept as an untouched source of information of the underlying distribution. Finally, the generalization gap can be estimated as

$$R_{\text{emp}}(\mathcal{D}_{\text{test}}, f) - R_{\text{emp}}(\mathcal{D}_{\text{train}}, f).$$

2.1.3 Hyperparameter tuning

Hyperparameters are the settings that modify the behavior of a supervised learning algorithm (GOODFELLOW; BENGIO; COURVILLE, 2016). These hyperparameters can be configurations of the optimization algorithm or parameters of the model that are not adjusted by the algorithm. Hyperparameter adjustment is impactful both in the runtime of the algorithm and in the quality of the function returned.

In the polynomial fitting example above (illustrated in Fig. 5), the polynomial degree is a hyperparameter. Different values for this hyperparameter result in totally different parameter spaces Θ over which the optimizer will search for a risk minimizer. If a ridge regression algorithm is used instead of the least squares to find the optimal function, then a hyperparameter is the weight of the ℓ_2-norm of θ. Such hyperparameter does not alter the function space associated with Θ but guides the optimizer towards different optima.

Given the impact of the hyperparameters in the function space, another way of looking at hyperparameters is as a means to encode prior beliefs of the underlying data distribution (MURPHY, 2013). Again on the polynomial fitting example, if one believes that the outputs are approximately related to the inputs through a 5-th degree polynomial, then this information can be directly encoded in the hyperparameters of the algorithm. Another example is the belief that the output has a seasonal component with respect to a given input, which could be used to restrict the parameter space to periodic functions.

Unfortunately, and mainly for complex, high-dimensional spaces, the existence of strong and sufficient prior beliefs is rarely assumed in supervised learning problems. On the contrary, the tuning, or optimization, of hyperparameters is a common step in machine learning projects. A new hold-out set is necessary to compare different choices of hyperparameter values. This is because hyperparameters that control model capacity are always biased towards greater capacity (GOODFELLOW; BENGIO; COURVILLE, 2016). In other words, if the impact of different hyperparameter values is evaluated on the training dataset $\mathcal{D}_{\text{train}}$, it will likely lead to choosing the value that implies a greater model capacity, which leads to overfitting. As it is assumed that the test dataset $\mathcal{D}_{\text{test}}$ is not available during training to avoid a biased estimation of the generalization error (MURPHY, 2013), a second held-out set is necessary.

In practice, the available data $\mathcal{D}$ is partitioned into three disjoint sets:

$\mathcal{D}_{\text{train}}$ the training set, which is used for fitting the model, i.e., the best parameter vector θ^* is chosen such that the cost function over $\mathcal{D}_{\text{train}}$ is minimized;

$\mathcal{D}_{\text{val}}$ the validation set, which is used to measure the impact of different values for the hyperparameters; and

$\mathcal{D}_{\text{test}}$ the test set, which is used to estimate the generalization error (empirical risk) of the function returned by the supervised learning algorithm with the best hyperparameter values found.

A usual ratio is to set apart 50% of the samples in $\mathcal{D}$ for the training set, 25% for the validation set, and 25% for the test set (HASTIE; TIBSHIRANI; FRIEDMAN, 2009).

2.2 DEEP NEURAL NETWORKS

Deep feedforward neural networks "are the quintessential deep learning models" (GOODFELLOW; BENGIO; COURVILLE, 2016). Feedforward neural networks (NNs) are compositions of differentiable functions such that the information flows without any sort of feedback connection. An NN model can be written as

$$\begin{aligned} f_\theta : \mathbb{R}^n &\longrightarrow \mathbb{R}^m \\ x &\longmapsto f_\theta(x) = f_\theta^{(L)}(f_\theta^{(L-1)}(\cdots(f_\theta^{(1)}(x))\cdots)), \end{aligned}$$

in which each $f_\theta^{(l)}$, $l = 1, \ldots, L$, is a *layer* of the network, and L, the number of layers. All layers except the last one are called *hidden* layers of the network. The last one is called the *output* layer. A simple layer of an NN can be a linear combination of its inputs followed by a nonlinear function $\sigma : \mathbb{R}^d \longrightarrow \mathbb{R}^d$, as in

$$f_\theta^{(l)} = \sigma\left(W^{(l)}x + c^{(l)}\right).$$

Note that an NN's parameter vector can be seen as the concatenation of the parameters of all layers, i.e., $\theta = (W^{(1)}, c^{(1)}, W^{(2)}, c^{(2)}, \ldots, W^{(L)}, c^{(L)})$. In the hidden layers, the default choice for σ, the *activation function*, is the rectified linear unit, or ReLU (GOODFELLOW; BENGIO; COURVILLE, 2016), defined element-wise as $\sigma(z) = \max\{0, z\}$. The activation function of the output layer is application-dependent.

Deep neural networks (DNNs) are a larger family of models. Although DNNs are also composed of differentiable functions, these functions are assembled in any form of directed acyclic graphs (MURPHY, 2013). Note that this opens up complex architectures that may contain feedback, memory, convolutions, etc. (GOODFELLOW; BENGIO; COURVILLE, 2016).

2.2.1 Gradient-based learning

Because of the inherent nonlinearities of NNs, the cost (or empirical risk) minimization problem (2.1) becomes nonconvex under the most common loss functions (GOODFELLOW; BENGIO; COURVILLE, 2016). As NNs are, by definition, differentiable, they are usually trained

by using gradient-based optimizers. The gradient-descent method is one of the oldest algorithms for finding local minima of a function (CAUCHY, 1847). Such algorithms take successive steps of the form

$$\theta \leftarrow \theta - \lambda \nabla \mathcal{L}(\theta), \tag{2.3}$$

such that with a sufficiently small *learning rate* $\lambda > 0$, the parameter vector is always modified such that the cost function is minimized.

A practical problem of gradient descent is that large training sets are often necessary to achieve low generalization errors, which, in turn, increases the computational cost of calculating the gradient. A solution is to break down the parameter vector update step (2.3) stochastically in an algorithm called stochastic gradient descent (SGD). This is intuitively sound because the gradient of the empirical risk is already an expectation of the gradient of the actual risk. Thus, instead of approximating the gradient using the entire training dataset, it is possible to approximate the gradient by sampling a limited number of examples from it. In other words, instead of computing the gradient as

$$g = \nabla \mathcal{L}(\theta) \leftarrow \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \nabla_\theta \ell(y, f_\theta(x)),$$

in SGD it is computed as

$$\widetilde{g} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{(x,y) \in \mathcal{B}} \nabla_\theta \ell(y, f_\theta(x)),$$

where $\mathcal{B} \subset \mathcal{D}_{\text{train}}$ such that $|\mathcal{B}| \ll |\mathcal{D}_{\text{train}}|$, and then update the parameter vector as

$$\theta \leftarrow \theta - \lambda \widetilde{g}.$$

The set $\mathcal{B}$ is called a *minibatch* and it is usually sampled uniformly from the training set (GOODFELLOW; BENGIO; COURVILLE, 2016).

Although gradient descent can provide convergence guarantees to local optima of nonconvex functions under mild conditions, it has been regarded as a slow and unreliable optimization method. However, deep learning applications stand out, as it has been empirically shown that SGD is a reliable method to achieving sufficiently small cost function values (GOODFELLOW; BENGIO; COURVILLE, 2016). Arguably, this can be traced back to the generalization problem. Differently from the traditional optimization paradigm, in which the performance is evaluated on the function being optimized, a learning algorithm optimizes indirectly, i.e., the performance metric of interest (generalization error) is not directly optimized. Therefore, achieving local or global optima is not as relevant for machine learning as it is for the field of optimization. In fact, local optima can be a source of overfitted parameter vectors.

A prominent problem of SGD is that it can become very slow in face of flat regions, saddle points, and noisy gradients, all of which are abundant in deep learning (GOODFELLOW; VINYALS; SAXE, 2015; GOODFELLOW; BENGIO; COURVILLE, 2016). To counter-act

such effects, one can add a *momentum* to the computation of the gradient estimate, akin to the physical intuition (NESTEROV, 1983; POLYAK; JUDITSKY, 1992). Another approach is to use an *adaptive learning rate*, which is dynamically adjusted based on the values of the computed gradient (JACOBS, 1988). The Adam optimizer (KINGMA; BA, 2015) combines both momentum and adaptive learning rates, with its name actually deriving from "adaptive moments."

2.2.2 Graph Neural Networks

Graph Neural Networks (GNNs) are a particular kind of DNN that is optimized for taking graphs as inputs (Sanchez-Lengeling et al., 2021). Note that this is not trivial. For example, NNs are usually defined over real-valued vector spaces, which would require a vector encoding of a graph for their representation. Such an encoding could be achieved through the adjacency matrix, but that would not account for graph symmetries, as distinct adjacency matrices can represent the same graph[3].

A GNN takes as input a graph with feature vectors associated to each node. GNNs are also feedforward networks, i.e., they can be represented by a stacking of layers. Each layer of the GNN works by propagating feature information between neighboring nodes, which can be seen as *message-passing* (GILMER et al., 2017). More specifically, a layer computes messages that each node emits to its neighbors based on their feature vectors. Then, each node's output feature vector is computed based on the input feature vector and the messages emitted by the node neighbors.

Let $G = (V, E)$ be a graph, $H = \left\{h_v \in \mathbb{R}^d : v \in V\right\}$ be an associated set of node feature vectors, and $f_\theta = f_\theta^{(L)} \circ \cdots \circ f_\theta^{(1)}$ be a GNN with L layers. Each layer of the GNN has two major components: a message function $M(\cdot)$ and an update function $U(\cdot)$. The message function computes the messages emitted by each node, while the update function feeds on these messages to update each node's feature vector. Putting it into terms, each layer $f_\theta^{(l)}$, $l = 1, \ldots, L$, of the GNN transforms the previous layer's feature vectors $H^{(l-1)} = \left\{h_v^{(l-1)} : v \in V\right\}$ through

$$h_v^{(l)} = U^{(l)}\left(h_v^{(l-1)}, \left\{M^{(l)}(h_u^{(l-1)}) : u \in \mathcal{N}(v)\right\}\right), \forall v \in V,$$

where $\mathcal{N}(v)$ denotes the set of neighbors of v, excluding v itself, unless a self-loop edge is present. Note that each layer maps feature vectors into features vectors, i.e., $f_\theta^{(l)} : H^{(l-1)} \mapsto H^{(l)}$. The structure of the graph is embedded into this computation by the neighbors function $\mathcal{N}(\cdot)$.

As each node can have an arbitrary number of neighbors, it is reasonable for update functions to contain some sort of aggregation, or *pooling*, mechanism, i.e., a way to compute a fixed-size representation of the messages received. A common choice is to sum all messages element-wise, as each message is a vector of the image of $M^{(l)}$ and, thus, has the same dimension.

[3] In fact, any permutation (row- or column-wise) of an adjacency matrix results in the same graph.

Other possibilities are to aggregate through element-wise averaging, maximum, or use some sort of attention mechanism that can be learned along the model parameters (VELIČKOVIĆ et al., 2018). Furthermore, a message function can easily be extended to consider edge weights (or even edge features) along with feature vectors of the neighbors.

As an example, let the input space be the set of colored graphs and consider a handcrafted GNN, as illustrated in Fig. 6. A colored graph will be represented as an undirected graph $G = (V, E)$ paired with a set of node features $H^{(0)}$ such that for each node $v \in V$, there exists a feature vector $h_v^{(0)} \in H^{(0)}$ with an encoding of a color in cyan-magenta-yellow (CMY) format, i.e., $h_v^{(0)} \in [0, 1]^3$. Let f_θ be a GNN with two layers ($L = 2$), such that both layers perform the same operation ($f_\theta^{(1)} = f_\theta^{(2)}$). The message functions of both layers is the complementarity function, which maps each node's color to its complementary color ($M(h_v) = \mathbf{1}_3 - h_v$). The update functions is a simple average between all messages received. In summary, the GNN will update each node with the average complementary color of all of its neighbors.

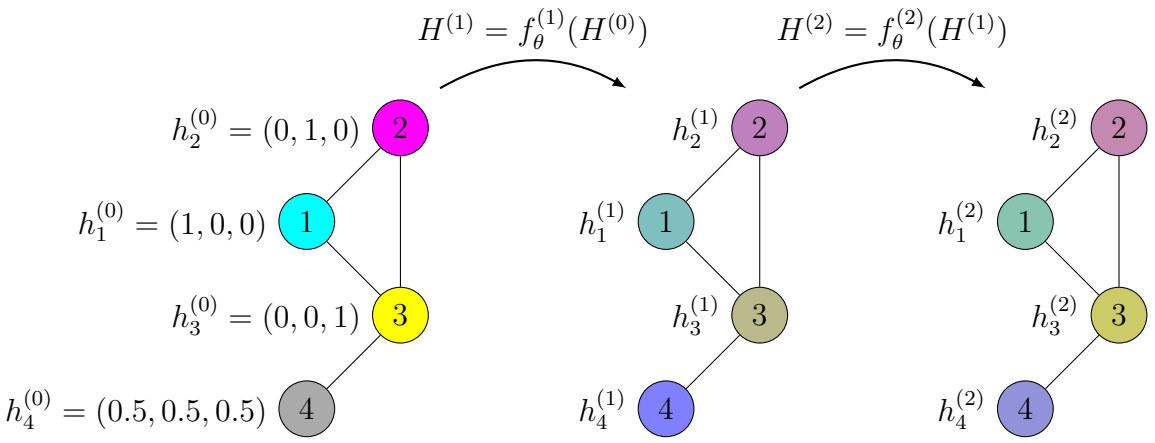


Figure 6 – Example of a GNN application to a colored graph. The node features are color in CMY format. Each layer of the model updates the node features with the average complementary color of all of its neighbors.

Kipf and Welling (2017) have proposed a GNN architecture, in which they use a linear combination of the node features as the messages and weigh them by the number of neighbors of both the emitter and the receiver. Their update function aggregates the messages by summing them and then feeds the result to a single-layer NN with ReLU activation. More precisely,

$$\begin{aligned} h_v^{(l)} &= U^{(l)}\left(h_v^{(l-1)}, \left\{\frac{1}{c_{v,u}} M^{(l)}(h_u^{(l-1)}) : u \in \mathcal{N}(v)\right\}\right) \\ &= \text{ReLU}\left(B^{(l)} h_v^{(l-1)} + W^{(l)} \sum_{u \in \mathcal{N}(v)} \frac{h_u^{(l-1)}}{c_{v,u}}\right), \end{aligned} \tag{2.4}$$

where $c_{v,u} = \sqrt{|\mathcal{N}(v)|}\sqrt{|\mathcal{N}(u)|}$, and the matrices $W^{(l)}$ and $B^{(l)}$ are trainable parameters of layer l (Sanchez-Lengeling et al., 2021).

Another relevant GNN architecture is the one proposed by Hamilton, Ying and Leskovec (2017) with the SAGE (SAmple and aGgrEgate) model. The authors also propose to use a

single-layer NN to generate the new feature vectors, but they use the identity as the message function and aggregate them with more complex functions, such as an LSTM (Long Short-Term Memory network). In such case, the GNN could be written

$$\begin{aligned} h_v^{(l)} &= U^{(l)}\left(h_v^{(l-1)}, \left\{h_u^{(l-1)} : u \in \mathcal{N}(v)\right\}\right) \\ &= \sigma\left(W^{(l)}\left[\text{LSTM}\left(h_u^{(l-1)}\right)_{u\in\mathcal{N}(v)}, h_v^{(l-1)}\right]\right), \end{aligned} \tag{2.5}$$

where square brackets indicate a vector concatenation (Sanchez-Lengeling et al., 2021). Note that the LSTM function could be replaced by any of the aggregation functions the authors propose.

Part II

Materials and Methods

3 SOLUTION PREDICTION MODELS FOR MILP PROBLEMS

This chapter introduces the methods available for training deep learning models for predicting solutions of MILP problems. The ability to efficiently predict solutions plays a pivotal role in the development of learning-based heuristics. In other words, this chapter is a bridge between Chapters 1 and 2 with a focus on (mat)heuristics.

This chapter begins by discussing the process of embedding of MILP problems, which involves transforming problem instances into a suitable format for deep learning models. Within this context, feature engineering and graph approaches are explored to represent the intricate relationships between the components of MILP problems. Moving forward, the methodologies employed in training deep learning models fed with embeddings of MILP problem instances are presented, highlighting the challenges and opportunities posed by the availability of multiple feasible solutions. The chapter ends with the approaches one can use to create primal (mat)heuristics from solution prediction models.

3.1 EMBEDDING OPTIMIZATION PROBLEMS

The first requirement that needs to be satisfied for training deep learning models to predict solutions to MILP problems is to be able to feed instances of MILP problems to such models. For this, it is necessary to convert an instance to a numerical format that the model can handle.

Naturally, an instance can be specified by a tuple $(\boldsymbol{c}, \boldsymbol{b}, A, n)$, as discussed in Sec. 1.2, which could be vectorized and input to a vanilla NN. This form of embedding, which is going to be referred to as *naïve* embedding, has several shortcomings. First, it does not represent the *symmetries* of the formulation, which are operations applied to the parameters that do not alter its solutions. For example, changing the order of the constraints, which can be seen as permutations of rows of $[A \mid \boldsymbol{b}]$, does not affect in any way the feasible space nor the objectives associated to feasible solutions, but generates different embeddings.

Furthermore, the naïve embedding can easily be an over-parametrization of the instance distribution, which often are sampled from a lower-dimensional space. For example, take the MILP formulation of the TSP by Miller, Tucker and Zemlin (1960),

$$\min_{\boldsymbol{u},\boldsymbol{y}} \quad \sum_{\substack{i,j=0 \\ i \neq j}}^{n} d_{ij} y_{ij}$$

$$
\begin{aligned}
\text{s.t.} \quad & \sum_{\substack{i=0 \\ i \neq j}}^{n} y_{ij} = 1, && j = 1, \ldots, n \\
& \sum_{\substack{j=0 \\ j \neq i}}^{n} y_{ij} = 1, && i = 1, \ldots, n \\
& u_i - u_j + n \cdot y_{ij} \leq n - 1, && i, j = 1, \ldots, n,\ i \neq j \\
& y_{ij} \in \{0, 1\}, && i, j = 0, \ldots, n \\
& \boldsymbol{u} \in \mathbb{R}^n
\end{aligned}
$$

and suppose one wants to solve it for instance $I \in \mathcal{I}$ over the same graph but with varying edge costs d_{ij}. Of course, embedding such instances naïvely would encode all the static parameters of the constraints, i.e., the information that does not change between the instances of interest, which do not carry relevant information for the model.

3.1.1 Feature Engineering

One way to mitigate the shortcomings of the naïve embedding is to extract *features* that well represent the instance with respect to their solutions. This approach is based on the hypothesis that, for a given application, the instances are sampled from a lower-dimensional space, i.e., that there exists a mapping $g^{-1} : X \subseteq \mathbb{R}^d \longrightarrow \mathcal{I}$ that associates features $x \in X$ to instances $I \in \mathcal{I}$, and that d is significantly smaller than the number of parameters (e.g., from the naïve embedding). The mapping is written as the *inverse* of a function g because, in practice, it is not necessary to know g^{-1} to be able to train a deep learning model, only g, i.e., it is only necessary to compute features given instances, and assume that the inverse is possible.

Continuing with the TSP example from above, suppose that the goal is to solve the TSP for a given city (which fixes the graph over which the tours are to be found) but with different traffic conditions and, therefore, different edge costs d_{ij}. The cost vector $\boldsymbol{c}$ (which is a vectorization of the d_{ij} parameters) can be said a feature vector for the instances, but calling this feature engineering would be a controversial statement. However, one could investigate what are the variables that influence the traffic conditions, e.g., hour of the day, day of the week, gas price, weather. Ideally, then, it would be possible to use these variables to define a feature space X, such that a mapping $g^{-1} : X \longrightarrow \mathcal{I}$ exists, and train models that are input with $x \in X$.

Embedding MILP problem instances as feature vectors is an approach suitable for NNs, as they require vector-valued inputs. However, there is an underlying restriction that is a fixed number of features. Although it seems natural, it is not always the case that all instances of a problem have the same number of variables or constraints. If in the TSP example above the underlying graph changes over the instance space, then the instances will have varying numbers of variables and constraints. In the naïve embedding, this translates directly to vectors of varying size, which are not directly suitable for NNs. To generate features that are suitable for NNs even

when the instances have varying size, the feature engineer must be able to translate the process that changes the size of the instances into a fixed number of features, which is not always easy or even feasible.

3.1.2 Graph Embedding

A well-used approach in the intersection between deep learning and combinatorial optimization is to embed MILP problem instances is through bipartite graphs (GASSE et al., 2019; NAIR et al., 2021; DING et al., 2020; KHALIL; MORRIS; LODI, 2022; HAN et al., 2023). Any instance of an LP problem can be represented as a weighted bipartite graph. Consider the problem

$$\begin{aligned} \max_{\boldsymbol{y}} \quad & \boldsymbol{c}^T \boldsymbol{y} \\ \text{s.t.:} \quad & A\boldsymbol{y} \leq \boldsymbol{b}, \end{aligned} \tag{3.1}$$

where $\boldsymbol{y} \in Y \subseteq \mathbb{R}^n$ and $\boldsymbol{b} \in \mathbb{R}^m$. It is possible to build a bipartite graph $G = (V_{\text{var}} \cup V_{\text{con}}, E)$, in which $v_{\text{con},i} \in V_{\text{con}}$ is the node associated to the i-th constraint, $v_{\text{var},j} \in V_{\text{var}}$ is the node associated to y_j, and $E = \{(v_{\text{con},i}, v_{\text{var},j}) : A_{i,j} \neq 0\}$. Furthermore, a weight function $w : V_{\text{var}} \cup V_{\text{con}} \cup E \longrightarrow \mathbb{R}$ such that $w(v_{\text{var},j}) = c_j$, $w(v_{\text{con},i}) = b_i$, and $w(e_{i,j} = (v_{\text{con},i}, v_{\text{var},j})) = A_{i,j}$, renders the weighted graph (G, w) a complete representation of any instance of the LP, i.e., the original LP instance can be reconstructed using solely the information in such weighted graph.

The extension to MILP problems requires solely the distinction between continuous and integer variables. This can be done, for example, by extending the weight function to a vector-valued function such that $w(v_{\text{var},j}) = (c_j, 0)$ if the j-th variable is continuous or $w(v_{\text{var},j}) = (c_j, 1)$ if x_j is an integer variable. In practice, however, the graph fed to a GNN is usually "weighted" with feature vectors $\boldsymbol{h}_v^{(0)}, \forall v \in V$ of arbitrary size, as seen in Sec. 2.2.2. In other words, the information contained in the weights (feature vectors) provided to the network is a design choice: it can contain the weights described above, but many other features might also help the model learn the graph-related task (see, for example, Gasse et al. (2019) and Nair et al. (2021)).

The graph embedding is perfectly suitable for GNNs. In comparison to the feature engineering approach, the graph embedding requires no effort from an human expert, and provides an effective result in terms of representation power and scalability. First, because the resulting graph contains all of the information present in the instance while being invariant to constraint and variable permutations. On top of that, the size of the GNN (number of parameters) does no scale with the size of the graph, but solely with the number of weights (dimension of the feature vector) associated to each node.

3.2 TRAINING UNDER SUPERVISION

Ideally, a solution prediction model is capable of predicting the *bias* of the integer variables in the optimal solutions of a given instance of an MILP problem (KHALIL; MORRIS; LODI, 2022). Intuitively, the bias of a variable towards a value indicates how likely that variable is to assume that value in an optimal solution. As the problem is linear over the continuous variables, their optimal value value can be determined in polynomial time given an optimal assignment for the integer variables, as the resulting problem is an LP. Therefore, the focus of solution prediction models for MILP is usually the integer variables.

More precisely, let $I \in \mathcal{I}$ be an instance of an MILP problem as in (MILP). The *bias* of variable y_j towards value $k \in \mathbb{Z}$ in the optimal solution will be denoted $p(y_j^* = k|I)$, in an allusion to its *probability* of taking said value in an optimal solution $\boldsymbol{y}^*$, which also implies that it is expected that $\sum_k p(y_j^* = k|I) = 1, \forall j$. Therefore, given an embedding $x \in \mathcal{X}$[1] associated to an instance of an optimization problem, a solution prediction deep learning model $f_\theta : \mathcal{X} \longrightarrow \mathcal{P}$ will ideally be such that, for $\hat{\boldsymbol{p}} = f_\theta(x)$, it is expected that, $\forall j$, $\hat{p}_{j,k} \approx p(y_j^* = k|I)$.

Given an embedding function and a suitable deep learning model (e.g., naïve embedding or engineered features and a NN, or graph embedding and GNN), the usual training algorithms for supervised learning apply. In other words, following a match between instance embedding and model architectures, the algorithmic approach presented in Sec. 2.1 applies. Therefore, the dataset required for training is composed of embeddings of instances associated to optimal solutions. Let $\boldsymbol{y}_I^*$ denote an optimal solution for instance $I \in \mathcal{I}$ of the MILP problem at hand, and let $g : \mathcal{I} \longrightarrow \mathcal{X}$ be a suitable embedding function. Then, the dataset necessary for training can be written as a set

$$\mathcal{D} = \{(x_I, \boldsymbol{y}_I^*) : I \in \mathcal{I}, x_I = g(I), \text{ and } \boldsymbol{y}_I^* \text{ is an optimal solution of } I\}.$$

Given such dataset, the training algorithm can be defined by picking any loss function that penalizes the distance between the predicted bias and the actual value. For example, following a maximum likelihood estimation approach (GOODFELLOW; VINYALS; SAXE, 2015) for a problem solely with binary variables, the binary cross-entropy loss can be applied to a model $f_\theta : \mathcal{X} \longrightarrow [0, 1]^n$ such that

$$\ell(\boldsymbol{y}, \hat{\boldsymbol{p}}) = \sum_{j=1}^{n} y_j \log \hat{p}_j + (1 - y_j) \log(1 - \hat{p}_j). \tag{3.2}$$

Note that, because there are only binary variables, the model is designed with output only for the bias towards $k = 1$, as $\hat{p}_j \approx p(y_j^* = 1|I) \iff 1 - \hat{p}_j \approx p(y_j^* = 0|I)$.

[1] Here, $\mathcal{X}$ is used to denote a more general embedding space, that can that of feature vectors or of graph embeddings.

3.2.1 Multiple Targets

Instead of approximating the bias of the optimal solution, Nair et al. (2021) proposed to approximate the bias of the *near*-optimal solutions. Intuitively, this approach provides the model with more information on the feasible region of the problem, and empirical results suggest that it has improved performance in the construction of heuristics (KHALIL; MORRIS; LODI, 2022; HAN et al., 2023). A proper definition of what will be referred to as a *multiple targets* training follows.

Given an instance of an optimization problem $I \in \mathcal{I}$[2], let

$$Y_\varepsilon = \left\{ \boldsymbol{y} \in Y : \boldsymbol{c}^T \boldsymbol{y} \leq (1+\varepsilon) \boldsymbol{c}^T \boldsymbol{y}_I^* \right\}$$

be the set of ε-optimal solutions, that is, the set of feasible solutions that are within ε distance (in relative terms of the cost) of the optimal solution $\boldsymbol{y}^*$. The multiple-targets approach implies that the output of a solution prediction deep learning model f_θ approximates the bias of the variables in the solutions in a set Y_ε, i.e., $\hat{p}_{j,k} \approx p(y_j = k | \boldsymbol{y} \in Y_\varepsilon)$. For that, Nair et al. (2021) propose to weight a loss function such as (3.2) by the cost associated to each solution in Y_ε. Therefore, the dataset $\mathcal{D}$ necessary for training will contain pairs of the form (x, Y_ε), where x is an embedding of an instance of an MILP problem, and Y_ε is a set of ε-optimal solutions of the same instance. In other words, the cost function becomes

$$\mathcal{L}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(x, Y_\varepsilon) \in \mathcal{D}} \sum_{\boldsymbol{y} \in Y_\varepsilon} \frac{e^{-\boldsymbol{c}^T \boldsymbol{y}}}{\sum_{\boldsymbol{y}' \in Y_\varepsilon} e^{-\boldsymbol{c}^T \boldsymbol{y}'}} \ell(\boldsymbol{y}, f_\theta(x)).$$

Note that multiple feasible solutions are taken into consideration for each instance of the MILP problem in our dataset, hence the name “multiple targets.”

3.3 LEARNING-BASED HEURISTICS

Given a properly trained solution prediction deep learning model, there are many ways to generate a primal heuristic. The most naïve heuristic, perhaps, would be to take the model’s output directly. However, it is very unlikely that, even for models trained extensively on large datasets, the output will have a high feasibility rate on realistic problems of a reasonable size. That is so because the characteristics of the feasible region usually make so that a single deviation (i.e., a single bit flip) can render an optimal solution infeasible. Therefore, the probabilistic nature of deep learning makes it very difficult to achieve a reasonable feasibility rate on problems with many variables, as ensuring output constraints on deep learning models is a difficult challenge (see, e.g., Chamon (2020)).

An alternative to balance the speed of solution prediction models with better feasibility (and optimality) expectations is to explore matheuristics (see Sec. 1.3.2.1). In this section, three structures of matheuristics that use solution prediction models are presented.

[2] In the following, the reference to I is omitted to ease the notation, but the definitions are specific to an instance.

3.3.1 Warm-starting MILP Solvers

A straightforward approach to is to use the output of a solution prediction model to provide (partial) solutions to a solver, warm-starting the optimization. For example, the SCIP solver (BESTUZHEVA et al., 2021) accepts complete and partial solutions, which are used to guide the inner heuristics of the optimization algorithm. We use the output of the model to determine which variables will compose the partial solution provided to the solver based on the *confidence* of the model's prediction. Such confidence is based on how strong the predicted bias is, i.e., the probability of the predicted value. In other words, the closer the model's output $\hat{p}_{j,k}$ is to 1, the more confident the model is that the y_j variable should take value k in an optimal solution.

Formally, given an instance $I \in \mathcal{I}$ for which $x \in X$ is an adequate embedding, we have $\hat{\boldsymbol{p}} = f_\theta(x)$ the output of the model. The model's predicted solution is a vector $\hat{\boldsymbol{y}}$ such that

$$\hat{y}_j = \arg\max_k \hat{p}_{j,k},\ j = 1, \ldots, n.$$

A partial solution based on the model's confidence is a set

$$\overline{\boldsymbol{y}}^{(N)} = \left\{(j, \hat{y}_j) : \hat{y}_j = \arg\max_k \hat{p}_{j,k}\right\} \tag{3.3}$$

with the N most confident predictions of the model. More precisely, $\overline{\boldsymbol{y}}^{(N)}$ has size N, and for any $(j_1, \hat{y}_{j_1}) \in \overline{\boldsymbol{y}}^{(N)}$ and any $(j_2, \hat{y}_{j_2}) \notin \overline{\boldsymbol{y}}^{(N)}$, then $\hat{p}_{j_1,\hat{y}_{j_1}} \geq \hat{p}_{j_2,\hat{y}_{j_2}}$.

The diagram of Figure 7 illustrates the building blocks of a warm-start based on a solution prediction deep learning model.

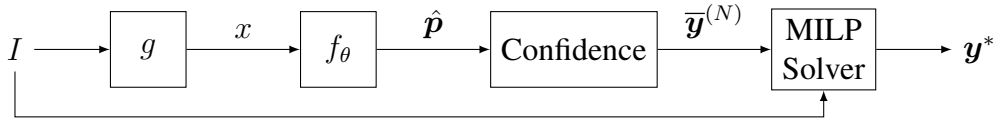


Figure 7 – Warm-starting an MILP solver with the output of a solution prediction deep learning model. In the diagram, I is an instance of an MILP problem for which the model was trained. g is an adequate embedding function. The "Confidence" block indicates the construction of a partial solution as described in Equation (3.3).

Note that warm-starting an MILP solver by itself does not configure a heuristic solution, as the optimality guarantees are maintained. More precisely, warm starting a solver will only (potentially) change the order in which the nodes of the branch-and-bound tree are explored, e.g., by influencing branching decisions. Because of that, it also maintains the optimality (and feasibility) guarantees of the MILP solver used, even if the solution prediction model provides an infeasible (partial) solution. However, under limited time, those guarantees are lost, as the solver may be interrupted before finding even a feasible solution. In fact, even without the warm-starting approach, the simplest matheuristic is to interrupt an MILP solver after a fixed amount of time.

3.3.2 Early-fixing Variable Assignments

Beyond merely indicating to the solver the partial solution that the deep learning model provides, it is possible to constrain the problem with that partial solution. This early-fixing approach, also called neural diving by Nair et al. (2021), can be interpreted, given an instance $I \in \mathcal{I}$ and a partial solution as in Equation (3.3), as the addition of constraints

$$y_j = \hat{y}_j, \forall (j, \hat{y}_j) \in \overline{\boldsymbol{y}}^{(N)} \tag{3.4}$$

to the optimization problem. Because such constraints limit those variables to assuming a single value, which is effectively the same as removing them from the poll of decision variables and treating them as parameters of the problem, the branch-and-bound tree gets significantly pruned. The diagram of Figure 8 illustrates this process.

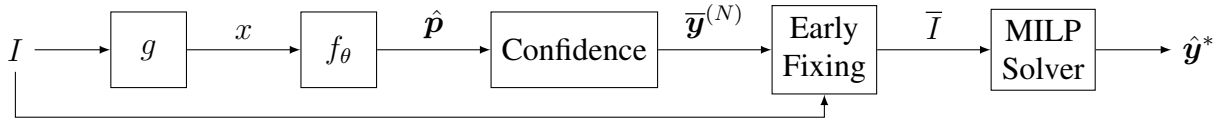


Figure 8 – Early-fixing integer variables based on the output of a solution prediction deep learning model. In the diagram, I is an instance of an MILP problem for which the model was trained. g is an adequate embedding function. The "Confidence" block indicates the construction of a partial solution as described in Equation (3.3). The "Early Fixing" block indicates the addition of constraints (3.4), resulting in the early-fixed instance $\overline{I}$. The solver output is written $\hat{\boldsymbol{y}}^*$ to indicate it as being a heuristic solution.

Naturally, the addition of the early-fixed constraints implies that there are no guarantees that the solution found is optimal. In fact, the resulting problem instance (denoted $\overline{I}$, as in Figure 8) might be infeasible, if the added constraints come from an infeasible assignment. However, by adjusting the size of the partial solution N, it is possible to indirectly adjust the size of the resulting branch-and-bound tree, as more variables in the partial solution, implies in more fixing constraints, which results in a smaller tree. In fact, $N \to n$ reduces the early-fixing matheuristic to the naïve use of the solution predicted by the deep learning model. Furthermore, it is easy to see that, instead of a fixed N, one can build partial solutions by choosing $N = n - n'$ (where $n' \ll n$ is a fixed value), such that the resulting problem instance always has n' variables and, thus, can be solved in a tractable manner.

3.3.3 Trust-region

Instead of strictly fixing the variables based on the model's output, Han et al. (2023) have proposed to allow a small deviation from that value. In other words, the solution prediction model's output is used to define a *trust region* in which an MILP solver can search for the optimal solution. Instead of constraints like in Equation (3.4), the instance is modified with the addition

of constraints[3] of the form

$$\sum_{(j,\hat{y}_j)\in\overline{\boldsymbol{y}}^{(N)}} |y_j - \hat{y}_j| \leq \Delta, \tag{3.5}$$

where $\Delta \in \mathbb{R}_+$ defines the size of the trust region. Note how the above equation limits the space of feasible solutions to a neighborhood of the partial solution derived from the model's output. The diagram in Figure 9 illustrates the trust region heuristic approach.

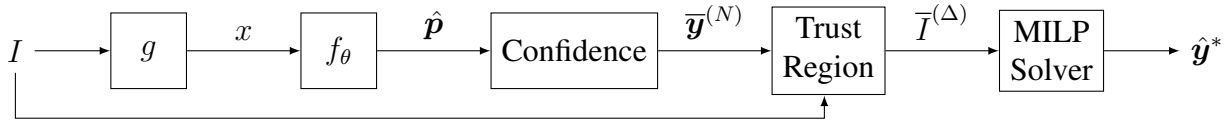


Figure 9 – Solving an instance of an MILP problem within a trust region based on the output of a solution prediction deep learning model. In the diagram, I is an instance of an MILP problem for which the model was trained. g is an adequate embedding function. The "Confidence" block indicates the construction of a partial solution as described in Equation (3.3). The "Trust Region" block indicates the addition of the constraint (3.5), resulting in the instance $\overline{I}^{(\Delta)}$ with limited solution space. The solver output is written $\hat{\boldsymbol{y}}^*$ to indicate it as being a heuristic solution.

It is easy to see that picking $\Delta = 0$ results in the early-fixing approach. A distinguishing feature of the trust region approach is that the parameter Δ can be adjusted to turn an infeasible instance into a feasible one, or, perhaps, to include a better solution in the feasible region. However, no optimality nor feasibility guarantees can be provided by this approach.

[3] Note that Equation (**??**) can be implemented in an MILP as multiple linear constraints.

4 OFFLINE NANOSATELLITE TASK SCHEDULING

This chapter delves into the application area of the experiments conducted for this dissertation, specifically focusing on the Offline Nanosatellite Task Scheduling (ONTS) problem. Over the past decade, nanosatellites have gained significant attention from both industry and academia, primarily due to their cost-effective development and launch processes (SHIROMA et al., 2011; LUCIA et al., 2021; NAGEL; NOVO; KAMPEL, 2020; SAEED et al., 2020). Despite these advantages, the limited computational and energy resources of nanosatellites present substantial challenges in mission planning. Effective task scheduling is essential to optimize resource utilization, enhance data quality, and ensure mission success, thereby securing a return on investment.

The ONTS problem is critical for the efficient development, deployment, and operation of nanosatellites. From launch to disposal, the ONTS problem must be solved repeatedly. At every communication window, new schedules must be generated and deployed, and the optimal schedule is determined by an iterative procedure, exploring different sets of tasks to be performed in the schedule's timespan. Given a nanosatellite and a collection of tasks, determining the schedule with maximum Quality of Service (QoS) is a combinatorial problem. Mathematical formulations have been proposed for this problem, from Integer Programming (IP) (RIGO et al., 2021b) to Mixed Integer Linear Programming (MILP) (RIGO et al., 2021a; SEMAN et al., 2022) and Continuous-Time techniques (CAMPONOGARA et al., 2022). However, the NP-hard nature of the problem renders multiple executions of the optimization algorithms (e.g., for different task configurations) within the timespan of a communication window an efficiency challenge.

The following section will present the problem statement with a description of the factors that are taken into consideration. Then, the MILP formulation of the problem, proposed by Rigo et al. (2021a), is presented, which will be the basis for the experiments presented in Part III.

4.1 PROBLEM STATEMENT

Nanosatellite scheduling problems involve making decisions regarding the start and finish times of each task. These tasks often require periodic execution and must be scheduled during specific moments along the satellite's orbit. In addition to temporal constraints, energy availability throughout the orbit is a crucial resource that must be taken into account. Figure 10 illustrates an example of optimal scheduling, where each job is represented by a different color, and the activation and deactivation of tasks are depicted as steps in the signal sequence.

Effective scheduling must incorporate energy management to ensure that tasks do not

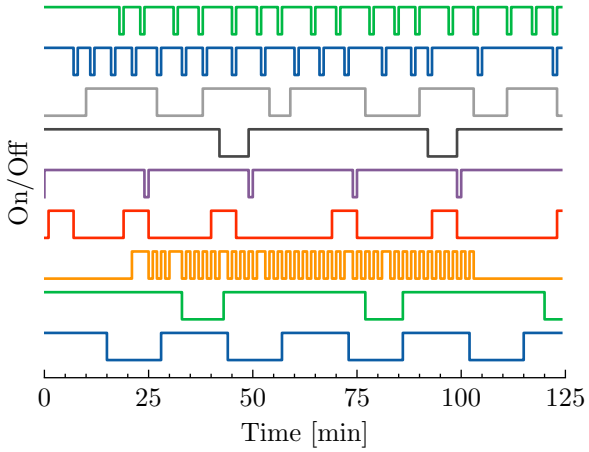


Figure 10 – Illustration of an optimum schedule for 9 tasks on a horizon of 125 time steps. Each color represents the executions of a different task.

consume more energy than the system can provide, thereby preventing the battery from depleting before the mission concludes. Energy management is particularly challenging because the nanosatellite relies on solar panels for power. The energy availability is influenced by the nanosatellite's attitude, which affects the orientation of the solar panels, and its trajectory relative to Earth's shadow, as depicted in Figure 11. On top of that, the shared energy resources steps up the problem complexity, as each task's activation must be determined while taking into consideration the other tasks.

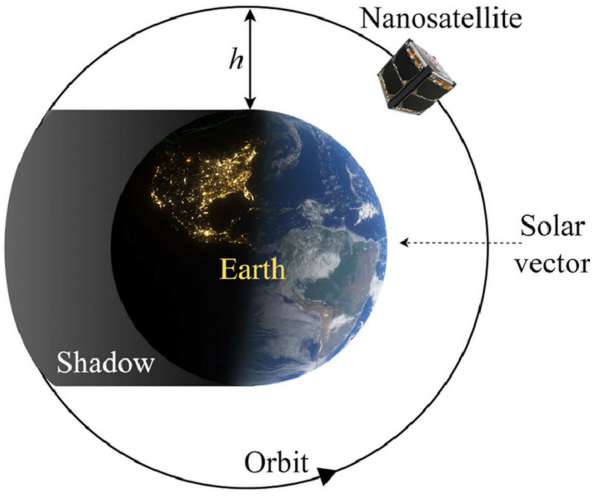


Figure 11 – Illustration of a nanosatellite's orbit around Earth. Image from Rigo et al. (2022).

Instances of the ONTS problem are solved during the mission design phase, before launching the nanosatellite into orbit, and during the mission execution, at every communication window. During mission design, the instances are solved to evaluate the impact of the design choices in the nanosatellite capabilities. During mission execution, the schedule is updated whenever possible to account for unexpected events (e.g., possible execution failures, hardware faults, unexpected battery drainage) or new requirements (e.g., new tasks, software updates).

More precisely, at every communication window, nanosatellite information (such as battery level, execution log, task results) is downloaded, the parameters of the ONTS problem are updated, a new schedule is generated, and the instructions are uploaded to the nanosatellite.

4.2 MILP FORMULATION

The formulation presented here was proposed by Rigo et al. (2021b) and the reader is advised to refer to the original work for further details.

Given a set $\mathcal{J} = \{1, ..., J\}$ of tasks (or jobs), and a set $\mathcal{T} = \{1, ..., T\}$ of time units of the scheduling horizon, let variables $x_{j,t}$ represent the allocation of task j at time t, $\forall j \in \mathcal{J}, \forall t \in \mathcal{T}$. Naturally, each $x_{j,t}$ is a binary variable, in which value 1 indicates that task j is scheduled to be in execution at time t. When convenient, these variables will be represented as a vector

$$\boldsymbol{x} = (x_{j,t})_{\substack{j=1,...,J \\ t=1,...,T}},$$

which is a notation also used for other variables and parameters.

Every task j has an associated priority $u_j > 0$, such that the mission's QoS is defined as

$$\mathrm{QoS}(\boldsymbol{x}; \boldsymbol{u}) = \sum_{j=1}^{J} \sum_{t=1}^{T} u_j x_{j,t}. \tag{4.1}$$

As mentioned previously, the goal of the optimization problem is to find a schedule that maximizes the QoS.

Auxiliary variables $\phi_{j,t}$ are defined for every $j \in \mathcal{J}$ and $t \in \mathcal{T}$, which represent the startup of the task, i.e., $\phi_{j,t} = 1$ indicates that task j is not running at time $t-1$ but its execution is started at time t. The behavior of these auxiliary variables is ensured by

$$\begin{aligned}
&\phi_{j,t} \geq x_{j,t}, && \forall j \in \mathcal{J},\, t = 1 \\
&\phi_{j,t} \geq x_{j,t} - x_{j,(t-1)}, && \forall j \in \mathcal{J},\, \forall t \in \mathcal{T} : t > 1 \\
&\phi_{j,t} \leq x_{j,t}, && \forall j \in \mathcal{J},\, \forall t \in \mathcal{T} \\
&\phi_{j,t} \leq 2 - x_{j,t} - x_{j,(t-1)}, && \forall j \in \mathcal{J},\, \forall t \in \mathcal{T} : t > 1.
\end{aligned} \tag{4.2}$$

The multiple requirements of each task are ensured by a series of constraints. Each task j may run only during a specified time window delimited by $w_j^{\min}$ and $w_j^{\max}$, which is imposed by

$$\begin{aligned}
&\sum_{t=1}^{w_j^{\min}} x_{j,t} = 0, && \forall j \in \mathcal{J} \\
&\sum_{t=w_j^{\max}+1}^{T} x_{j,t} = 0, && \forall j \in \mathcal{J}.
\end{aligned} \tag{4.3}$$

Such constraints can be used to force a task to be executed when the nanosatellite is passing over a predetermined region.

To limit continuous executions, a task j is constrained to run without interruption for least $t_j^{\min}$, and at most $t_j^{\max}$ time steps, which is imposed by

$$\begin{aligned} \sum_{l=t}^{t+t_j^{\min}-1} x_{j,l} &\geq t_j^{\min}\phi_{j,t}, \quad \forall t \in \{1,...,T-t_j^{\min}+1\}, \forall j \in \mathcal{J} \\ \sum_{l=t}^{t+t_j^{\max}} x_{j,l} &\leq t_j^{\max}, \qquad\qquad \forall t \in \{1,...,T-t_j^{\max}\}, \forall j \in \mathcal{J}. \end{aligned} \tag{4.4}$$

Complementary, and having in mind that the end of schedule is not the end of the nanosatellite life, the addition of constraint

$$\sum_{l=t}^{T} x_{j,l} \geq (T-t+1)\phi_{j,t}, \quad \forall t \in \{T-t_j^{\min}+2,...,T\}, \forall j \in \mathcal{J} \tag{4.5}$$

enables the start of the execution of a task close to the end of the schedule horizon, and keep it running until the end.

For a task to be executed periodically, at least every $p_j^{\min}$ time steps, and at most every $p_j^{\max}$ time steps, the following constraints are added:

$$\begin{aligned} \sum_{l=t}^{t+p_j^{\min}-1} \phi_{j,l} &\leq 1, \quad \forall t \in \{1,...,T-p_j^{\min}+1\}, \forall j \in \mathcal{J} \\ \sum_{l=t}^{t+p_j^{\max}-1} \phi_{j,l} &\geq 1, \quad \forall t \in \{1,...,T-p_j^{\max}+1\}, \forall j \in \mathcal{J}. \end{aligned} \tag{4.6}$$

On top of that, a task may be required to run multiple times over the planning horizon. The constraints

$$\begin{aligned} \sum_{t=1}^{T} \phi_{j,t} &\geq y_j^{\min}, \quad \forall j \in \mathcal{J} \\ \sum_{t=1}^{T} \phi_{j,t} &\leq y_j^{\max}, \quad \forall j \in \mathcal{J} \end{aligned} \tag{4.7}$$

are added to ensure at least $y_j^{\min}$ and at most $y_j^{\max}$ runs are performed for task j.

The energy-management restrictions are ensured through multiple constraints. Given r_t, the power available from the solar panels at each time t, q_j, the power required by each task j, and $\gamma\ V_b$, the maximum power the battery can provide, then

$$\sum_{j=1}^{J} q_j x_{j,t} \leq r_t + \gamma\ V_b, \quad \forall t \in \mathcal{T} \tag{4.8}$$

limits the power consumption to realistic levels. Auxiliary variables b_t and SoC_t represent, resp., the exceeding power and the State of Charge (SoC) over each time $t \in \mathcal{T}$. Given Q, the battery capacity, and e, the discharge efficiency, the exceeding power is ensured by

$$b_t = r_t - \sum_{j \in \mathcal{J}} q_j x_{j,t}, \quad \forall t \in \mathcal{T}, \tag{4.9}$$

while the SoC is ensured by

$$\begin{aligned} &\text{SoC}_{t+1} = \text{SoC}_t + \frac{b_t\, e}{60\, Q\, V_b}, && \forall t \in \mathcal{T} \\ &\text{SoC}_t \leq 1, && \forall t \in \mathcal{T} \\ &\text{SoC}_t \geq \rho, && \forall t \in \mathcal{T}, \end{aligned} \tag{4.10}$$

where ρ is the allowed lower limit for the battery, which is usually greater than zero for safety purposes.

Finally, the ONTS problem is formulated as an MILP

$$\begin{aligned} \max_{\boldsymbol{x},\boldsymbol{\phi},\text{SoC},\boldsymbol{b}} \quad & (4.1) \\ \text{s.t.} \quad & (4.2\text{–}4.10) \\ & x_{j,t}, \phi_{j,t} \in \{0,1\}, \quad \forall j \in \mathcal{J}, t \in \mathcal{T}. \end{aligned} \tag{ONTS}$$

Note that the constraints (4.9) and (4.10) imply that the continuous variables $\boldsymbol{b}$ and SoC are uniquely determined by a given assignment for the binary variables $\boldsymbol{x}$ and $\boldsymbol{\phi}$. Therefore, the problem can be reduced to finding an assignment $\boldsymbol{y} = (\boldsymbol{x}, \boldsymbol{\phi}) \in \{0,1\}^n$, where $n = 2JT$.

Let $\mathcal{I}$ be the space of all MILP problems of the form (ONTS). Any instance $I \in \mathcal{I}$ is parameterized by $\pi_I = \big(\boldsymbol{u}, \boldsymbol{q}, \boldsymbol{y}^{\min}, \boldsymbol{y}^{\max}, \boldsymbol{t}^{\min}, \boldsymbol{t}^{\max}, \boldsymbol{p}^{\min}, \boldsymbol{p}^{\max}, \boldsymbol{w}^{\min}, \boldsymbol{w}^{\max}, \boldsymbol{r}, \rho, e, Q, \gamma, V_b\big)$, and, implicitly, by the number of tasks J and the number of time units T. Let $\Pi^{J,T}$ denote the space of parameter vectors as above, such that any instance $I \in \mathcal{I}$ can be uniquely determined by a parameter vector π_I (given adequate J and T).

5 EVALUATION OF PRIMAL HEURISTICS

As discussed in Chapter 1, Section 1.3.2, a primal heuristic abides from optimality guarantees to focus on a trade-off between solution quality and computational cost (speed). Consequently, two primal heuristics for MILP problems can be compared with respect to how fast they provide solutions and how good they are, and whether the solutions are feasible or not. In this chapter, multiple metrics are presented to cover the two perspectives in which a heuristic can be said superior to another.

The objective value and the time taken to provide a solution are natural evaluation metrics. However, they have significant shortcomings. The value of the cost function (supposing a minimization problem) is hard to interpret without bounds. For example, it is difficult to judge how better one heuristic approach is with respect to the other just by knowing that the first provided a solution with a cost of 10, while the second provided a solution with a cost of 15. If the optimal solution costs -1000 and a trivial solution costs 20, then it can be said that both heuristics performed poorly. On the other hand, if the optimal solution has a cost of 9 and a trivial solution has cost 15, then it can be said that the first heuristic performed much better than the second.

Therefore, to make a fair judgment about the solution quality of a set of heuristics being evaluated, one needs to know both the cost of the solutions as well as upper and lower bounds for the problem. Beyond the difficulty of determining such bounds, having a multidimensional metric for judging solution quality can make it more challenging to compare the performance across different optimization problems and even across different instances of the same problem. An alternative to summarizing all these values is to compute the cost of the heuristic solution normalized between the optimal cost (0 %) and the cost of the trivial solution (100 %). Let $\hat{\boldsymbol{y}}$ be a heuristic solution, $\boldsymbol{y}^*$ be the optimal solution, and $\overline{\boldsymbol{y}}$ be a trivial solution for an instance of an optimization problem as in (MILP). Then, *relative cost* of $\hat{\boldsymbol{y}}$ can be defined as

$$\text{RelCost}(\hat{\boldsymbol{y}}) = \frac{\boldsymbol{c}^T\hat{\boldsymbol{y}} - \boldsymbol{c}^T\boldsymbol{y}^*}{\boldsymbol{c}^T\overline{\boldsymbol{y}} - \boldsymbol{c}^T\boldsymbol{y}^*}. \tag{5.1}$$

Similarly, if the problem of interest is a maximization problem, then one can define the *relative objective* of a candidate solution $\hat{\boldsymbol{y}}$ as

$$\text{RelObj}(\hat{\boldsymbol{y}}) = \frac{\boldsymbol{c}^T\hat{\boldsymbol{y}} - \boldsymbol{c}^T\overline{\boldsymbol{y}}}{\boldsymbol{c}^T\boldsymbol{y}^* - \boldsymbol{c}^T\overline{\boldsymbol{y}}}. \tag{5.2}$$

Certain conditions must be met to evaluate an approach's efficiency, ensuring that the time taken to compute a solution (runtime) is meaningful and fair. One way to do so is to ensure that the computational resources are fairly available to all approaches. Usually, this implies in allowing all approaches to have plain access to a common hardware, i.e., that there are no other processes using the resources during the time of execution.

Still, there are a few caveats in this performance metric, of which parallelization abilities are probably the most prominent. Comparing single-process with multi-process approaches is quite difficult, as a process that can compute its results in parallel uses more resources in the same runtime. Furthermore, projecting empirical results to different setups is challenging when the processes being evaluated use parallel computation. In this work, it is considered that parallelization is commonly supported in modern hardware configurations. Thus, the runtime is computed as the user-perceived real time (or wall time) for the heuristic to provide a solution given an instance of an optimization problem, regardless of whether parallelization was used or not.

Part III

Experiments and Results

6 EXPERIMENTS

To address the objective of this dissertation, experiments are conducted to evaluate learning-based primal heuristics for Mixed-Integer Linear Programming (MILP). The ONTS problem (see Chap. 4) serves as a realistic application to benchmark the selected techniques.

As discussed in Section 4.1, during mission execution (with the nanosatellite in orbit), a new schedule must be generated during the communication window. This involves optimizing multiple instances of the ONTS problem, given varying sets of tasks and updated nanosatellite information. Each set of tasks is evaluated based on the resulting schedule, in an iterative process of including new tasks until scheduling becomes infeasible. Therefore, during the communication window, quickly finding a good solution to a problem instance is more crucial than finding an optimal solution. In other words, an efficient heuristic is crucial to allow for more iterations, which leads to a better set of tasks scheduled for execution.

The remaining of this chapter details the development and the experiments with the proposed learning-based heuristics for the ONTS problem. This includes data acquisition, solution prediction model architecture, model training, and experiment setup. Furthermore, the performance of the proposed learning-based heuristics is assessed on realistic instances of the ONTS problem.

6.1 DATA

High-quality data is necessary both to train solution prediction models and to evaluate the proposed learning-based heuristics for the ONTS problem. The datasets for both training and evaluation must be composed of instance-solution pairs, as discussed in Sec. 3.2. The quality of these instance-solution pairs is measured through their faithfulness, both the instance with respect to the true data distribution, and the solution with respect to the optimal.

6.1.1 Instance space: the FloripaSat I mission

The instance space is defined from the parameters of the FloripaSat-I mission (MARCELINO et al., 2020). Their nanosatellite is in orbit at an altitude of 628 kilometers and an orbital period of 97.2 minutes. The planning horizon is fixed at $T = 125$ time slots, with one slot per minute, to account for a continuous scheduling, allowing for task executions that extend the communication window. Any instance $I \in \mathcal{I}$ has either 9, 13, 18, 20, 22, or 24 tasks.

Once the orbit of the FloripaSat-I is stable and its received solar flux is constant, the power input vector $\boldsymbol{r}$ can be calculated deterministically from solar irradiance measurements as in Filho et al. (2020). Two years of solar irradiance data are used as a basis for the power input

vectors of the instances in the instance space. The set R is used to denote all possible values of $\boldsymbol{r}$ from the historical data. The other battery-related parameters (see Sec. 4.2) are fixed as

$$\begin{aligned} e &= 0.9 \\ Q &= 5 \\ \gamma &= 5 \\ V_b &= 3.6 \\ \rho &= 0.0 \end{aligned}$$

The remaining parameters are constrained to ranges that match previous works in the area (RIGO et al., 2022; SEMAN et al., 2022; RIGO et al., 2021b). Therefore, following the notation established in Sec. 4.2, the parameter space is defined as

$$\Pi = \bigcup_{\substack{J \in \{9,13,18,20,22,24\} \\ T \in \{125\}}} \Pi^{J,T}, \tag{6.1}$$

where each $\Pi^{J,T}$ is a set of a parameter vectors

$$\pi_I = \left(\boldsymbol{u}, \boldsymbol{q}, \boldsymbol{y}^{\min}, \boldsymbol{y}^{\max}, \boldsymbol{t}^{\min}, \boldsymbol{t}^{\max}, \boldsymbol{p}^{\min}, \boldsymbol{p}^{\max}, \boldsymbol{w}^{\min}, \boldsymbol{w}^{\max}, \boldsymbol{r}, \rho, e, Q, \gamma, V_b\right) \in \Pi^{J,T}$$

such that

$$\left.\begin{aligned} & u_j \in [1, J] \\ & q_j \in [0.3, 2.5] \\ & y_j^{\min} \in [1, \lceil T/45 \rceil] \\ & y_j^{\max} \in [y_j^{\min}, \lceil T/15 \rceil] \\ & t_j^{\min} \in [1, \lceil T/10 \rceil] \\ & t_j^{\max} \in [t_j^{\min}, \lceil T/4 \rceil] \\ & p_j^{\min} \in [t_j^{\min}, \lceil T/4 \rceil] \\ & p_j^{\max} \in [p_j^{\min}, T] \\ & w_j^{\min} \in [0, \lceil T/5 \rceil] \\ & w_j^{\max} \in [\lfloor T - \lceil T/5 \rceil \rfloor, T] \end{aligned}\right\} \forall j = 1, \ldots, J$$

$$\boldsymbol{r} \in R,\ e = 0.9,\ Q = 5,\ \gamma = 5,\ V_b = 3.6,\ \rho = 0.0.$$

Finally, the input space is then defined from the parameter space, such that

$$I \in \mathcal{I} \iff \pi_I \in \Pi.$$

6.1.2 Data acquisition

As historical data is not available for the ONTS problem, the dataset is built from randomly generated instances sampled uniformly from the instance space of the FloripaSat-I

mission. More precisely, the dataset is built with instances drawn uniformly from the instance space defined above.

As the addition of an element to the dataset requires a solution to the ONTS problem, the computational cost of building a large dataset with hard instances is very high. To alleviate this cost, the training set is built solely with instances with fewer tasks, which are, on average, faster to solve than instances with many tasks. However, the instances of interest are those with plenty of tasks, which are harder to solve in practice, and, thus, motivate the use of heuristics. Therefore, the validation and test datasets are built from instances with many tasks, which are, on average, significantly harder to solve. Table 1 details the number of instances by size (number of tasks) in each dataset generated.

Table 1 – Number of instances by size in each dataset. The datasets were generated through Algorithm 1.

	Training	Validation	Test
$J = 9$	200	0	0
$J = 13$	200	0	0
$J = 18$	200	0	0
$J = 20$	0	20	20
$J = 22$	0	20	20
$J = 24$	0	20	20
Total	600	60	60

Distinguishing the size of the instances in each dataset allows for the construction of a large training set, which enables the models to properly learn the problem, while maintaining a challenging evaluation scenario. On top of that, this approach also enables the evaluation of the generalization capabilities of the proposed solution prediction models and the derived learning-based heuristics.

The algorithm to generate the datasets is presented in Algorithm 1. Note that an instance is rejected if no feasible solution is found during the time budget or if the solver proves infeasibility. As the time horizon is fixed, the algorithm is executed once for each number of tasks. Similar to the parameter space definition (6.1), the resulting dataset can be described as

$$\mathcal{D} = \bigcup_{\substack{J \in \{9,13,18,20,22,24\} \\ T \in \{125\}}} \mathcal{D}^{(J,T)}, \tag{6.2}$$

where each $\mathcal{D}^{(J,T)}$ is obtained through Algorithm 1. The algorithm is such that each element of the output dataset $(I, Z_I^\star) \in \mathcal{D}^{(J,T)}$ is composed of a *feasible* instance I sampled uniformly from the parameter space (see Eq. (6.1)) and deemed feasible by an MILP solver. Furthermore, the accompanying set $Z_I^\star$ contains the best solutions found by the solver, which are necessary for training the model with multiple solutions as a target (see Sec. 3.2.1).

For our experiments, the algorithm was executed such that every new instance I was solved using the SCIP solver (BESTUZHEVA et al., 2021) with a limited time budget of 5 minutes. The best 500 solutions of each instance I were recorded, i.e., for every $(I, Z^*) \in \mathcal{D}$, $|Z_I^*| = 500$. Finally, the dataset is divided as $\mathcal{D} = \mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{val}} \cup \mathcal{D}_{\text{test}}$ following Table 1.

Algorithm 1: Dataset generation algorithm. π is the parameter vector and $\Pi^{J,T}$ is the parameter space (see Sec. 4.2), Z_I represents the set of all feasible solutions of instance I, and $Z_I^\star \subset Z_I$ the set of feasible solutions the solver finds. `ONTS` represents a function that takes as input a parameter vector and constructs an instance of the ONTS problem. `Solver` is any MILP solver. Note that the parameters are drawn uniformly from the parameter space.

Data: Time horizon T, number of jobs J, number of instances (final dataset size) n.
Result: Dataset $\mathcal{D}^{(J,T)} = \{(I, Z_I^\star) : Z_I^\star \subset Z_I\}$.
while $|\mathcal{D}^{(J,T)}| < n$ **do**
  $\pi \sim \mathcal{U}\left(\Pi^{J,T}\right)$
  $I \leftarrow \texttt{ONTS}(\pi)$
  $Z_I^\star \leftarrow \texttt{Solver}(I)$
  if $|Z_I^\star| > 0$ **then**
    $\mathcal{D}^{(J,T)}$.add$(I, Z_I^\star)$
  end
end

6.2 SOLUTION PREDICTION MODEL

The instance space of the ONTS problem, as defined in Sec. 6.1.1, imposes specific architectural requirements for solution prediction models. The variable number of tasks over the instances of interest leads to parameter vectors of variable length, which is a natural embedding (feature vector) for the instances. At the same time, the uneven number of tasks in the instance space implies that instances have different number of binary variables. Therefore, models must predict a different number of variable assignments for each instance.

Graph Neural Networks (GNNs) are particularly promising in this context and are considered state-of-the-art for such applications (CAPPART et al., 2022). As discussed in Sec. 2.2.2, GNNs naturally handle variable input and output sizes due to their convolutional nature. On top of that, results shown by Gasse et al. (2019) point that GNNs are capable of generalizing to instances larger than those seen during training, which alleviates data acquisition costs, as discussed in Sec. 6.1.2. Therefore, the deep learning models trained for the ONTS problem are all built with GNNs at their core.

6.2.1 Instance embedding

Because GNNs are at the core of the solution prediction models, the instances are embedded as bipartite graphs, following the approach presented in Sec. 3.1.2. Many authors

have presented different approaches for defining node features when applying GNNs to MILP problems. Khalil, Morris and Lodi (2022) add variable and constraint degrees[1] to the baseline (the one presented in Sec. 3.1.2) embedding. Chen et al. (2022) embeds variables' upper and lower bounds as well as constraint type (inequality or equality). Works that applied GNNs to generate branch-and-bound heuristics (e.g., for branching), such as Ding et al. (2020) and Gasse et al. (2019), have captured features usually available at the nodes of the branch-and-bound tree, directly collecting solver-computed features. As the focus of the present work is on primal heuristics, the features should be solver-independent, i.e., they must be computed prior to the branch-and-bound application.

The resulting features used are based on the set proposed by Han et al. (2023), which extends both Khalil, Morris and Lodi (2022) and Chen et al. (2022) with simple features (solver-independent) that are also present in Gasse et al. (2019). A summary is presented in Table 2. Note that, following the literature, instead of describing all constraints as inequality constraints (i.e., in the normal form), the constraint type (inequality or equality) is informed through the constraint node features, reducing the graph size. Furthermore, this feature design is problem agnostic, i.e., it does not use any information particular to the ONTS problem.

Features of constraint nodes ($\boldsymbol{f}_{v_{\text{con}}}$)	Features of variable nodes ($\boldsymbol{f}_{v_{\text{var}}}$)
Constraint's upper bound ($\boldsymbol{b}$)	Variable's coefficient in the objective ($\boldsymbol{c}$)
Constraint's average coefficient (mean of A_{i*})	Variable's average coefficient in the constraints (mean of A_{*j})
Number of neighbors/non-null coefficients ($\lvert\mathcal{N}(v_{\text{con}})\rvert$)	Number of neighbors/non-null coefficients ($\lvert\mathcal{N}(v_{\text{var}})\rvert$)
Whether it is an equality or an inequality constraint	Largest coefficient in the constraints ($\max(A_{*j})$)
	Smallest coefficient in the constraints ($\min(A_{*j})$)
	Whether it is a continuous or binary variable

Table 2 – Description of node features for the graph embedding of instances of the ONTS problem.

6.2.2 Architecture

The structure of the solution prediction models is illustrated in Fig. 12. The model inputs are the embedding described in the previous section, i.e., the bipartite graph representation of the

[1] A variable's degree is the number of constraints in which it has a nonzero coefficient. On the other hand, a constraint's degree is the number of variables in it with non-zero coefficient.

instance and the sets of feature vectors F_{con} and F_{var}. Each feature vector $\boldsymbol{f}_v$ is encoded into a hidden feature vector $\boldsymbol{h}_v^{(0)}$ with size d by neural networks

$$\begin{aligned} \text{NN}_{\text{var}} : \mathbb{R}^6 &\longrightarrow \mathbb{R}_+^d \\ \boldsymbol{f}_{v_{\text{var}}} &\longmapsto \boldsymbol{h}_{v_{\text{var}}}^{(0)} = \text{NN}_{\text{var}}(\boldsymbol{f}_{v_{\text{var}}}) \end{aligned}$$

and

$$\begin{aligned} \text{NN}_{\text{con}} : \mathbb{R}^4 &\longrightarrow \mathbb{R}_+^d \\ \boldsymbol{f}_{v_{\text{con}}} &\longmapsto \boldsymbol{h}_{v_{\text{con}}}^{(0)} = \text{NN}_{\text{con}}(\boldsymbol{f}_{v_{\text{con}}}), \end{aligned}$$

both with a single layer and ReLU (Rectified Linear Unit) activation (GOODFELLOW; VINYALS; SAXE, 2015). More precisely, the first layer of hidden features of the GNN $H^{(0)} = \left\{ \boldsymbol{h}_v^{(0)} \in \mathbb{R}^d : v \in V \right\}$ is such that

$$\boldsymbol{h}_v^{(0)} = \begin{cases} \text{NN}_{\text{con}}(\boldsymbol{f}_v) & v \in V_{\text{con}} \\ \text{NN}_{\text{var}}(\boldsymbol{f}_v) & v \in V_{\text{var}} \end{cases}.$$

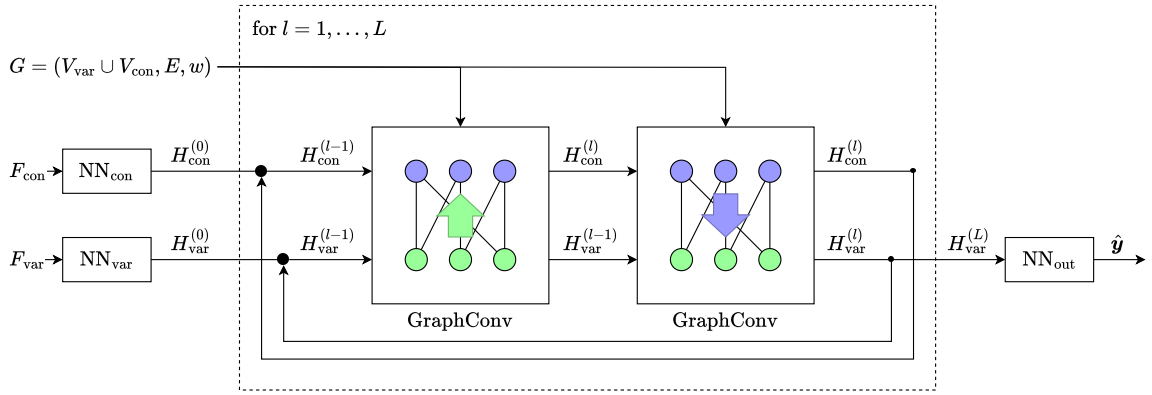


Figure 12 – Architectural components of the solution prediction models trained for the ONTS problem. *GraphConv* indicates the graph convolution operators as described in Sec. 2.2.2. F and H indicate sets of feature vectors. $\text{NN}_\star$ are FCNs applied to each vector in the respective input set.

Each layer of the proposed model performs the graph convolution in the form of two interleaved half-convolutions, as proposed by Gasse et al. (2019) and widely adopted by similar applications (HAN et al., 2023; KHALIL; MORRIS; LODI, 2022; DING et al., 2020). The half-convolutions are applied to each partition of the graph, such that first the hidden feature vectors of the constraint nodes are updated using the hidden features of the variable nodes, and then the hidden features of the variable nodes are updated with the new hidden features of the constraint nodes. These two operations are illustrated by the *GraphConv* blocks in Fig. 12. The choice for the convolution operator is left as a hyperparameter, being either the FCN convolution proposed by Kipf and Welling (2017) or the SAGE model proposed by Hamilton, Ying and Leskovec (2017), as discussed in Sec. 2.2.2.

After L layers, the resulting hidden features are transformed into the predicted bias by another FCN. The 2-layer network

$$\begin{aligned} \mathrm{NN}_{\mathrm{out}} : \mathbb{R}_+^d &\longrightarrow (0,1) \\ \boldsymbol{h}_{v_{\mathrm{var}}}^{(L)} &\longmapsto \hat{p}_{v_{\mathrm{var}}} = \mathrm{NN}_{\mathrm{out}}(\boldsymbol{h}_{v_{\mathrm{var}}}^{(L)}) \end{aligned}$$

combines each variable's output hidden features into a single value. While the first layer has ReLU activation, the last layer has a sigmoid activation, constraining the output to the unit interval, which is adequate to the target range.

6.3 TRAINING

The solution prediction models for the ONTS problem are trained with supervision to approximate the variable bias in the optimal solutions (see Sec. 3.2). This approach is based on having a single (quasi-)optimal solution for each problem instance, and will be referred to in the following as *OS* (Optimal Solution training). Another approach is to exploit multiple near-optimal solutions, as discussed in Sec. 3.2.1, training the model to approximate the variable bias of the solutions near optimal solutions. This latter approach will be referred to as *MS* (Multiple Solution training). The dataset $\mathcal{D}$ built as described in Sec. 6.1.2 is used for both approaches, the only difference being that in OS the best solution is retrieved from Z^*, while in MS all solutions are used along with their objective value.

As the ideal performance metric, i.e., the quality of the predicted solution, is not computable[2] for a vector of variable biases, the Binary Cross-Entropy (BCE) loss is used, as it is the loss of choice of the majority of works with similar applications (NAIR et al., 2021; HAN et al., 2023; KHALIL; MORRIS; LODI, 2022; GASSE et al., 2019). In MS, the BCE loss is weighed by a normalized value, which, as Nair et al. (2021) has shown, is equivalent to the Kullback-Leibler divergence between the predicted variable biases and the actual variable biases of the near-optimal solutions.

For all experiments, Adam (KINGMA; BA, 2015) was used to perform stochastic gradient descent on the cost function (average BCE loss) over the training set.

6.3.1 Hyperparameter Tuning

Beyond usual hyperparameters from deep learning, such as learning rate and number of layers, the solution prediction model built for the ONTS problem has several hyperparameters that do not have well-established values in the literature. Several experiments were performed to search for hyperparameter configurations that lead to the best solution prediction model for the ONTS problem. The validation set was used for such experiments and the BCE loss was defined as the metric to be minimized.

[2] In fact, it is not even defined over infeasible solutions.

The choice between the graph convolution operator proposed by Kipf and Welling (2017) (henceforth referred to GCN) and the SAGE model is treated as a hyperparameter. Early experiments with the SAGE model showed no benefits from complex aggregation functions such as an LSTM, as was proposed by Hamilton, Ying and Leskovec (2017). In fact, from the aggregation functions proposed by the authors, the one that performed the best was the element-wise max-pooling. The GCN graph convolution was implemented as proposed originally.

Considering the choice for graph convolution to apply to both half-convolutions (from variable nodes to constraint nodes, and from constraint nodes to variable nodes), it becomes possible to evaluate the effects of parameter sharing. When parameter sharing is enabled, both half-convolutions are performed using the same parameters, instead of parameter vectors specific for that function. Parameter sharing, also called weight tying, has been successfully applied to transformers (deep learning models based on the attention mechanism) (INAN; KHOSRAVI; SOCHER, 2017; PRESS; WOLF, 2017). Due to the proximity between GNNs and transformers (JOSHI, 2020), parameter sharing is evaluated in the context of the ONTS problem.

Beyond the choice for the graph convolution operator and whether or not to share the parameters, hyperparameters related to the model structure and the training are also taken into consideration. Table 3 summarizes all hyperparameters considered for tuning along with value ranges, which were determined in early experiments on the validation set. A random search was performed to select the best hyperparmeter configuration for OS and MS. The configurations were used to train a model for 10 epochs on $\mathcal{D}_{\text{train}}$, after which the model was evaluated on the validation set. Further implementation details and hyperparameter search results can be seen in this project's code repository[3].

Table 3 – Hyperparameters adjusted for the solution prediction models trained with either OS or MS for the ONTS problem. The columns *OS* and *MS* present the best hyperparameter configuration found through random search for both training types.

Hyperparameter	Ranges	OS	MS
Training			
Learning rate	$\{10^{-2}, 10^{-3}, 10^{-4}\}$	10^{-2}	10^{-3}
Architecture			
Number of hidden features (d)	$\{2^5, 2^6, 2^7, 2^8\}$	2^6	2^8
Number of layers (L)	$\{1, 2, 3\}$	2	3
GraphConv			
Operator	{GCN, SAGE}	SAGE	SAGE
Parameter sharing	{Yes, No}	Yes	Yes

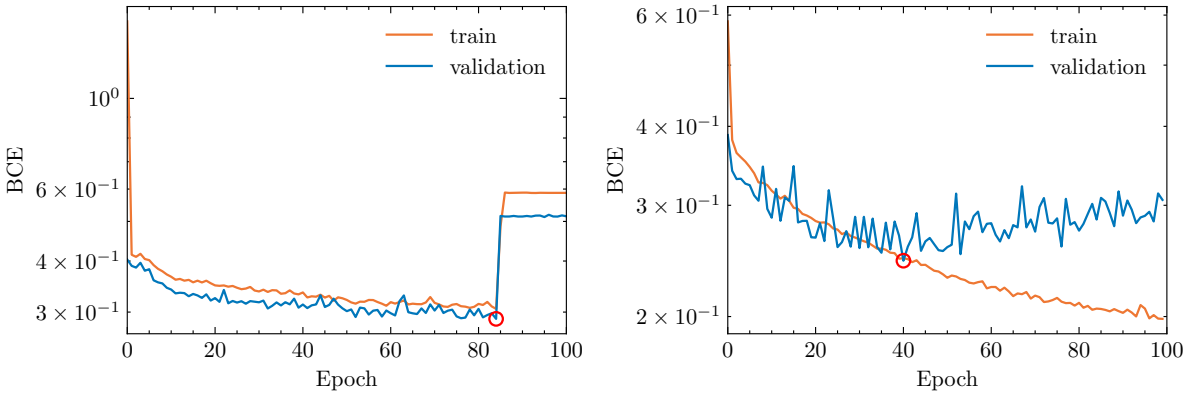
The hyperparameter configuration that resulted in the best model for both OS and MS is presented in Table 3. For both cases, the best model uses the SAGE function instead of GCN, although the best model found for MS is considerably bigger, with more layers and hidden

[3] <https://github.com/gos-ufsc/sat-gnn>

features. Note that both models also perform parameter sharing, suggesting that the proposed approach is effective.

6.3.2 Final solution prediction models

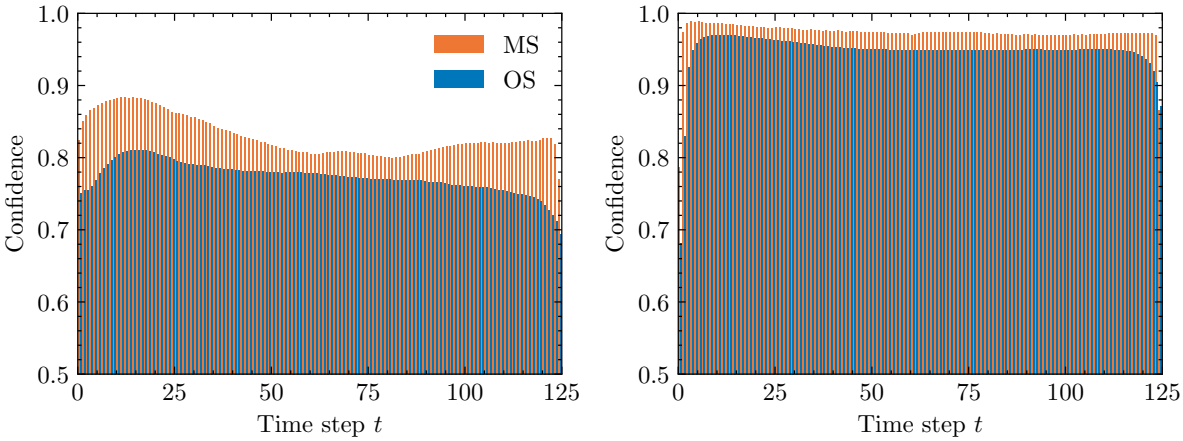
The best hyperparameter configuration found through random search for both OS and MS is used to train new models with a training budget of 100 epochs. However, early-stopping is performed using the validation set, i.e., during the training, the model that performs the best on the validation set (over the epochs) is selected. Early-stopping allows avoiding overfitting without the need to tune the training budget. The training curves can be seen in Fig. 13.



(a) Optimal Solution (b) Multiple Solutions

Figure 13 – Training curves for the best solution prediction models trained through OS (a) or MS (b). The average BCE on the validation set is used for early-stopping the training (highlighted in red).

The model trained with OS achieved a validation cost of 0.2887 and a test cost of 0.2873, whereas the model trained with MS achieved a validation cost of 0.2451 and a test cost of 0.2482. These values cannot be used to compare the training approaches, once the cost functions are different, but they indicate an absence of overfitting in both models, as the validation and test values are very close. However, a comparison across training approaches can be performed when analyzing the confidence of the models on the test set. A model's confidence is the predicted bias of the most likely assignment, i.e., in a binary problem, a model's confidence on the predicted value for the j-th variable is $\hat{p}_j$, if $\hat{y} = 1$, and $1 - \hat{p}_j$ if $\hat{y} = 0$. To consider the entire test set, the confidence is measured at each time step, averaging over all tasks of all instances. The result can be seen in Fig. 14. It is possible to note that the MS model was, on average, much more confident of its predictions than the OS model. Furthermore, both models provide significantly more confident predictions for the $\boldsymbol{\phi}$ variables than the $\boldsymbol{x}$ variables.



(a) $x_{j,t}$ (b) $\phi_{j,t}$

Figure 14 – Average confidence of predicted values for (a) $\boldsymbol{x}$ and (b) $\boldsymbol{\phi}$ variables of the models trained via OS and MS. Each bar is the average confidence over the predictions for all tasks j of all instances I in the test set.

6.4 LEARNING-BASED HEURISTICS

The two solution prediction models (the one with OS and the one with MS) were each used to build three primal matheuristics. Namely, warm-starting, early-fixing and trust-region, as per Sec. 3.3. As these matheuristics have hyperparameters of their own, another set of experiments was performed to find the best values for these hyperparameters. The matheuristics were evaluated considering two possible goals: reducing the time to find a feasible solution, and finding the best solution under 2 minutes. Both goals are directly related to finding a new schedule during the communication window of the nanosatellite's orbit, as detailed in Sec. 6.1.1. Therefore, each model is tuned and evaluated with respect to both goals. The SCIP solver is used both as the baseline and within the matheuristics.

6.4.1 Tuning

All three matheuristics implemented are based on partial solutions, thus, naturally, the size of the partial solution can be seen as a hyperparameter, which can be adjusted through the confidence threshold. On top of that, the trust-region method also has the radius $\Delta \in \mathbb{N}$.

Two sets of experiments are performed for each model using the validation set. For once, the hyperparameters are adjusted with the goal of reducing the average time to find a feasible solution. In parallel, the hyperparameters are adjusted to maximize the average relative objective value (QoS) during a 2-minute time budget. For each instance, the resulting objective values are normalized by the known maximum, following Equation (5.2), but assuming that the trivial solution has 0 objective. This ensures all instances have equal influence in the aggregated value. Because there are few hyperparameters to be tuned, the experiments were performed manually, ensuring equal effort was dedicated to all models and resulting heuristics. The best values found

are reported in Table 4.

Table 4 – Best values for partial solution size N and trust-region radius Δ (when applicable) for each heuristic resulting from both solution prediction models (either trained via OS or MS). Columns *Objective* indicates the values that maximized the relative objective value in the validation set, while columns *Feasibility* indicate the values tuned to minimize the time taken to find a feasible solution.

Training		Objective		Feasibility	
Approach	Heuristic	N	Δ	N	Δ
OS	Warm-start	750	-	1000	-
	Early-fix	500	-	750	-
	Trust region	1000	5	1000	1
MS	Warm-start	1750	-	1500	-
	Early-fix	1000	-	1250	-
	Trust region	1250	1	1750	1

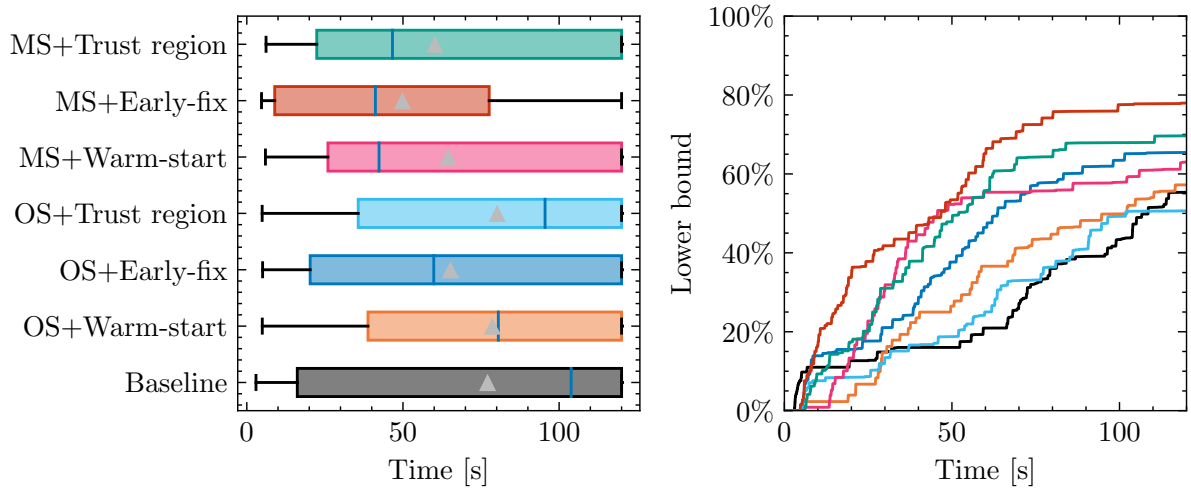
6.4.2 Evaluation

The heuristics with the best partial solution size and trust-region radius (Table 4) are evaluated on the test set, which was not used in any tuning experiment. The models are evaluated with respect to both goals, following the same metrics as for the heuristic hyperparameter tuning experiments, namely, the average time to find a feasible solution and the average relative objective value, under a 2-minute budget. Note that the objective values are normalized following Equation (5.2), with the optimal solution being the best known solution of each instance, but with an "artificial" trivial solution that has null objective (i.e., $QoS = 0$). A null objective is also assumed if the instance is deemed infeasible or as long as the solver cannot find a feasible solution. The progress of the relative lower bound (relative objective of the candidate solution over time) is also measured within the time budget to evaluate how the heuristics perform for smaller budgets. The performance of each heuristic in the test set is presented in Figure 15.

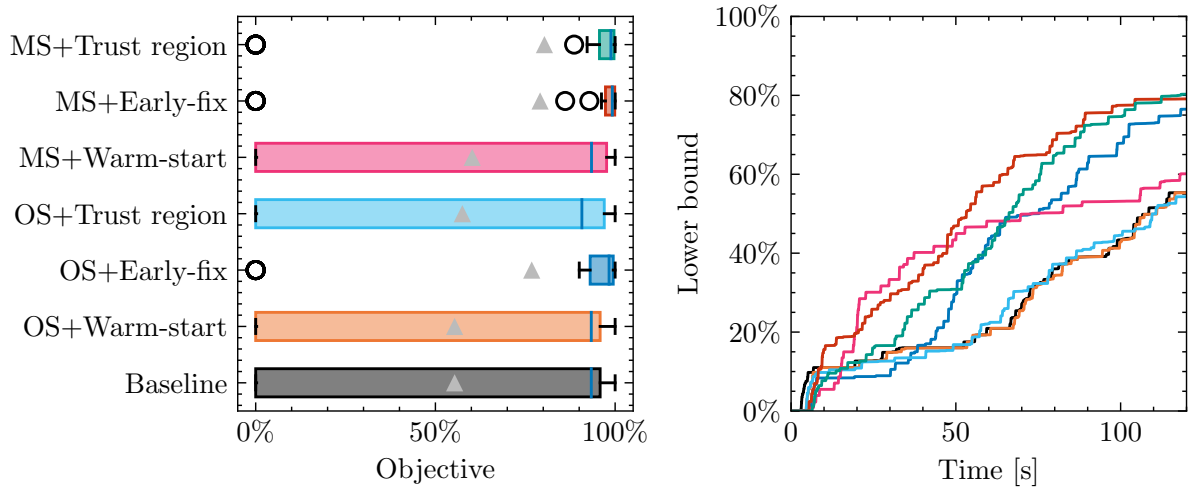
The results indicate that most of the learning-based heuristics provide a clear improvement over the baseline approach of using the SCIP solver, given the limited time budget. To assess the statistical significance of the gains, the Wilcoxon signed-rank test (WILCOXON, 1945) was applied to the test set results. The Wilcoxon signed-rank test is a non-parametric version of the Student's t-test for matched pairs, which implies that normality is not assumed for the distribution of the metrics[4]. The test is applied in pairs, comparing each matheuristic to every other, including the baseline. The results of the statistical significance test are summarized in Figure 16.

For both goals, the early-fixing matheuristics were able to significantly overcome the baseline. In particular, using the solution prediction model with MS to perform early-fixing

[4] This is particularly important for the relative objective value, which is highly skewed and limited to unit interval.



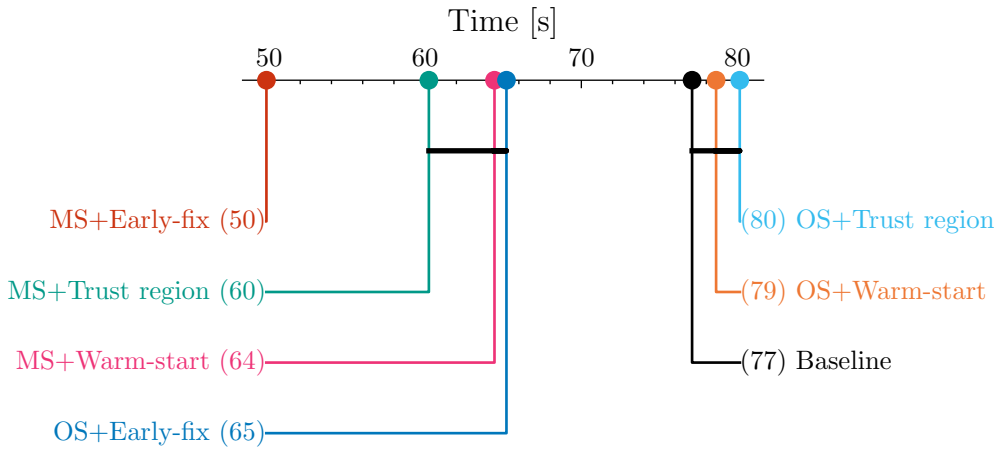
(a) Time to find a feasible solution (lower is better).



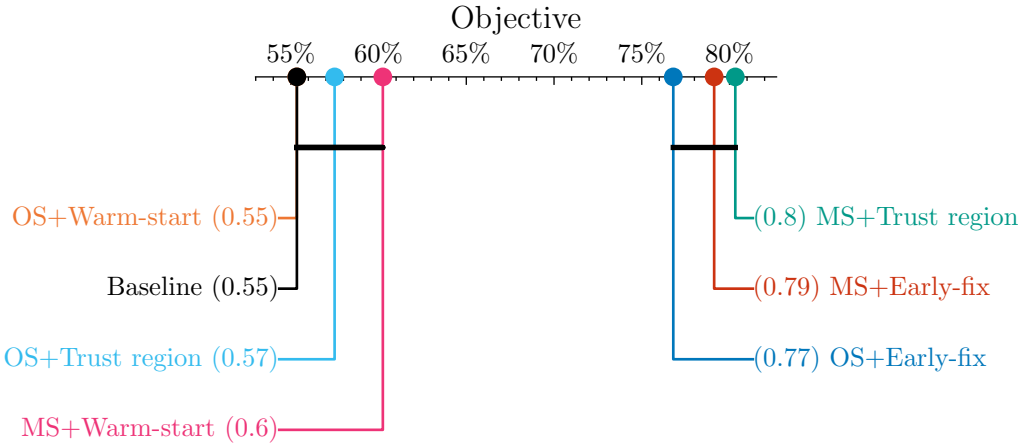
(b) Normalized objective value within 2 minutes (higher is better).

Figure 15 – Test performance of the learning-based heuristics for the ONTS problem. In (a), the boxplots show the quartiles and outliers (circles) of the values that correspond to the heuristics adjusted to minimize the average time to find a feasible solution (on the validation set). The vertical blue lines indicate the medians, while the triangle icons indicate the average values. In (b), the heuristics used were those adjusted to maximize the average relative objective value (also on the validation set). The box plots show the distribution of the metric of interest (time to find a feasible solution in (a), and relative objective value in (b)) over the test set, with the small triangle indicating the average value. The plots on the right show the progress of the relative lower bound (relative objective value of the candidate solution) during the 2 minutes time budget, averaged over all instances of the test set.

provided the most consistent results, being not only significantly better than the baseline, but also significantly better than all other heuristics on the goal of finding a feasible solution the fastest.



(a) Time to find a feasible solution (lower is better).



(b) Normalized objective value within 2 minutes (higher is better).

Figure 16 – Critical difference diagram of the test set performance of the learning-based heuristics for the ONTS problem. Figures (a) and (b) show the performance of heuristics adjusted for minimizing the time to find a feasible solution and maximizing the relative objective value within a 2 minute budget, respectively. The round marker in the axes indicates the heuristic's average performance. A crossbar connecting multiple approaches indicates that their performance (distribution on the test set) was not significantly different (p-value > 0.05) in the paired Wilcoxon signed-rank test.

7 DISCUSSION

The experiments described in Chapter 6 aimed at evaluating the effectiveness of learning-based heuristics in a realistic application, namely, the ONTS problem. The problem setup involves finding the best set of tasks that results in a high-quality, feasible schedule, at every communication window. This translates into solving multiple instances of MILP problems in a small window of time. As a consequence, the NP-hard nature of MILP makes the algorithmic approach to solving such instances challenging. In this case, the baseline solution consists of running an MILP solver with limited time, which has no guarantees of finding feasible or optimal solutions[1].

Although all experiments were performed using data from the ONTS problem, the setup is very general, which renders the results relevant to many different applications. More specifically, the general problem setup is that of repeatedly solving instances of an optimization that follow an unknown distribution, under limited time. This setup appears, e.g., in the management of energy distribution networks, vehicle routing under varying traffic conditions, workload apportioning across workers, and maritime inventory routing (GASSE et al., 2022; PAPAGEORGIOU et al., 2014). In other words, the solution approach evaluated in the presented experiments is of interest in many different application areas.

7.1 SOLUTION PREDICTION MODELS

Two approaches were evaluated: training solely with the optimal solution for each instance (OS) and training with multiple solutions for each instance (MS). Both approaches were successful in training solution prediction models, with no signs of overfitting. Although a direct comparison between solution prediction models was not possible using their own (different) cost functions, the MS approach generated a more confident model, as Figure 14 illustrates. Further experiments demonstrated that, indeed, the models trained via MS resulted in better primal heuristics.

GNNs were used as the core of the solution prediction models, as they are perfectly suitable for instances with a varying number of variables, as is the case of the ONTS problem based on the FloripaSat-I mission. Key architectural features of the model were adjusted through hyperparameter tuning using the validation set. These tuning experiments indicate that the SAGE operator (HAMILTON; YING; LESKOVEC, 2017) is a better graph convolution than the GCN (by Kipf and Welling (2017)) for solution prediction models trained through either MS or OS. These experiments also indicate that MS may profit from larger models, as the best

[1] And, thus, can be said a heuristic solution approach.

hyperparameter configuration found for MS has a larger number of layers and a significantly larger number of hidden features than the best for OS. Finally, the proposed parameter-sharing approach (using the same parameter vector for both half-convolutions) was also beneficial for both training strategies.

7.2 MATHEURISTICS

The solution prediction models were used to build three distinct matheuristics: warm-starting, early-fixing and trust-region. All matheuristics are based on partial solutions generated with the deep learning models. The partial solution size, along with the trust-region radius, was adjusted using the validation set twice for each matheuristic and each solution prediction model (OS and MS): once with the goal of reducing the time to find a feasible solution, and then another aiming to maximize solution quality given 2 minutes.

Warmstarting provided, at best, marginal gains in comparison to the baseline approach. However, a careful inspection of the performance of the heuristic over time (right-hand side plots in Figure 15) shows that there is a "sweet-spot" in terms of time budget (around 60 seconds) for which using the MS model for warmstarting an MILP solver might provide significant improvements. Although the trust-region approach can be seen as a generalization of early-fixing, the results do not back its use. Solving through the trust-region method only marginally outperformed the early-fixing when using the MS model *and* when adjusted for maximizing the normalized objective value. Even then, it is not a statistically significant result and the candidate solutions found over time (right-hand side plot in Figure 15(b)) show, on average, an advantage of the early-fixing approach for almost the entirety of the time budget.

The results show that early-fixing consistently provided significant improvements over the baseline, as the statistical tests illustrated in Fig. 16 show. In particular, the early-fixing matheuristic using the MS model achieved, on one hand, a 35 % reduction in the time to find a feasible solution, and, on the other hand, a 43 % gain in the normalized objective value for the candidate solution found within 2 minutes. These results are not only statistically significant, but impactful with respect to the ONTS problem perspective.

7.3 DATA ACQUISITION AND GENERALIZATION

A shared challenge across applications of learning-based heuristics is that of data acquisition. Historical data is seldom available in the volume necessary to compose a training set suitable for modern deep learning techniques, which leads practitioners to resort to data generation (BENGIO; LODI; PROUVOST, 2021). Generating instances, by itself, is not usually a problem, as parameter ranges can be defined with enough margin to encompass values encountered in practice. However, the solution to these randomly sampled instances is needed.

On top of that, the interest in learning-based heuristic solutions is directly related to the problem difficulty, which, in turn, increases the cost for data generation. In other words, the bigger the potential for learning-based heuristics, the more expensive it is to acquire training data.

A notable limitation of generating instances with a limited time to find an optimal solution, as done in the experiments, is that it restricts the generalization of the results. As discussed by Yehuda, Gabel and Schuster (2020), sampling instances from NP-Hard problems solvable in tractable time essentially means sampling from an easier sub-problem (see also Cappart et al. (2022)). This, however, underscores the generalization capabilities of GNNs demonstrated in this work, indicating that such models can effectively tackle instances harder than those seen during training, reinforcing the results of Gasse et al. (2019).

CONCLUSION

This dissertation evaluates the effectiveness of primal heuristics for Mixed-Integer Linear Programming (MILP) that leverage deep learning-based solution prediction models. The overarching goal was to contribute towards answering the three foundational questions posed in the Introduction regarding the design, training, and integration of these models in primal heuristics.

The key contributions of this work were presented in Chapters 3, 6, and 7. The Offline Nanosatellite Task Scheduling (ONTS) problem served as the application context, providing a challenging benchmark for evaluating the techniques of interest.

First, the architectural components of solution prediction models were analyzed. The selected architecture was based on graph neural networks (GNNs) with layers featuring two half-convolutions, a structure commonly employed in similar optimization problems (GASSE et al., 2019; NAIR et al., 2021; KHALIL; MORRIS; LODI, 2022; CAPPART et al., 2022). Experiments demonstrated that the SAGE operator (HAMILTON; YING; LESKOVEC, 2017) outperformed the original operator proposed by Kipf and Welling (2017). Additionally, the approach of sharing parameters between the two half-convolutions yielded the best-performing models.

Two distinct approaches for training solution prediction models were implemented and evaluated. The results indicated that using multiple solutions from a given instance as targets during training, as suggested by Nair et al. (2021), produced more confident and effective models compared to using only a (quasi-)optimal solution.

Another aspect of training solution prediction models evaluated during the experiments is data acquisition. In the absence of enough historical data, a common challenge to be overcome is the high cost of generating enough data for training solution prediction models (BENGIO; LODI; PROUVOST, 2021; CAPPART et al., 2022; YEHUDA; GABEL; SCHUSTER, 2020).

Data acquisition emerged as a significant challenge, particularly in the absence of sufficient historical data, due to its high computational cost (BENGIO; LODI; PROUVOST, 2021; CAPPART et al., 2022; YEHUDA; GABEL; SCHUSTER, 2020). However, the experiments demonstrated that the GNN architecture could generalize well to instances *harder*[2] than those seen during training, alleviating some of the data acquisition costs. This finding aligns with the results of Gasse et al. (2019), underscoring the robustness and versatility of GNNs in this context.

Finally, the incorporation of solution prediction models into primal heuristics was investigated through experiments with three matheuristic strategies compared to a baseline MILP solver

[2] In terms of computational cost.

with limited time. The approach of early-fixing integer variables based on model predictions consistently outperformed both the trust-region and warmstarting heuristics. This strategy not only provided the best solutions within limited time frames but also found feasible solutions more rapidly.

In summary, this dissertation demonstrated that deep learning-based primal heuristics offer a promising avenue for addressing the challenges of MILP. The research contributes to the growing field of machine learning for combinatorial optimization by providing a thorough evaluation of these heuristics' practical benefits in a representative application. By achieving its objectives and offering valuable insights into the development and application of these heuristics, this work lays the groundwork for further advancements, ultimately contributing to more efficient and adaptable optimization solutions in practice.

Looking ahead, future research should explore the limits of the generalization capacity of GNNs in combinatorial optimization contexts. Understanding these limits more precisely could improve estimates of the trade-off between data acquisition cost and model performance, potentially reducing the overall cost of training solution prediction models. Additionally, enhancing instance generation techniques will be crucial for better supporting the training of deep learning models. As highlighted by Smith-Miles and Bowly (2015), the quality of generated instances is vital for both training and evaluation. Improved instance generation could not only lower data acquisition costs but also enhance confidence in the models' outputs, further advancing the field.

BIBLIOGRAPHY

BENGIO, Y.; LODI, A.; PROUVOST, A. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, v. 290, n. 2, p. 405–421, 2021. ISSN 0377-2217.

BESTUZHEVA, K. et al. The SCIP Optimization Suite 8.0. 2021. ISSN 1438-0064.

BISHOP, C. M. *Pattern Recognition and Machine Learning*. New York: Springer, 2006. (Information Science and Statistics). ISBN 978-0-387-31073-2.

BREIMAN, L. et al. *Classification And Regression Trees*. 1. ed. [S.l.]: Routledge, 2017. ISBN 978-1-315-13947-0.

CAMPONOGARA, E.; NAZARI, L. F. Models and algorithms for optimal piecewise-linear function approximation. *Mathematical Problems in Engineering*, v. 2015, p. 1–9, 2015. ISSN 1024-123X, 1563-5147.

CAMPONOGARA, E. et al. A continuous-time formulation for optimal task scheduling and quality-of-service assurance in nanosatellites. *Computers & Operations Research*, v. 147, p. 105945, 2022. ISSN 03050548.

CAPPART, Q. et al. *Combinatorial Optimization and Reasoning with Graph Neural Networks*. [S.l.]: arXiv, 2022.

CAUCHY, A. Methode generale pour la resolution des systemes d'equations simultanees. *C.R. Acad. Sci. Paris*, v. 25, p. 536–538, 1847.

CHAMON, L. F. D. O. *Constrained Learning And Inference*. Tese (Doutorado) — University of Pennsylvania, 2020.

CHEN, Z. et al. *On Representing Mixed-Integer Linear Programs by Graph Neural Networks*. [S.l.]: arXiv, 2022.

DANTZIG, G. B. On the significance of solving linear programming problems with some integer variables. *Econometrica*, v. 28, n. 1, p. 30, 1960. ISSN 00129682.

DING, J.-Y. et al. Accelerating primal solution findings for mixed integer programs based on solution prediction. *Proceedings of the AAAI Conference on Artificial Intelligence*, v. 34, n. 02, p. 1452–1459, 2020. ISSN 2374-3468, 2159-5399.

FILHO, E. M. et al. A comprehensive attitude formulation with spin for numerical model of irradiance for CubeSats and Picosats. *Applied Thermal Engineering*, v. 168, p. 114859, 2020. ISSN 13594311.

FISCHETTI, M.; FISCHETTI, M. Matheuristics. In: MARTÍ, R.; PANOS, P.; RESENDE, M. G. (Ed.). *Handbook of Heuristics*. Cham: Springer International Publishing, 2016. p. 1–33. ISBN 978-3-319-07153-4.

FISCHETTI, M.; LODI, A. Heuristics in Mixed Integer Programming. In: ____. *Wiley Encyclopedia of Operations Research and Management Science*. 1. ed. [S.l.]: Wiley, 2011. ISBN 978-0-470-40063-0 978-0-470-40053-1.

GASSE, M. et al. The Machine Learning for Combinatorial Optimization Competition (ML4CO): Results and Insights. In: *Proceedings of the NeurIPS 2021 Competitions and Demonstrations Track*. [S.l.]: PMLR, 2022. p. 220–231. ISSN 2640-3498.

GASSE, M. et al. *Exact Combinatorial Optimization with Graph Convolutional Neural Networks*. [S.l.]: arXiv, 2019.

GILMER, J. et al. Neural message passing for quantum chemistry. In: *Proceedings of the 34th International Conference on Machine Learning*. [S.l.]: PMLR, 2017. p. 1263–1272. ISSN 2640-3498.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. [S.l.]: MIT Press, 2016.

GOODFELLOW, I. J.; VINYALS, O.; SAXE, A. M. *Qualitatively Characterizing Neural Network Optimization Problems*. [S.l.]: arXiv, 2015.

HAMILTON, W.; YING, Z.; LESKOVEC, J. Inductive representation learning on large graphs. *Advances in neural information processing systems*, v. 30, 2017.

HAN, Q. et al. *A GNN-Guided Predict-and-Search Framework for Mixed-Integer Linear Programming*. [S.l.]: arXiv, 2023.

HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. H. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd ed. ed. New York, NY: Springer, 2009. (Springer Series in Statistics). ISBN 978-0-387-84857-0 978-0-387-84858-7.

INAN, H.; KHOSRAVI, K.; SOCHER, R. Tying word vectors and word classifiers: A loss framework for language modeling. In: *International Conference on Learning Representations*. [S.l.: s.n.], 2017.

JACOBS, R. A. Increased rates of convergence through learning rate adaptation. *Neural Networks*, v. 1, n. 4, p. 295–307, 1988. ISSN 08936080.

JOSHI, C. K. *Transformers Are Graph Neural Networks*. 2020.

KARP, R. M. Reducibility among Combinatorial Problems. In: MILLER, R. E.; THATCHER, J. W.; BOHLINGER, J. D. (Ed.). *Complexity of Computer Computations*. Boston, MA: Springer US, 1972. p. 85–103. ISBN 978-1-4684-2003-6 978-1-4684-2001-2.

KHALIL, E. B.; MORRIS, C.; LODI, A. MIP-GNN: A data-driven framework for guiding combinatorial solvers. *Proceedings of the AAAI Conference on Artificial Intelligence*, v. 36, n. 9, p. 10219–10227, 2022. ISSN 2374-3468.

KINGMA, D. P.; BA, J. Adam: A Method for Stochastic Optimization. In: BENGIO, Y.; LECUN, Y. (Ed.). *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. [S.l.: s.n.], 2015.

KIPF, T. N.; WELLING, M. *Semi-Supervised Classification with Graph Convolutional Networks*. [S.l.]: arXiv, 2017.

LARSEN, E. et al. Predicting Tactical Solutions to Operational Planning Problems Under Imperfect Information. *INFORMS Journal on Computing*, INFORMS, v. 34, n. 1, p. 227–242, 2022. ISSN 1091-9856.

LUCIA, B. et al. Computational Nanosatellite Constellations: Opportunities and Challenges. *GetMobile: Mobile Computing and Communications*, v. 25, n. 1, p. 16–23, 2021. ISSN 2375-0529, 2375-0537.

MANIEZZO, V.; BOSCHETTI, M. A.; STÜTZLE, T. *Matheuristics: Algorithms and Implementations*. Cham: Springer International Publishing, 2021. (EURO Advanced Tutorials on Operational Research). ISBN 978-3-030-70276-2 978-3-030-70277-9.

MARCELINO, G. M. et al. A Critical Embedded System Challenge: The FloripaSat-1 Mission. *IEEE Latin America Transactions*, v. 18, n. 02, p. 249–256, 2020. ISSN 1548-0992.

MILLER, C. E.; TUCKER, A. W.; ZEMLIN, R. A. Integer Programming Formulation of Traveling Salesman Problems. *Journal of the ACM*, v. 7, n. 4, p. 326–329, 1960. ISSN 0004-5411.

MURPHY, K. P. *Machine Learning: A Probabilistic Perspective*. 4. print. (fixed many typos). ed. Cambridge, Mass.: MIT Press, 2013. (Adaptive Computation and Machine Learning Series). ISBN 978-0-262-01802-9.

NAGEL, G. W.; NOVO, E. M. L. D. M.; KAMPEL, M. Nanosatellites applied to optical Earth observation: A review. *Ambiente e Agua - An Interdisciplinary Journal of Applied Science*, v. 15, n. 3, p. 1, 2020. ISSN 1980-993X.

NAIR, V. et al. *Solving Mixed Integer Programs Using Neural Networks*. [S.l.]: arXiv, 2021.

NEMHAUSER, G.; WOLSEY, L. The Scope of Integer and Combinatorial Optimization. In: *Integer and Combinatorial Optimization*. 1. ed. [S.l.]: Wiley, 1988. ISBN 978-0-471-82819-8 978-1-118-62737-2.

NEMHAUSER, G. L.; WOLSEY, L. A. *Integer and Combinatorial Optimization*. New York, NY Weinheim: Wiley, 1999. (Wiley-Interscience Series in Discrete Mathematics and Optimization). ISBN 978-0-471-35943-2 978-0-471-82819-8.

NESTEROV, Y. *A Method of Solving a Convex Programming Problem with Convergence Rate O(1/K**2)*. 1983. 543 p.

PAPAGEORGIOU, D. J. et al. MIRPLib – A library of maritime inventory routing problem instances: Survey, core model, and benchmark results. *European Journal of Operational Research*, v. 235, n. 2, p. 350–366, 2014. ISSN 03772217.

POLYAK, B. T.; JUDITSKY, A. B. Acceleration of Stochastic Approximation by Averaging. *SIAM Journal on Control and Optimization*, v. 30, n. 4, p. 838–855, 1992. ISSN 0363-0129, 1095-7138.

PRESS, O.; WOLF, L. Using the output embedding to improve language models. In: LAPATA, M.; BLUNSOM, P.; KOLLER, A. (Ed.). *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*. Valencia, Spain: Association for Computational Linguistics, 2017. p. 157–163.

RIGO, C. A. et al. A nanosatellite task scheduling framework to improve mission value using fuzzy constraints. *Expert Systems with Applications*, v. 175, p. 114784, 2021. ISSN 09574174.

RIGO, C. A. et al. Task scheduling for optimal power management and quality-of-service assurance in CubeSats. *Acta Astronautica*, v. 179, p. 550–560, 2021. ISSN 00945765.

RIGO, C. A. et al. A branch-and-price algorithm for nanosatellite task scheduling to improve mission quality-of-service. *European Journal of Operational Research*, v. 303, n. 1, p. 168–183, 2022. ISSN 03772217.

SAEED, N. et al. CubeSat Communications: Recent Advances and Future Challenges. *IEEE Communications Surveys & Tutorials*, v. 22, n. 3, p. 1839–1862, 2020. ISSN 1553-877X, 2373-745X.

Sanchez-Lengeling, B. et al. A Gentle Introduction to Graph Neural Networks. *Distill*, v. 6, n. 8, p. 10.23915/distill.00033, 2021. ISSN 2476-0757.

SEMAN, L. O. et al. An Energy-Aware Task Scheduling for Quality-of-Service Assurance in Constellations of Nanosatellites. *Sensors*, v. 22, n. 10, p. 3715, 2022. ISSN 1424-8220.

SHIROMA, W. A. et al. CubeSats: A bright future for nanosatellites. *Central European Journal of Engineering*, v. 1, n. 1, p. 9–15, 2011. ISSN 2081-9927.

Smith-Miles, K.; BOWLY, S. Generating new test instances by evolving in instance space. *Computers & Operations Research*, v. 63, p. 102–113, 2015. ISSN 03050548.

VANDERBEI, R. J. *Linear Programming: Foundations and Extensions*. 3. printing. ed. Boston: Kluwer Acad. Publ, 1998. (International Series in Operations Research & Management Science, 4). ISBN 978-0-7923-8141-9 978-0-7923-9804-2.

VAPNIK, V. N. *The Nature of Statistical Learning Theory*. Second edition. New York, NY: Springer New York : Imprint : Springer, 2000. ISBN 978-1-4757-3264-1.

VELIČKOVIĆ, P. et al. Graph attention networks. In: *International Conference on Learning Representations*. [S.l.: s.n.], 2018.

WILCOXON, F. Individual comparisons by ranking methods. *Biometrics Bulletin*, [International Biometric Society, Wiley], v. 1, n. 6, p. 80–83, 1945. ISSN 00994987.

WOLSEY, L. Formulations. In: *Integer Programming*. 1. ed. [S.l.]: Wiley, 2020. p. 1–23. ISBN 978-1-119-60653-6 978-1-119-60647-5.

WOLSEY, L. A. *Integer Programming*. New York: Wiley, 1998. (Wiley-Interscience Series in Discrete Mathematics and Optimization). ISBN 978-0-471-28366-9.

YEHUDA, G.; GABEL, M.; SCHUSTER, A. It's not what machines can learn, it's what we cannot teach. In: III, H. D.; SINGH, A. (Ed.). *Proceedings of the 37th International Conference on Machine Learning*. [S.l.]: PMLR, 2020. (Proceedings of Machine Learning Research, v. 119), p. 10831–10841.