Exploring the Synergies Between Operations Research and Machine Learning

by

Xinglu Wang

M.Sc., Zhejiang University, 2021
B.Sc., Zhejiang University, 2018

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

Declaration of Committee

Name:	**Xinglu Wang**
Degree:	**Doctor of Philosophy**
Thesis title:	**Exploring the Synergies Between Operations Research and Machine Learning**
Committee:	**Chair:** Jiangchuan Liu Professor, Computing Science

Jian Pei
Co-Supervisor
Professor Emeritus, Computing Science

Jiannan Wang
Co-Supervisor
Associate Professor, Computing Science

Tianzheng Wang
Committee Member
Associate Professor, Computing Science

Wuyang Chen
Examiner
Assistant Professor, Computing Science

Wei Wang
External Examiner
Professor
Data Science and Analytics
The Hong Kong University of Science and Technology

Abstract

Operations Research (OR) and Machine Learning (ML) are two powerful tools for solving complex real-world problems, each following its own distinct problem-solving paradigm. However, as real-world problems grow in scale, relying on a single tool in isolation faces efficiency challenges, and we seek to overcome these challenges by leveraging the strengths of the other tool. Specifically, this thesis explores the mutual enhancement of OR and ML, examining how ML can improve key steps in the OR pipeline and how OR can strengthen specific stages of the ML lifecycle. The guiding principle is to learn from historical data when available and to optimize when trade-offs exist.

We present three representative works demonstrating these synergies. For ML enhancing OR, we begin by proposing an ML-guided initialization strategy for the Simplex method to accelerate linear program (LP) solving in industrial applications. Leveraging graph neural networks and past solving experience, this strategy substantially reduces solving time compared with rule-based heuristics. Furthermore, we introduce a human-aligned evaluation metric for the Natural Language to Linear Program (NL2LP) task, based on graph edit distance, offering a more semantically faithful assessment of mathematical formulations generated by large language models (LLMs). For OR enhancing ML, we design HyperFlexis, an OR-inspired scheduling framework for LLM serving systems that manages heterogeneous service-level objectives (SLOs) while balancing user satisfaction and operational cost.

Together, these studies illustrate how OR and ML can mutually enhance each other to enable more efficient and scalable real-world problem solving. The thesis concludes by outlining future research directions and broader opportunities for synergy between the two fields.

Keywords: Operations Research; Machine Learning; Linear Programming; Bipartite Graph; Large Language Model; Contribution Valuation; Model Serving

Dedication

To my family, for their unwavering love and support.

Acknowledgements

I would like to express my deepest gratitude to Professor Jian Pei for his invaluable guidance, insightful advice, and unwavering support throughout my PhD journey. His intellectual rigor and high standards have profoundly influenced not only this dissertation but also my broader approach to research and engineering. From carefully defining research problems and examining underlying assumptions to thinking from new perspectives and developing novel technical contributions, he has taught me how to approach complex challenges in a structured and principled manner. Meanwhile, I am also deeply thankful to Professor Jiannan Wang for his thoughtful feedback and continuous support throughout the preparation of this thesis. His constructive comments significantly improved the quality and clarity of this work. I am especially grateful for his generous investment of time and effort throughout the dissertation process, including his coordination and support at critical stages of the defense preparation.

My sincere appreciation extends to Professor Jiangchuan Liu for chairing my defense, and to Professor Tianzheng Wang for his valuable discussions and insightful comments as a committee member, particularly on the part of this dissertation concerning LLM serving systems. I am also grateful to Professor Wuyang Chen and Professor Wei Wang for serving as examiners and providing constructive feedback that has strengthened this work. I sincerely thank all committee members for their time, expertise, and thoughtful evaluation.

I would also like to thank my collaborators in industry, with special thanks to Dr. Zhenan Fan and Dr. Yong Zhang, for their practical support and valuable resources, which enabled the application and validation of our research methods in real-world settings.

Finally, I am profoundly grateful to my friends and colleagues at Simon Fraser University, especially Chirong Zhang, Xuan Luo, Zicun Cong, and Chen Xu, for their companionship and support throughout the years. I am equally grateful to my family for their constant encouragement and unconditional support. Their presence has been a continuous source of strength and motivation throughout this journey.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Operations Research (OR) and Machine Learning (ML) are two important tools for addressing a wide range of real-world problems. On the one hand, OR [156, 132] continues to underpin large-scale industrial problems such as supply chain management, where practitioners formulate decision problems with explicit objectives and constraints and solve them using optimization algorithms. On the other hand, advances in ML [44] have made it increasingly attractive for a wide range of data-driven tasks, from prediction to language understanding, where models learn complex patterns directly from data. These two problem-solving paradigms excel at different classes of problems because their respective strengths align with different problem structures: OR specializes in handling trade-offs and operational constraints, making it well-suited for management and decision-making tasks, whereas ML specializes in learning high-dimensional patterns from data, making it naturally suited for prediction problems.

As modern problems grow in scale and complexity, the OR and ML pipelines for solving real-world problems encounter new efficiency challenges. In particular, we highlight three representative efficiency challenges. First, solving large-scale optimization problems increasingly faces a computational efficiency challenge as modern instances grow in both size and frequency. In many industrial settings, decision makers must solve a sequence of large-scale optimization problems under tight time windows. Second, the formulation stage of OR encounters a modeling efficiency challenge. As real-world decision problems expand in scope, their descriptions become more lengthy and complex. A user's requirements may span multi-page documents containing objectives, operational rules, exceptions, and domain constraints. Translating such descriptions into a precise mathematical program requires substantial expert knowledge, and this manual process does not scale. Even experienced practitioners may struggle to produce formulations that are both accurate and complete, as new modeling variants continue to proliferate across textbooks, research papers, and industry reports. Third, the deployment and serving stages of the ML lifecycle exhibit a system efficiency challenge. As modern ML services scale to thousands or millions of concurrent users, the serving system must handle a large volume of requests under stringent latency requirements. Balancing user satisfaction against the computational cost of operating cloud-hosted models is inherently difficult, and this tension becomes more pronounced as demand grows.

These efficiency challenges, however, also reveal opportunities for synergy, meaning that the strengths of one paradigm could mitigate the weaknesses of the other. Consider the computational efficiency challenge. Traditional optimization algorithms typically treat each problem instance in isolation, even when the instances arise repeatedly and share strong structural similarities—such as the daily or hourly optimization tasks common in industrial operations. These repeated optimization workflows generate extensive historical data— past instances, solution trajectories, and structural patterns—from which ML models can learn to improve existing solving strategies or guide the solver's search, thereby accelerating classical optimization methods. The modeling efficiency challenge likewise benefits from ML's ability to learn from large collections of unstructured text on mathematical programming. As problem descriptions, domain rules, and modeling strategies accumulate across documents and formulated examples, ML, especially large language models (LLMs), can assist in automating the formulation process, improving consistency and reducing expert workload. The system efficiency challenge, however, is one that OR techniques are especially effective at addressing. Scheduling, resource allocation, and the management of latency–cost trade-offs are classical OR problems, suggesting that OR methods can provide structured and principled strategies for operating large-scale LLM serving systems more efficiently.

This thesis explores the efficiency challenges that arise across the OR and ML pipelines and examines how the two paradigms can complement one another in addressing these challenges. Figure 1.1 provides a high-level overview of this idea and the resulting research questions.

1.1 Motivation for Enhancing the OR Pipeline with ML

The existing OR pipeline is highly effective for structured decision-making problems and is well known for its ability to express trade-offs among different constraints and objective. It has been widely and stably used in large-scale industrial applications such as supply chain management. Given this success, why do we still need to enhance it? Why does ML have the opportunity to improve it? We start with a concrete example.

Example 1 (A simple production and inventory planning problem)**.** *Imagine a fruit manufacturer producing a single product, boxes of apple slices, over a planning horizon of four days. Customer demand is known in advance: the manufacturer must deliver 20 boxes on day 1, 30 boxes on day 2, 1 boxes on day 3, and 40 boxes on day 4. Demand on each day must be satisfied on that same day.*

On each day, the manufacturer may choose to produce apple slices. Daily production is limited by available machines and labor, and at most 30 boxes can be produced on any

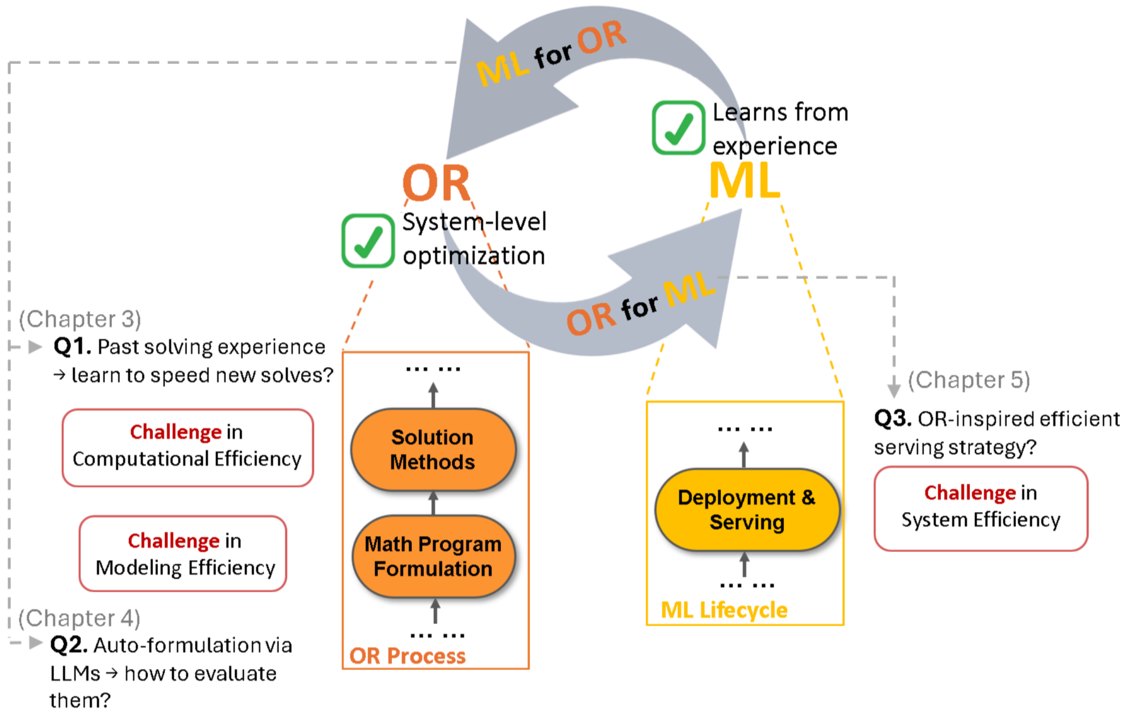

ML for OR
Learns from experience
OR
ML
System-level optimization
OR for ML
(Chapter 3)
Q1. Past solving experience → learn to speed new solves?
Challenge in Computational Efficiency
Challenge in Modeling Efficiency
(Chapter 4)
Q2. Auto-formulation via LLMs → how to evaluate them?
Solution Methods
Math Program Formulation
OR Process
Deployment & Serving
ML Lifecycle
(Chapter 5)
Q3. OR-inspired efficient serving strategy?
Challenge in System Efficiency

Figure 1.1: This thesis is guided by the idea that when real-world problems are solved through one tool (OR or ML), challenges may arise, which can potentially be addressed by leveraging the strengths of the other tool. This idea leads to three research questions: Q1 and Q2 study how ML can enhance OR, while Q3 explores how OR can enhance ML.

single day. If the manufacturer decides to produce on a given day, a fixed setup cost of $100 is incurred for that day, regardless of how many boxes are produced. In addition, producing each box incurs a unit production cost of $2. Boxes produced on a given day can be used immediately to satisfy that day's demand. If more boxes are produced than needed, the excess can be stored as inventory and used to satisfy demand on later days. However, storing inventory from one day to the next incurs a holding cost of $1 per box per day due to refrigeration and spoilage risk.

The manufacturer must decide, for each day, whether to produce and how many boxes to produce, satisfying all daily demand at minimum total cost. The core challenge is balancing fixed setup costs incurred when production takes place against inventory holding costs arising from producing items ahead of demand.

At first glance, Example 1 appears simple: a reader can typically grasp the problem setting within a short amount of time. This simplicity, however, is not inherent to the underlying decision-making task. Rather, it is the result of abstraction and refinement in the OR literature. In fact, Example 1 corresponds to the well-studied capacitated lot-sizing problem [88]. In practice, real-world supply chain settings are described through lengthy and complex requirement documents that contain numerous operational details. OR experts systematically analyze these documents, distill the essential decision-relevant information, and abstract away secondary complexities to arrive at a concise and structured problem description.

In practice, planning problems encountered in real-world applications are rarely specified directly as clean mathematical models. Instead, they are often described through lengthy and unstructured documents. More critically, these descriptions may evolve rapidly as business conditions change—for example, when new products are introduced or resource configurations are adjusted. Meanwhile, consider settings in which the planning horizon is on the order of hours. Under such time pressure, domain experts must rapidly interpret evolving requirements, distill decision-relevant information, and reformulate the underlying optimization model before it can be deployed to support operational decisions. This process is highly labor-intensive and does not scale well.

These considerations reveal a fundamental *modeling efficiency challenge* in the OR pipeline. There is a strong need for automated or semi-automated methods that can assist OR experts in rapidly translating evolving, unstructured problem descriptions into precise mathematical programs. In the extreme, one may even envision fully automated approaches that generate optimization models directly from problem specifications. Fortunately, decades of prior work and accumulated practice have resulted in a large collection of historical problem descriptions and their corresponding formulations. This abundance of data makes it possible for machine learning models to play a meaningful role in supporting the formulation process. Once ML-enhanced formulation methods are introduced, an additional question naturally arises. How should the quality of automatically generated formulations be evaluated? Ideally, an evaluation metric should satisfy two properties: it should be automated, enabling large-scale assessment without human intervention, and it should be *human-aligned*, meaning that it reflects how domain experts judge the correctness and quality of a formulation.

As for the solution stage of Example 1, due to the presence of capacity constraints, this problem cannot be solved using dynamic programming and instead requires a linear programming (LP) formulation. While it is straightforward to write down the LP for the toy example, extending this setting to a realistic application quickly leads to a substantial increase in scale. For instance, consider a setting in which planning must support thousands of products, where production resources and machines are shared across products, and the planning horizon is defined at an hourly granularity. Such seemingly modest extensions naturally result in large-scale linear programs with thousands of decision variables and constraints.

Solving problems of this scale is computationally demanding. Even with highly optimized modern LP solvers [38], computing a high-quality solution can take tens of minutes or longer. At the same time, practical planning systems often operate under tight time constraints.

Customer demand may fluctuate on an hourly basis, and updated production plans must be generated quickly in order to support downstream manufacturing and logistics decisions.

While this computational efficiency challenge arises, there are also opportunities. In many practical settings, similar planning problems are solved repeatedly over time, resulting in a rich repository of historical instances and solution experience. Machine learning methods could leverage such past experience to enhance the computational efficiency of solving large-scale optimization problems.

To summarize, real-world OR practice exposes efficiency limitations that ML is particularly well-suited to address. Importantly, when such efficiency limitations arise, they are often accompanied by abundant historical data. This creates not only challenges but also significant opportunities: historical data can be leveraged to mitigate the aforementioned efficiency limitations. The guiding principle in this direction is to *learn from historical data when available*, as illustrated in Figure 1.1. We identify the following research questions:

- **Q1.** Can ML accelerate optimization algorithms by learning from past solving experience?
- **Q2.** Can ML be used to automate model formulation, and if so, what evaluation metrics are needed to assess ML-generated formulations?

1.2 Motivation for Enhancing the ML Lifecycle with OR

ML models are typically developed and operated through a lifecycle that includes data collection, model training, evaluation, deployment, and continuous operation. Much of the ML literature has focused on the training and evaluation stages, with the primary goal of improving predictive accuracy and generalization performance. This extensive body of work has led to astonishing performance improvements in recent years. In particular, large language models (LLMs) have achieved remarkable success across a wide range of applications, demonstrating strong capabilities in language understanding, reasoning, and generation. These models are typically very large, often containing billions of parameters, and are deployed as centralized services on large-scale computing clusters, serving a diverse set of applications and a large number of concurrent users [44, 169]. Under this serving paradigm, deployment-time decisions—such as request scheduling and resource allocation—directly affect both user experience and operational cost, giving rise to fundamental system efficiency challenges.

Example 2 (System Efficiency Challenges in LLM Serving)**.** *Imagine an LLM serving system that processes requests from two applications: (i) an offline paper summarization service, where each request contains a long document, incurs heavy computational load, and is tolerant to large end-to-end delays (e.g., completing within hours is acceptable), and (ii)* *an interactive chatbot service, where requests are short and lightweight but require low and stable latency at the sub-second timescale for responsive, streaming interaction.*

Heterogeneous latency requirements. *Requests are dispatched from a global queue to multiple LLM instances that share GPU resources. If dispatching decisions ignore heterogeneous load and latency requirements, long-running summarization requests may interfere with latency-sensitive chatbot requests. Such interference introduces head-of-line blocking and sustained resource contention, degrading the interactive user experience and increasing overall system cost.*

Time-varying workloads. *Consider the chatbot application is under a highly time-varying request arrival rate. During daytime hours, the system may receive thousands of requests per second, whereas at night, the arrival rate may drop to only tens of requests per second. If the serving system cannot dynamically scale the number of active instances and allocated GPU resources, it must provision capacity for peak demand. This leads to severe underutilization during low-load periods, or excessive queuing and latency violations during peak periods if capacity is provisioned conservatively.*

Example 2 highlights the system efficiency challenges in large-scale LLM serving. Practical LLM serving systems operate in highly complex environments: they must simultaneously handle *heterogeneous workloads* with diverse latency requirements, adapt to *load variation* over time, and account for other practical considerations. This complexity is further amplified by competing objectives, as serving systems must balance user satisfaction (e.g., request latency) against system-level costs (e.g., GPU rental expense).

Despite its success in prediction and representation learning, ML itself lacks principled mechanisms for addressing such resource management problems and for managing these tradeoffs. In contrast, OR is explicitly designed to optimize decision-making when tradeoffs exist. As illustrated in Figure 1.1, we ask the third research question, **Q3**: *Is there an OR-inspired efficient serving strategy?*

Together, these research questions define the scope of this thesis, which examines how ML can enhance key steps in the OR pipeline and how OR can improve the deployment stage of the ML lifecycle.

1.3 Overview of Contributions

This thesis focuses on OR–ML synergies and presents three research works [55, 162, 167] that answer the three research questions, respectively.

- **Smart Initial Basis Selection for Linear Programs** [55]. We focus on accelerating the solution of large-scale linear programs (LPs), which arise in many real-world applications such as supply chain planning. In practice, LP instances may involve millions of variables and constraints. Even with highly optimized commercial solvers, computing near-optimal solutions can still take hours [82, 83]. While Simplex algorithm remains the workhorse of modern LP solvers [63], its performance is highly sensitive to initialization, since the algorithm iteratively moves from an initial feasible point toward an optimal solution [15]. Traditional initialization strategies are largely rule-based. They typically exploit only algebraic information from a single instance and do not leverage historical solving experience. Motivated by the correlations commonly observed across real-world LP instances, we propose a learning-based approach for selecting a high-quality initial basis. Specifically, using past solving experience, we learn a mapping from a new LP instance to a near-optimal initial point. Designing such a learnable mapping is nontrivial: (1) It must generalize across LPs of varying sizes. (2) It must be permutation equivariant with respect to variables and constraints. (3) It must also produce a valid feasible initialization. To meet the first two requirements, we represent LPs as graphs and employ a graph neural network. To ensure validity, we identify the mathematical conditions required for feasible initialization and design corresponding post-processing steps. Extensive experiments demonstrate substantial reductions in both iteration count and total solving time compared with rule-based initialization strategies.

- **Towards Human-Aligned Evaluation for Natural Language to Linear Program (NL2LP)** [162]. Recent progress has made it increasingly practical to use LLMs to automate mathematical program formulation from natural language, and such LLM-based auto-formulation has been actively studied in our survey [54] and adopted in the OR literature [133, 127]. As this direction matures, we focus on a further challenge: designing evaluation metrics for generated formulations. In large-scale settings, a desirable evaluation metric should be *automated*, meaning that it does not require human experts in the evaluation loop, and *human-aligned*, meaning that it aligns with human judgments of formulation quality. Existing NL2LP evaluation metrics satisfy the automation requirement, but fail to achieve good human alignment. In particular, they do not faithfully capture the semantic equivalence between formulations and are sensitive to superficial syntactic differences, such as permutations of variables or constraints, even though such differences do not affect how human experts assess the formulations [133, 127]. To address this gap, we propose a human-aligned evaluation metric for NL2LP based on graph edit distance. Our metric converts predicted and reference LPs into attributed bipartite graphs and measures their similarity in a permutation-invariant manner, thereby better reflecting the semantic correspondence between decision variables, objectives, and constraints. Human studies further show that the proposed metric aligns more closely with human judgments than prior metrics, providing a reliable and automated evaluation tool for LLM-generated LP formulations.

- **HyperFlexis: Joint Design of Algorithms and Systems for Multi-SLO Serving and Fast Scaling** [167]. We propose a unified LLM serving system designed to efficiently handle highly heterogeneous workloads, where each request has a latency requirement, referred to as a service-level objective (SLO). A fundamental trade-off exists between achieving high SLO attainment and maintaining low overall system cost. Additional challenges include (1) designing scheduling algorithms capable of real-time decision-making under dynamic and unpredictable workloads, (2) supporting rapid scaling to accommodate workload fluctuations and minimizing cold-start latency for newly provisioned instances, and (3) remaining compatible with different deployment modes used in modern LLM serving frameworks. To address these challenges, we explore designs at both the system and algorithmic levels. At the system level, we implement fast scaling mechanisms to minimize cold-start time while being compatible with different deployment modes. At the algorithmic level, we develop OR-inspired greedy heuristics to guide scheduling and scaling decisions, aiming to maximize SLO attainment while minimizing operational costs.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background on OR and ML, reviewing their key concepts and steps, and the literature relevant to the synergies explored in this thesis. Chapter 3 presents our work on smart initial basis selection for linear programs, demonstrating how ML can enhance OR efficiency — accelerate solution methods in the OR process by learning from past solving experiences. Chapter 4 further advances the ML-for-OR direction by studying the NL2LP task, which aims to automate the formulation of mathematical optimization models, and introduces a human-aligned evaluation metric tailored for this task. Chapter 5 reverses the direction and examines how OR principles can enhance ML system performance. It presents HyperFlexis, a unified serving system for heterogeneous LLM workloads, which leverages OR-inspired scheduling and multi-SLO optimization to improve model serving efficiency and resource allocation in large-scale ML deployments. Finally, Chapter 6 concludes the thesis with a summary of our contributions, discusses the broader implications of synergies between OR and ML, and suggests directions for future research.

Chapter 2

Representative Problem-Solving Steps and Related Works

OR and ML solve real-world problems following distinct disciplinary processes. OR approaches decision-making by formulating mathematical programs and solving them under explicit constraints [156, 132]. A typical OR workflow consists of several stages: (1) *problem identification*, where the decision variables, objectives, and constraints of the task are specified; (2) *mathematical program formulation*, where these elements are encoded into a structured optimization model; (3) *parameter generation*, which supplies model coefficients such as costs or constraint matrices; (4) *solution methods*, where algorithms such as Simplex, interior-point methods, or branch-and-bound are used to compute optimal or near-optimal solutions; and (5) *interpretation and validation*, which ensure that the computed solution is feasible and meaningful in the application context [43]. While this pipeline is well established, recent advances in ML create new opportunities to improve formulation and solver efficiency [14, 50].

ML follows a data-driven lifecycle comprising: (1) *problem definition*, which clarifies the prediction or generation objective; (2) *data preparation*, including collection, cleaning, and feature construction; (3) *model training*, where models are optimized using appropriate architectures and algorithms; (4) *evaluation*, which assesses predictive performance using suitable metrics; and (5) *deployment and serving*, which integrate the trained model into production systems that must meet resource, scalability, and latency constraints [44, 169]. These later stages often involve system-level optimization challenges that align naturally with OR techniques [93, 169].

In this thesis, we focus on the following steps — *solution methods* and *math program formulation* steps within the OR process, alongside the *model serving* in ML lifecycle. This chapter provides background for these focused steps by first reviewing traditional methods for performing them, then highlighting recent advancements and improvements, including research that explores the synergies between the two fields

2.1 Solution Methods

Solution methods in OR aim to identify optimal or near-optimal solutions to mathematical programs, often formulated as optimization problems with well-defined objectives and constraints. This section reviews traditional solution methods for continuous and discrete optimization problems, followed by recent advancements where ML techniques enhance these methods. This section draws upon relevant content from our survey [54].

Traditional Solution Methods For continuous unconstrained problems, such as nonlinear optimization in engineering design, gradient-based methods like gradient descent [12], momentum methods [150], and adaptive algorithms including AdaGrad [49] and Adam [90] iteratively update solutions based on objective function gradients. For continuous constrained problems, common in resource allocation or network optimization, interior point methods navigate the interior of the feasible region using barrier functions, excelling in large-scale linear and nonlinear programs [157]. The Alternating Direction Method of Multipliers (ADMM) is suited for problems with separable objectives and coupled constraints, such as distributed machine learning or power grid management, by decomposing the problem into coordinated subproblems [20]. Discrete optimization, such as mixed-integer linear programs used in scheduling or logistics, employs branch-and-bound to partition the feasible region for integer solutions [97].

As Chapter 3 focuses on improving the initialization of the Simplex method, we provide its background in the following. The Simplex method is a classical algorithm for solving linear programs. Starting from an initial basis—which can be understood as a vertex of the feasible region (a formal definition is provided in Section 3.1)—the method iteratively pivots along adjacent vertices of the feasible polyhedron until an optimal solution is reached [41]. Textbooks typically assume the method starts from a feasible initial basis, which is reasonable because methods such as the two-stage method [120, 125] or big-M [70] can handle infeasibility. However, beyond feasibility, it remains unclear which initial basis leads to faster convergence.

In the literature, classical initial-basis strategies mainly exploit algebraic properties to ensure the first few pivot steps are efficient. For example, the logical basis [34, 14] is a simple and commonly used strategy in which all slack variables form the initial basis. This choice corresponds to an identity matrix for the basis constraint matrix, allowing efficient computation of its inverse. Consequently, it is the default option in many state-of-the-art solvers, such as HiGHS [82] and CPLEX [65]. However, as pivoting proceeds, the constraint matrix associated with the basis typically drifts away from the identity, slowing computations. When equality constraints appear in an LP problem, artificial variables must be introduced, which should not remain in the final basis. Motivated by this, CPLEX's Crash Basis strategy quickly guesses an initial basis that lowers the priority of artificial variables and encourages a sparse, well-conditioned basis. While this can improve efficiency, both logical and crash strategies may still produce a starting basis far from optimal. The Idiot Crash algorithm [60] aims to achieve a near-optimal solution by using the augmented Lagrangian method to generate a basis close to the optimum. However, the augmented Lagrangian method often converges slowly. Overall, traditional initial-basis strategies rely solely on information from a single LP instance and do not leverage potential correlations with historical LPs.

Once an initial basis is chosen, the method iteratively performs pivoting, which intuitively corresponds to moving from the current vertex to a neighboring vertex to improve the objective function. The selection of the entering variable, or pivoting strategy, is critical for the efficiency of the Simplex method. Classical strategies [41, 58] have been extensively studied. For example, the Dantzig rule [41] selects the edge that yields the largest overall decrease in the objective, while the steepest-edge rule [58] chooses the edge along which the objective decreases most rapidly *per unit* of movement, i.e., the "steepest" descent. Both the choice of the initial basis and the pivoting strategy can substantially affect convergence speed, motivating research into improving them.

ML-enhanced Solution Methods Recent advancements leverage ML to enhance the efficiency and scalability of OR solution methods, as surveyed in [54]. For gradient-based methods, recurrent neural networks, such as Long Short-Term Memory (LSTM) networks, learn adaptive update rules tailored to specific problem classes, while reinforcement learning (RL) optimizes steps by treating objective reductions as rewards, particularly for non-differentiable objectives [7, 101, 78]. In ADMM, reinforcement learning (RL) and graph neural networks (GNNs) improve penalty parameter tuning, enhancing convergence in applications like distributed optimal power flow [170, 86].

Branch-and-bound methods have been enhanced using imitation learning and GNNs to optimize variable and node selection, prioritizing paths toward optimal solutions and accelerating MIP problem solving [89, 95]. The application of GNNs to assist in solving MIP problems was first proposed in [64]. This work introduced a constraint-variable bipartite graph representation for mixed-integer LPs and designed a GNN model to learn a strong branching policy, accelerating MIP solvers. Building on this GNN-based framework for improving MIP solving, many subsequent works followed [45, 74, 131, 103]. Gupta et al. identify a missing "lookback" property in the trained GNN and incorporate it into training, yielding additional speedups for MIP solvers [74]. Qu et al. develop a reinforcement-learning method that builds on imitation learning [131]. To reduce reliance on high-end GPUs, Gupta et al. propose a CPU-friendly hybrid branching model that maintains competitive speedups [73]. Ding et al. introduce a tripartite graph representation for MIP and use GNNs to predict solution values for binary variables [45].

The Simplex method has been enhanced through GNN-based basis initialization [55] and RL-driven pivoting strategies [146]. To the best of our knowledge, our work [55], presented in Chapter 3, is the first to apply machine learning techniques to improve the initialization of the Simplex basis from both experimental and real-world problem-solving perspectives. Meanwhile, the theoretical foundation for the representation power of GNNs for linear programming (LP) was established in [31], demonstrating that, for any given LP, a GNN can be constructed to map the LP to its feasibility, boundedness, and an optimal solution. Despite this progress in leveraging GNNs to assist optimization solvers, there remains no prior work on initial basis selection for LPs in practical settings. The approach proposed in Chapter 3 addresses this gap by introducing a machine-learning-based strategy for selecting an initial Simplex basis.

For ML-enhanced pivoting, it is shown that pivoting performance depends strongly on problem instances, with different strategies yielding markedly different outcomes [146]. A reinforcement-learning agent switches between Dantzig's rule [41] and the steepest-edge rule [58]; at each Simplex iteration, the agent selects one of these two pivoting rules. Alternatively, [99] aims to learn a new pivoting strategy by applying the Monte Carlo Tree Search (MCTS) method. Their contribution includes (1) transforming the Simplex method into a pseudo-tree structure, (2) constructing appropriate reinforcement learning models, (3) finding the optimal pivot sequence under the guarantee of theory, and (4) providing a complete method for discovering multiple optimal pivot paths. They propose a novel imitative tree structure, SimplexPseudoTree, for exploring optimal pivot paths and construct reinforcement-learning models to determine the optimal pivot paths using the MCTS method. Both theoretical analysis and computational experiments demonstrated that the proposed MCTS rule effectively avoids redundant searches and identifies the shortest pivot paths, thereby reducing the number of iterations required to solve linear programs.

2.2 Math Program Formulation

Mathematical program formulation is a pivotal step in Operations Research (OR), where real-world problems are abstracted into structured mathematical models comprising decision variables, constraints, and objective functions [156, 132]. Traditionally, this process relies on manual abstraction, requiring significant domain expertise to define decision variables (e.g., production quantities, routing decisions), constraints (e.g., resource availability, demand requirements), and objective functions (e.g., cost, profit, or time) [40]. This manual approach can be error-prone and inefficient, particularly for poorly defined or large-scale problems, making automation highly desirable. In this section, we review recent advancements in mathematical program formulation and discuss how Machine Learning (ML), particularly large language models (LLMs), is transforming this process by automating the translation of natural language problem descriptions into formal mathematical representations.

Recent advancements One important category of improvements is automated reformulation techniques, which often utilize group theoretical methods to exploit structural patterns in optimization models [112, 17]. For example, symmetry detection identifies variables or constraints that are essentially equivalent, allowing the solver to avoid redundant computations. Constraint relaxation temporarily loosens certain restrictions to simplify the search space, while still guiding the solver toward feasible and optimal solutions. By leveraging these techniques, the effective complexity of the model is reduced, enabling solvers to find solutions more efficiently [112]. However, methods in this category primarily focus on simplifying existing formulations. In practice, effective model formulation often relies on the accumulated experience of OR practitioners. For example, in combinatorial and scheduling problems, practitioners carefully design assignment and ordering constraints, incorporate symmetry-breaking inequalities, and exploit problem structures such as flow conservation or network decompositions. This process involves defining decision variables, constraints, and objective functions. While crucial, it is often time-consuming and requires considerable expertise.

Large language models (LLMs) offer a way to break through this limitation. Recent advances in models such as ChatGPT [21] and LLaMA [149] have created new opportunities for automating model formulation by translating natural language descriptions directly into mathematical models [117, 116].

LLMs are trained through pre-training on large text corpora followed by fine-tuning with specific datasets, sometimes enhanced by reinforcement learning from human feedback [119, 178, 148]. These steps endow LLMs with emergent abilities such as contextual reasoning, coherent text generation, and mathematical problem solving. Formulating a mathematical program from a natural language description is not a well-defined task: there is no explicit set of rules or step-by-step procedure to follow. LLMs, however, combine language understanding, reasoning, and structured generation capabilities, which allow them to interpret ambiguous problem descriptions and generate formal optimization models. We reuse content from our survey [54] to demonstrate LLMs' ability at two levels: textbook-level problems and real-world problems.

LLMs' ability for textbook-level problems NL4OPT competition at NeurIPS 2022 [134] provides a benchmark for translating natural language into mathematical models. The dataset pairs problem descriptions with human-authored formulations, and declaration-level mapping accuracy is used as the evaluation metric. Table 2.1 reports the accuracy of several LLMs alongside the competition's winning submission. Results show that even without task-specific training, Code-Llama-34b-instruct achieves 65% accuracy, and fine-tuning Llama-2-13b-chat boosts performance to 82%. The competition winner, based on a fine-tuned BART model, achieves 90% on sub-task 2, but under the assumption of perfect entity extraction, making it an optimistic upper bound. These results demonstrate that LLMs are effective and

Index	**LLMs**	Model size	Finetuning dataset	**Acc.**
1	Llama-2-13b-chat	52 GB	Open domain	24%
2	Code-Llama-34b-instruct	136 GB	Programming/math domain	65%
3	Llama-2-70b-chat	280 GB	Open domain	37%
4	Llama-2-13b-chat (SFT)	52 GB	Open domain+NL4OPT	82%
5	NL4OPT winning submission	1 GB	NL4OPT train set	90%

Table 2.1: Declaration-level mapping accuracy (“Acc.") for different LLMs and the NL4OPT winning system.

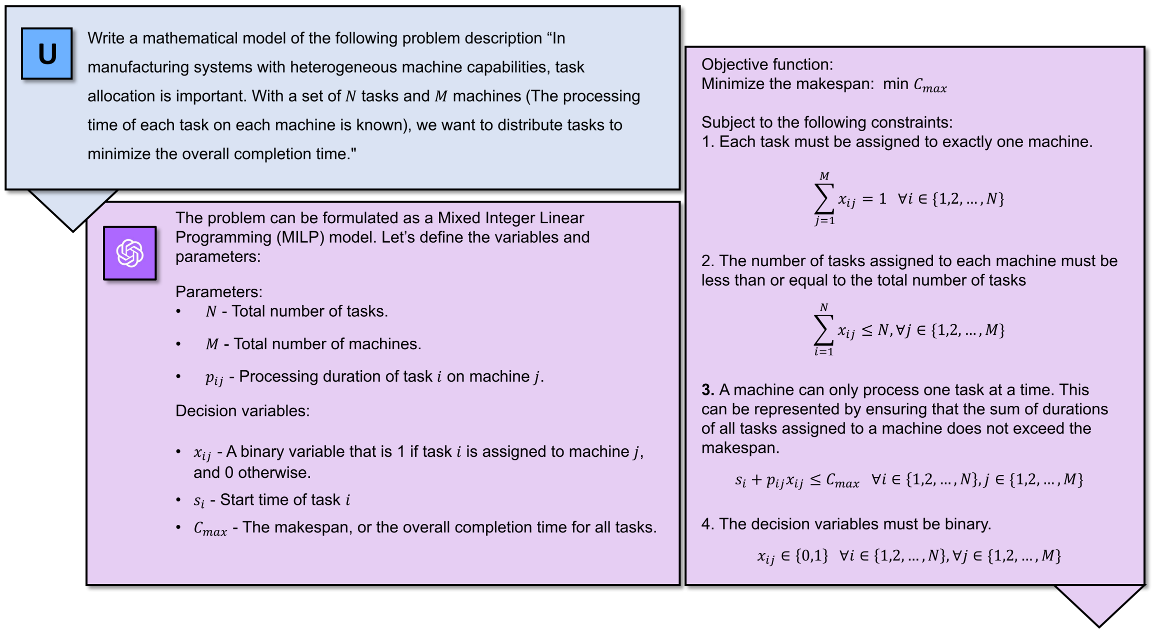
easy to apply to textbook-level modeling tasks, especially when fine-tuned with in-domain data.

LLMs' ability for real-world problems We assessed GPT-3.5 (version released December 2023) [2] on more complex tasks, including scheduling, bin packing, vehicle routing, portfolio optimization, and staff scheduling. Figure 2.1 illustrates an example for the Unrelated-Machine Scheduling Problem. The model correctly identifies sets, parameters, and decision variables, but also introduces errors such as redundant or missing constraints. For instance, it defines a start-time variable s_i without enforcing non-overlap of tasks and generates unnecessary constraints. Nevertheless, when provided with corrective feedback, the LLM iteratively improves its formulation, eliminating redundant variables and refining constraints. These results suggest that LLMs can provide a valuable starting point for OR experts by generating skeletal model structures. However, expert verification and refinement remain essential to ensure correctness, particularly for real-world and less common problems.

Overall, we conclude with three insights. First, LLMs achieve strong performance on textbook-style problems, where fine-tuning with in-domain data yields competitive accuracy. Second, larger model size alone does not ensure superior performance; the quality and relevance of fine-tuning data are more decisive. Finally, for real-world problems, LLMs facilitate rapid prototyping but require expert guidance for reliable formulations. Taken together, these findings highlight the potential of LLMs to accelerate OR model development while underscoring the necessity of human oversight.

2.3 Model Serving

Model serving refers to the deployment and runtime execution of trained machine learning (ML) models to handle inference requests in production. Traditional ML models, such as feedforward networks for tabular data or convolutional neural networks (CNNs) for image recognition, are relatively small and compute-bound. These models typically operate as stateless functions with deterministic execution: given a fixed-size input batch, they produce


U

Write a mathematical model of the following problem description "In manufacturing systems with heterogeneous machine capabilities, task allocation is important. With a set of N tasks and M machines (The processing time of each task on each machine is known), we want to distribute tasks to minimize the overall completion time."

The problem can be formulated as a Mixed Integer Linear Programming (MILP) model. Let's define the variables and parameters:

Parameters:

- N - Total number of tasks.
- M - Total number of machines.
- p_{ij} - Processing duration of task i on machine j.

Decision variables:

- x_{ij} - A binary variable that is 1 if task i is assigned to machine j, and 0 otherwise.
- s_i - Start time of task i
- C_{max} - The makespan, or the overall completion time for all tasks.

Objective function:
Minimize the makespan: min C_{max}

Subject to the following constraints:
1. Each task must be assigned to exactly one machine.

$$\sum_{j=1}^{M} x_{ij} = 1 \quad \forall i \in \{1,2,\dots,N\}$$

2. The number of tasks assigned to each machine must be less than or equal to the total number of tasks

$$\sum_{i=1}^{N} x_{ij} \leq N, \forall j \in \{1,2,\dots,M\}$$

3. A machine can only process one task at a time. This can be represented by ensuring that the sum of durations of all tasks assigned to a machine does not exceed the makespan.

$$s_i + p_{ij}x_{ij} \leq C_{max} \quad \forall i \in \{1,2,\dots,N\}, j \in \{1,2,\dots,M\}$$

4. The decision variables must be binary.

$$x_{ij} \in \{0,1\} \quad \forall i \in \{1,2,\dots,N\}, \forall j \in \{1,2,\dots,M\}$$

Figure 2.1: Example on the Unrelated-Machine Scheduling Problem.

outputs in a single forward pass without maintaining internal context. This regularity allows for straightforward batching, static placement, and predictable latency. As a result, serving systems for traditional ML models primarily focus on input batching and simple scheduling policies.

In contrast, LLMs introduce unique challenges due to their massive parameter sizes, autoregressive decoding, and reliance on key-value (KV) caching. A single LLM inference is typically split into two stages: Prefill (prompt processing) and Decode. The Decode phase is incremental—each token depends on prior context—which makes it difficult to batch requests efficiently. Naive batching introduces latency or compute waste due to padding. To overcome this, modern systems adopt continuous batching, where in-flight Decode requests are grouped dynamically at every token step to maintain high GPU utilization without sacrificing latency [94].

Beyond batching, LLM serving systems must now meet diverse service-level objectives (SLOs) such as latencies on different inference stages while adapting to bursty workloads. This has motivated the emergence of multi-SLO-aware, disaggregated, and elastic serving systems that optimize batching, placement, scheduling, and scaling decisions across single-node and distributed deployments.

Single-Node LLM Serving Early monolithic LLM serving systems, such as vLLM [94], exploit techniques like PagedAttention for efficient KV cache management but lack explicit SLO-awareness. FlexGen [142] operates on a single node, emphasizing CPU–GPU offloading to reduce memory footprint at the cost of real-time performance. More recently, Apt-

Serve [61] improves first-token latency by combining hybrid caching and adaptive batching strategies. It selectively caches either key-value attention states or hidden states based on resource availability, and dynamically composes batches according to request characteristics to reduce contention between short and long sequences. Sarathi-Serve [4] adopts chunked prefill scheduling to better saturate GPU compute while bounding Decode latency. Instead of processing the entire prompt in a single pass, it splits long prompts into smaller chunks, overlapping their execution with Decode steps from other requests. This improves pipeline utilization and reduces head-of-line blocking caused by large inputs.

Disaggregation [177] is an emerging deployment mode in LLM serving that decouples different components of the inference pipeline—such as Prefill and Decode stages, or compute and memory resources—across distributed nodes. This separation allows the system to assign resources more precisely based on each stage's compute and latency characteristics. For example, Decode stages often require fine-grained scheduling and low latency, while Prefill stages benefit from parallelism and high throughput. Recent systems such as EPD [145] further extend this paradigm to multimodal large language models (LMMs), where inputs may include both text and images. In such models, an additional Encode stage processes visual embeddings before the Prefill and Decode phases. EPD disaggregates these three stages—Encode, Prefill, and Decode—to isolate their distinct computational patterns, mitigating cross-stage interference and improving end-to-end throughput in multimodal inference pipelines. Building upon the disaggregation technique, Arrow [160] introduces adaptive scheduling mechanisms that dynamically reallocate instances between Prefill and Decode pools based on real-time workload statistics. This evolution from static to adaptive disaggregation reflects a key trend in modern LLM serving—toward elastic, stage-aware orchestration, where resource allocation, batching, and scheduling are jointly optimized to balance throughput, latency, and efficiency in large-scale deployments.

Distributed Multi-SLO LLM Serving At the cluster scale, distributed LLM serving frameworks introduce multi-node coordination and SLO-aware execution to handle high request volumes and heterogeneous workloads. Frameworks such as Llumnix [173] migrate priority requests across instances to reduce latency tails, while USHER [144] incorporates interference-aware GPU sharing. SpotServe [114] resumes preempted workloads at token granularity on spot GPUs. Scheduling heuristics in ELIS [33], QLM [124], and SCOOT [32] exploit predictions of remaining output lengths or SLO-aware objective functions to prioritize requests. DynaServe [139] introduces micro-request abstraction for adaptive Prompt /Decode balancing, while CoCoServe [158] supports fine-grained module-level replication and migration for elastic scaling.

Advanced strategies integrate scheduling, placement, and scaling. PD-Multiplexing [39] improves throughput by multiplexing multiple Decode requests on shared GPUs, leveraging the low compute intensity of decoding to increase GPU utilization. Proteus [5] dynamically trades model accuracy and precision to balance throughput and latency. PolyServe [181] and SLOs-Serve [29] incorporate multi-SLO constraints into resource placement, while HAS-GPU [71] enables fine-grained vertical partitioning with hybrid autoscaling. Collectively, these integrated designs illustrate how multi-node coordination, phase decoupling, and continuous execution can jointly optimize throughput, tail latency, and SLO adherence.

Operations Research for Enhancing Model Serving OR techniques provide a principled framework to optimize resource allocation, scheduling, and scaling in model serving, particularly under uncertainty and diverse SLOs. By modeling serving as a constrained optimization problem, OR enables strategies such as priority-based dispatch, queue reordering, and joint placement of requests across heterogeneous resources. In the context of LLM serving, OR methods have been applied to formulate linear programs for SLO-aware dispatch [123], and solve bin-packing problems for joint placement and autoscaling [118]. Specifically, Aladdin [118] adopts a best-fit bin-packing algorithm to jointly optimize request placement and autoscaling decisions. Beyond single-node optimization, OR can coordinate scheduling, placement, and scaling across distributed clusters, adapting to bursty workloads, heterogeneous GPU capacities, and multi-stage (Prefill/Decode) computation. The overarching objective is to enhance throughput, reduce latency, and improve GPU utilization—making OR a key enabler of efficient, multi-SLO-aware LLM serving.

Chapter 3

Smart Initial Basis Selection for Linear Programs

Linear program (LP) is a cornerstone of optimization in many industrial and scientific applications, and the Simplex method remains one of the most widely used techniques for solving LP problems efficiently. Despite its practical success, the efficiency of the Simplex method heavily depends on the choice of the initial basis, and traditional rule-based strategies often fail to consistently improve performance across similar LP instances. In this study [55], we focus on a learning-based approach for selecting advanced initial bases, leveraging graph neural networks to capture the relationship between LP problems and their optimal solutions.

3.1 Preliminaries on Linear Programs

We consider the following standard format of LP with m constraints and m decision variables ($m \leq n$).

$$\begin{array}{rllll} \min\limits_{\boldsymbol{x}\in\mathbb{R}^n, \boldsymbol{s}\in\mathbb{R}^m} & \boldsymbol{c}^\top \boldsymbol{x} & & & \\ \text{s.t.} & \mathbf{A}\boldsymbol{x} & = \boldsymbol{s} & & \\ & \boldsymbol{\ell}^x & \leq \boldsymbol{x} & \leq & \boldsymbol{u}^x \\ & \boldsymbol{\ell}^s & \leq \boldsymbol{s} & \leq & \boldsymbol{u}^s \end{array} \tag{P}$$

where $\mathbf{A} \in \mathbb{R}^{m\times n}$ is the constraint matrix, $\boldsymbol{c} \in \mathbb{R}^n$ is the cost vector, $\boldsymbol{x} \in \mathbb{R}^n$ is the decision variables, $\boldsymbol{s} \in \mathbb{R}^m$ is the constraint variables, $\boldsymbol{\ell}^x \in \mathbb{R} \cup \{-\infty\}^n$ and $\boldsymbol{u}^x \in \mathbb{R} \cup \{+\infty\}^n$ are lower and upper bounds for $\boldsymbol{x}$, and $\boldsymbol{\ell}^s \in \mathbb{R} \cup \{-\infty\}^m$ and $\boldsymbol{u}^s \in \mathbb{R} \cup \{+\infty\}^m$ are lower and upper bounds for $\boldsymbol{s}$. This formulation agrees with the input formulation for many LP solvers such as HiGHS [82], OptVerse [83], CPLEX [38], and Gurobi [75].

Next, we formally introduce the concept of basis and solution for an LP problem.

Definition 1. *Given two index sets $\mathcal{B}_x \subset [n]$ and $\mathcal{B}_s \subseteq [m]$, the tuple $(\mathcal{B}_x, \mathcal{B}_s)$ is called a* ***basis*** *for the LP problem* (P) *if $|\mathcal{B}_x| + |\mathcal{B}_s| = m$ and the matrix $[\mathbf{A}_{\mathcal{B}_x} \;\; -\mathbf{I}^m_{\mathcal{B}_s}]$ is non-singular,* *where $\mathbf{A}_{\mathcal{B}_x}$ denotes the submatrix of $\mathbf{A}$ consisting of the columns indexed by $\mathcal{B}_x$, and $\mathbf{I}^m_{\mathcal{B}_s}$ denotes the submatrix of the $m \times m$ identity matrix $\mathbf{I}^m$ corresponding to the columns indexed by $\mathcal{B}_s$. Let $(\mathcal{N}_x, \mathcal{N}_s) = ([n] \setminus \mathcal{B}_x,\ [m] \setminus \mathcal{B}_s)$. A tuple $(\boldsymbol{x}^{\mathcal{B}}, \boldsymbol{s}^{\mathcal{B}}) \in \mathbb{R}^n \times \mathbb{R}^m$ is called a **basic solution** for* (P) *with respect to the basis $(\mathcal{B}_x, \mathcal{B}_s)$ if $\mathbf{A}\boldsymbol{x}^{\mathcal{B}} = \boldsymbol{s}^{\mathcal{B}}$ and for any $i \in \mathcal{N}_x$ and $j \in \mathcal{N}_s$, we have*

$$x_i^{\mathcal{B}} = \begin{cases} l_i^x & \text{if } u_i^x = +\infty \\ u_i^x & \text{if } l_i^x = -\infty \\ l_i^x \text{ or } u_i^x & \text{otherwise} \end{cases} \quad \text{and} \quad s_j^{\mathcal{B}} = \begin{cases} l_j^s & \text{if } u_j^s = +\infty \\ u_j^s & \text{if } l_j^s = -\infty \\ l_j^s \text{ or } u_j^s & \text{otherwise.} \end{cases}$$

*Moreover, if $\boldsymbol{\ell}^x \leq \boldsymbol{x}^{\mathcal{B}} \leq \boldsymbol{u}^x$ and $\boldsymbol{\ell}^s \leq \boldsymbol{s}^{\mathcal{B}} \leq \boldsymbol{u}^s$, then $(\boldsymbol{x}^{\mathcal{B}}, \boldsymbol{s}^{\mathcal{B}})$ is called a **basic feasible solution** for* (P)*.*

To distinguish the different components within a solution vector, we introduce a further notation. In particular, we separate the solution vector into *basic* and *nonbasic* components and denote by $\boldsymbol{x}_{\mathcal{B}}$ (similarly $\boldsymbol{s}_{\mathcal{B}}$) the subvector of $\boldsymbol{x}$ (respectively $\boldsymbol{s}$) containing only the entries indexed by $\mathcal{B}$. Likewise, $\mathbf{A}_{\mathcal{B}}$ denotes the submatrix of $\mathbf{A}$ formed by the columns indexed by $\mathcal{B}$.

Using this notation, the relationship between a basis and its corresponding solution can be expressed as follows. Given a basis $(\mathcal{B}_x, \mathcal{B}_s)$ and the values of the nonbasic components $(\boldsymbol{x}^{\mathcal{B}}_{\mathcal{N}_x}, \boldsymbol{s}^{\mathcal{B}}_{\mathcal{N}_s})$, the remaining values in the basic solution $(\boldsymbol{x}^{\mathcal{B}}, \boldsymbol{s}^{\mathcal{B}})$ are uniquely determined by

$$\begin{bmatrix} \boldsymbol{x}^{\mathcal{B}}_{\mathcal{B}_x} \\ \boldsymbol{s}^{\mathcal{B}}_{\mathcal{B}_s} \end{bmatrix} = [\mathbf{A}_{\mathcal{B}_x} \quad -\mathbf{I}^m_{\mathcal{B}_s}]^{-1} \left(\mathbf{I}^m_{\mathcal{N}_s} \boldsymbol{s}^{\mathcal{B}}_{\mathcal{N}_s} - \mathbf{A}_{\mathcal{N}_x} \boldsymbol{x}^{\mathcal{B}}_{\mathcal{N}_x} \right),$$

Then, we briefly introduce the primal Simplex method for solving problem (P). The primal Simplex method is initialized with a primal feasible basis. If the basis is not dual feasible, it selects a dual infeasible index p to enter the basis and a leaving index $q \in \mathcal{B}_x \cup \mathcal{B}_s$ such that the updated basis $(\mathcal{B}_x, \mathcal{B}_s) \setminus \{q\} \cup \{p\}$ remains primal feasible. This step is called a *pivot*, which corresponds to moving from the previous basis to the next candidate basis. Similarly, the dual Simplex method is initialized with a dual feasible basis and iteratively updates the basis to maintain dual feasibility.

In summary, the Simplex method is an algorithm that starts with an initial basis $(\mathcal{B}_x^{(0)}, \mathcal{B}_s^{(0)})$, obtains a basic solution $(\boldsymbol{x}^{(0)}, \boldsymbol{s}^{(0)})$, and then iteratively visits candidate bases and corresponding basic solutions until it encounters an optimal basic feasible solution.

$$(\mathcal{B}_x^{(0)}, \mathcal{B}_s^{(0)}) \to (\boldsymbol{x}^{(0)}, \boldsymbol{s}^{(0)}) \to (\mathcal{B}_x^{(1)}, \mathcal{B}_s^{(1)}) \to (\boldsymbol{x}^{(1)}, \boldsymbol{s}^{(1)}) \quad \ldots$$

We refer interested readers to [113] for a more detailed description of the Simplex method.

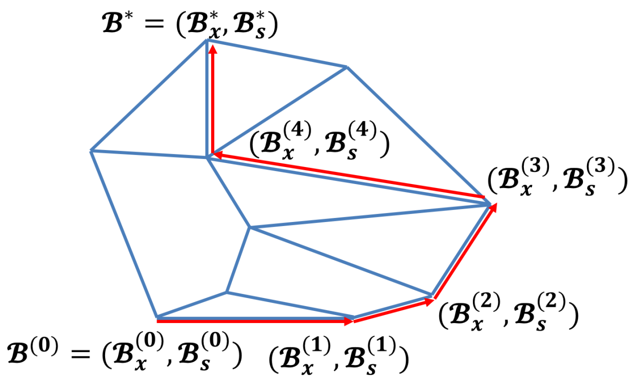

$\boldsymbol{\mathcal{B}}^* = (\boldsymbol{\mathcal{B}}_x^*, \boldsymbol{\mathcal{B}}_s^*)$
$(\boldsymbol{\mathcal{B}}_x^{(4)}, \boldsymbol{\mathcal{B}}_s^{(4)})$
$(\boldsymbol{\mathcal{B}}_x^{(3)}, \boldsymbol{\mathcal{B}}_s^{(3)})$
$(\boldsymbol{\mathcal{B}}_x^{(2)}, \boldsymbol{\mathcal{B}}_s^{(2)})$
$\boldsymbol{\mathcal{B}}^{(0)} = (\boldsymbol{\mathcal{B}}_x^{(0)}, \boldsymbol{\mathcal{B}}_s^{(0)})$
$(\boldsymbol{\mathcal{B}}_x^{(1)}, \boldsymbol{\mathcal{B}}_s^{(1)})$

Figure 3.1: Illustration of the Simplex algorithm. It starts with an initial basis $(\mathcal{B}_x^{(0)}, \mathcal{B}_s^{(0)})$, and routinely pivots to a neighbouring basis with improvement till it reaches an optimal basis $(\mathcal{B}_x^*, \mathcal{B}_s^*)$.

Although the Simplex method is one of the most widely adopted algorithms for solving LP problems, the theoretical guarantee of its performance is actually weak. It is shown that the Simplex method has exponential time complexity in the worst case [91]. Moreover, many empirical results demonstrate that the performance of the Simplex method depends largely on the selection of the initial basis [15, 80, 140].

3.2 GNN Model for Initial Basis Selection

In this section, we describe our approach for learning a mapping from a linear programming (LP) instance to a valid basis. Algorithm 1 presents the overall procedure, which consists of an inference stage and a training stage.

In the inference stage, given an LP instance, we first convert it into a bipartite graph representation. A graph neural network (GNN) then predicts, for each primal and slack variable, its basis status. Based on these predictions, we construct a candidate basis and apply post-processing steps to ensure its validity.

In the training stage, optimal bases obtained from a solver provide supervision. From these, we derive per-variable labels and train the GNN using a cross-entropy loss.

We describe the inference formulation in Section 3.2.1, provide an overview of the training stage in Section 3.2.2, detail the GNN architecture and training design in Section 3.2.3, and detail the post-processing steps (i.e., the basis inference and adjustment procedure) in Section 3.2.4.

3.2.1 Overview of Inference Stage

In the inference stage, we leverage a learned mapping $f(\theta; \cdot)$ such that, given an LP instance P with n variables and m constraints, it produces

$$f(\theta; P) = \{\mathbf{p}_{x,i} \in \Delta^3, \mathbf{p}_{s,j} \in \Delta^3 \mid i \in [n],\ j \in [m]\},$$

where θ denotes the set of learnable parameters in f, and $\mathbf{p}_{x,i}$ and $\mathbf{p}_{s,j}$ represent the probability distributions over the labels of the corresponding variables and slacks, respectively. Here, Δ^d denotes the d-dimensional simplex, i.e., $\Delta^d = \{p \in \mathbb{R}^d_+ \mid \sum_{i=1}^d p_i = 1\}$.

Concretely, we define

$$\begin{aligned} \mathbf{p}_{x,i} &= [\mathbb{P}(x_i = l_i^x),\ \mathbb{P}(l_i^x < x_i < u_i^x),\ \mathbb{P}(x_i = u_i^x)]^\top, \\ \mathbf{p}_{s,j} &= \left[\mathbb{P}(s_j = l_j^s),\ \mathbb{P}(l_j^s < s_j < u_j^s),\ \mathbb{P}(s_j = u_j^s)\right]^\top. \end{aligned} \tag{3.1}$$

From the above probabilities, we first generate a candidate basis $(\mathcal{B}_x, \mathcal{B}_s)$ according to the probabilities given by $f(\theta; P)$, *i.e.*,

$$(\mathcal{B}_x, \mathcal{B}_s) \in \underset{|\mathcal{B}_x| + |\mathcal{B}_s| = m}{\operatorname{argmax}} \left[\prod_{i \in \mathcal{B}_x} \mathbf{p}_{x,i}[2] \prod_{j \in \mathcal{B}_s} \mathbf{p}_{s,j}[2] \right].$$

We then adjust the basis to make it valid, *i.e.*, ensuring that the matrix $[\mathbf{A}_{\mathcal{B}_x} \quad -\mathbf{I}^m_{\mathcal{B}_s}]$ is non-singular (Definition 1).

In summary, the inference stage consists of two components: a GNN-based prediction module and a post-processing step that ensures the basis is valid.

3.2.2 Overview of Training Stage

In the training stage, we aim to learn the mapping $f(\theta; \cdot)$ from previously solved LP instances. Given a dataset

$$\mathcal{D} = \left\{ [(P^k), (\boldsymbol{x}^k, \boldsymbol{s}^k)] \right\}_{k=1}^K,$$

each sample consists of an LP instance P^k and a corresponding optimal basic feasible solution $(\boldsymbol{x}^k, \boldsymbol{s}^k)$.

For each k, the LP instance P^k is first transformed into a bipartite graph representation. Based on the associated optimal basic feasible solution $(\boldsymbol{x}^k, \boldsymbol{s}^k)$, we assign to every primal and slack variable a three-class label indicating its basis status: *lower bound*, *basic*, or *upper bound*.

A graph neural network (GNN) is then trained to predict these basis-status labels. The model parameters θ are optimized using a weighted cross-entropy loss. To mitigate potential class imbalance among the three basis categories, label-dependent weights are incorporated into the loss formulation.

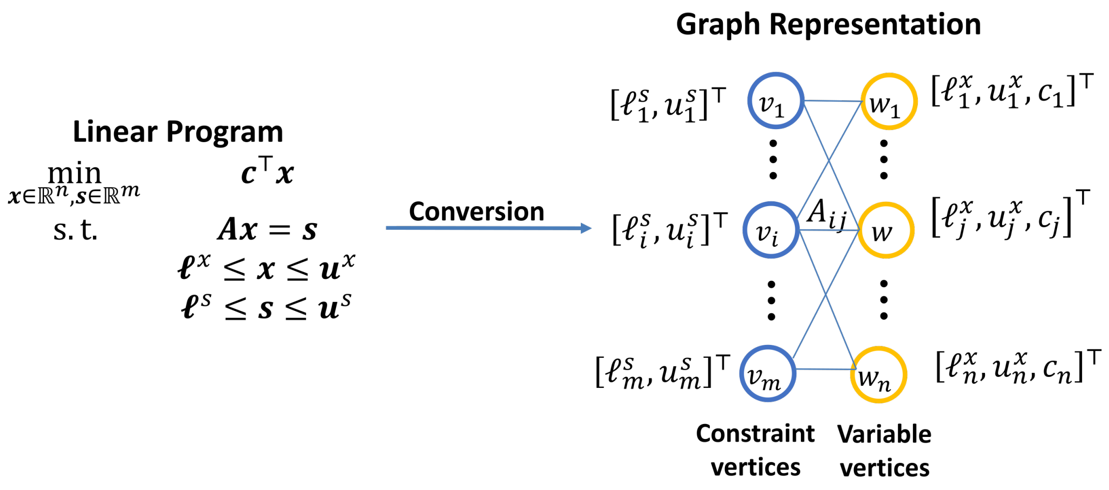


Figure 3.2: Represent the LP problem as a weighted bipartite graph.

3.2.3 GNN Model Design

In this section, we describe how to represent each (P^k) as a graph, construct labels from $(\boldsymbol{x}^k, \boldsymbol{s}^k)$, and detail the message-passing GNN architecture and its training loss.

Graph representation Inspired by [64], who suggests representing a mixed-integer linear problem as a weighted bipartite graph, we similarly adopt a graph-based representation for our task. As illustrated in Figure 3.2, a weighted bipartite graph $\mathcal{G} = (V, W, E)$ consists of two disjoint vertex sets V and W and a set E of weighted edges, where each edge connects exactly one vertex in V with one vertex in W. Specifically, when representing an LP as a weighted bipartite graph:

- The variable vertex set V consists of n nodes, each representing a decision variable x_j. We use the symbol v_j to refer to the j-th variable node. Each variable node is associated with an attribute vector $\mathbf{v}_j \in \mathbb{R}^p$ that encodes information about x_j, where p denotes the number of attributes. In its simplest form, as illustrated in Figure 3.2, this vector includes the variable's lower and upper bounds and objective coefficient: $[\ell_j^x, u_j^x, c_j]^\top$. We can further enrich $\mathbf{v}_j$ with additional structural and numerical information. This may include the sparsity of the corresponding column $\mathbf{A}_{:j}$ in the constraint matrix, as well as the similarity between $\mathbf{A}_{:j}$ and the bound vectors $(\boldsymbol{\ell}^s, \boldsymbol{u}^s)$. In this way, each variable node captures both structural information from the constraint matrix and numerical characteristics from the optimization model. A detailed description of these attributes is provided in Table 3.1.

- The constraint vertex set W contains m nodes. We use the symbol w_i to represent the i-th constraint node. Each constraint node is associated with an attribute vector $\mathbf{w}_i \in \mathbb{R}^q$ that encodes information about w_j, where q denotes the number of attributes.

Attribute index	Constraint node j	Variable node i
1	$\langle \mathbf{A}_{j:}, \mathbf{c} \rangle$	c_j
2	$nnz(\mathbf{A}_{j:})/n$	$nnz(\mathbf{A}_{:i})/m$
3	$\langle \mathbf{A}_{j:}, \boldsymbol{\ell}^x \rangle$	$\langle \boldsymbol{\ell}^s, \mathbf{A}_{:i} \rangle$
4	$\langle \mathbf{A}_{j:}, \boldsymbol{u}^x \rangle$	$\langle \boldsymbol{\ell}^u, \mathbf{A}_{:i} \rangle$
5	$\begin{cases} l_j^s & \text{if } l_j^s \neq -\infty \\ 0 & \text{else} \end{cases}$	$\begin{cases} l_i^x & \text{if } l_i^x \neq -\infty \\ 0 & \text{else} \end{cases}$
6	$\begin{cases} 0 & \text{if } l_j^s \neq -\infty \\ -1 & \text{else} \end{cases}$	$\begin{cases} 0 & \text{if } l_i^x \neq -\infty \\ -1 & \text{else} \end{cases}$
7	$\begin{cases} u_j^s & \text{if } u_j^s \neq \infty \\ 0 & \text{else} \end{cases}$	$\begin{cases} u_i^x & \text{if } u_i^x \neq \infty \\ 0 & \text{else} \end{cases}$
8	$\begin{cases} 0 & \text{if } u_j^s \neq \infty \\ 1 & \text{else} \end{cases}$	$\begin{cases} 0 & \text{if } u_i^x \neq \infty \\ 1 & \text{else} \end{cases}$

Table 3.1: Attributes of each constraint node j and variable node i, used as input features for the GNN model. Here, $\langle \cdot, \cdot \rangle$ denotes the cosine similarity between two vectors. For attributes 5–8, since the bounds can be infinite, additional tag dimensions are included to indicate their presence, ensuring numerical stability.

In its simplest form, as illustrated in Figure 3.2, the attribute vector for a constraint node includes the constraint's lower and upper bounds, $[\ell_i^s, u_i^s]^\top$. A more enriched version of the attribute vector includes the similarity between the row $\mathbf{A}_{i:}$ and the objective vector $\mathbf{c}$, the similarity between $\mathbf{A}_{i:}$ and the variable bound vectors $(\boldsymbol{\ell}^x, \boldsymbol{u}^x)$, and the sparsity of $\mathbf{A}_{i:}$.

- The edge set E is defined by the nonzero entries of the constraint matrix $\mathbf{A}$. Specifically, if $A_{ji} \neq 0$, then there exists an edge (v_i, w_j) connecting variable node v_i and constraint node w_j, with the edge attribute given by the corresponding matrix entry A_{ji}. Formally,
$$E_{(v_i,w_j)} = A_{ji}, \quad \forall (i,j) \in [n] \times [m].$$

Note that a major advantage of representing a linear program (LP) as a weighted bipartite graph is *permutation equivalence*. This concept refers to the equivalence of two LP instances when the decision variables, cost vector, bound vectors, and columns of the constraint matrix are permuted in the same order. In the graph context, this property is formally referred to as *permutation equivariance*, meaning that predictions at the node level are equivariant to such permutations. Specifically, if we permute two decision variables in the input LP, the underlying problem remains unchanged, and thus we expect the model's output to be invariant under these permutations.

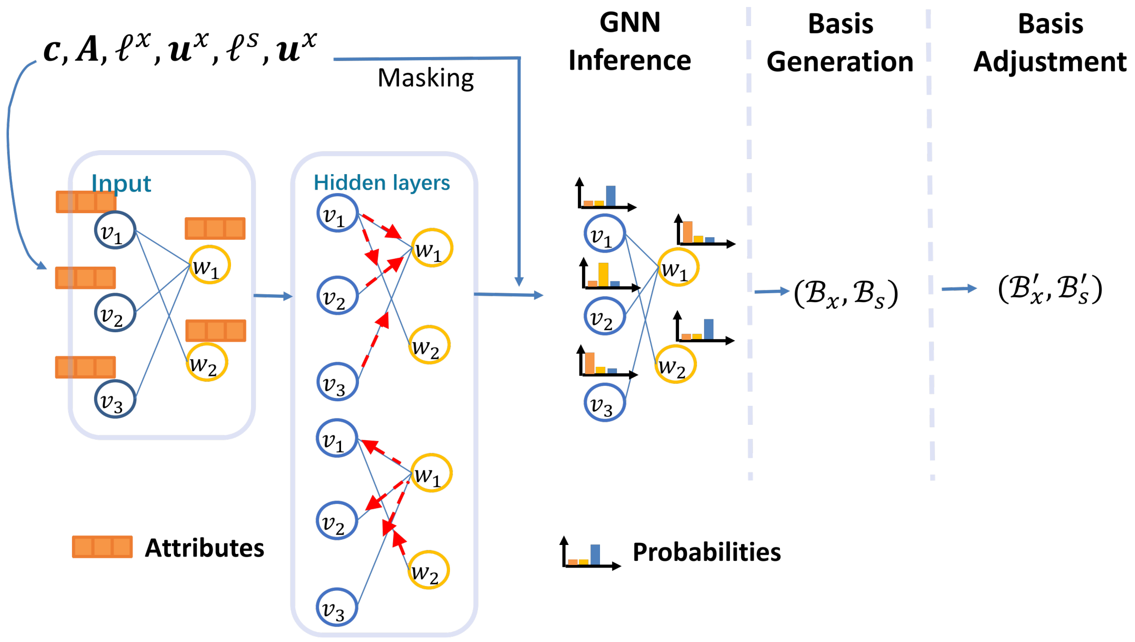

$\boldsymbol{c}, \boldsymbol{A}, \ell^x, \boldsymbol{u}^x, \ell^s, \boldsymbol{u}^x$
Masking
GNN Inference
Basis Generation
Basis Adjustment
Input
Hidden layers
v_1
v_2
v_3
w_1
w_2
$(\mathcal{B}_x, \mathcal{B}_s)$
$(\mathcal{B}'_x, \mathcal{B}'_s)$
Attributes
Probabilities

Figure 3.3: Overall procedure of the inference step.

Following this bipartite graph representation, for any problem instance (P), we can uniquely transform it into a weighted bipartite graph, *i.e.*, $(P) \to \mathcal{G} = (V, W, E)$.

Label construction. A valid initial basis for the Simplex method requires the following information:

- A valid basis $(\mathcal{B}_x, \mathcal{B}_s)$.
- For any $i \in \mathcal{N}_x$, whether $x_i^{\mathcal{B}} = \ell_i^x$ or $x_i^{\mathcal{B}} = u_i^x$.
- For any $j \in \mathcal{N}_s$, whether $s_j^{\mathcal{B}} = \ell_j^s$ or $s_j^{\mathcal{B}} = u_j^s$.

For this purpose, given any optimal basic feasible solution $(\boldsymbol{x}, \boldsymbol{s})$, we transform it into $n+m$ one-hot labels

$$\left\{ \boldsymbol{y}_{x,i}, \boldsymbol{y}_{s,j} \in \mathsf{OneHot}(3) \mid i \in [n], j \in [m] \right\}.$$

Specifically,

$$\boldsymbol{y}_{x,i} = \begin{cases} [1\ \ 0\ \ 0]^\top & \text{if } x_i = \ell_i^x \\ [0\ \ 1\ \ 0]^\top & \text{if } \ell_i^x < x_i < u_i^x \\ [0\ \ 0\ \ 1]^\top & \text{if } x_i = u_i^x \end{cases} \quad \text{and} \quad \boldsymbol{y}_{s,j} = \begin{cases} [1\ \ 0\ \ 0]^\top & \text{if } s_j = \ell_j^s \\ [0\ \ 1\ \ 0]^\top & \text{if } \ell_j^s < s_j < u_j^s \\ [0\ \ 0\ \ 1]^\top & \text{if } s_j = u_j^s \end{cases} \tag{3.2}$$

Message-passing functions. We adopt the standard message-passing framework for the graph neural network [141]. A graph neural network with L message-passing steps will be learned. The learnable message-passing functions are denoted as

$$f_l^V(\theta_l^V;\cdot):\mathbb{R}^{d_{l-1}^V}\times\mathbb{R}^{d_{l-1}^W}\to\mathbb{R}^{d_l^V}\ \text{ and}$$
$$f_l^W(\theta_l^W;\cdot):\mathbb{R}^{d_{l-1}^W}\times\mathbb{R}^{d_{l-1}^V}\to\mathbb{R}^{d_l^W}\ \text{ for } l\in[L],$$

where $\{\theta_l^V,\theta_l^W \mid l\in[L]\}$ are the learnable parameters, $d_0^V=p$, $d_0^W=q$ and $d_L^V=d_L^W=3$. The node embeddings will be updated via these message-passing functions, *i.e.*, for any $i\in[n]$, $j\in[m]$ and $l\in[L]$,

$$\begin{aligned}\mathbf{v}_i^l &= f_l^V\left(\theta_l^V;\mathbf{v}_i^{l-1},\ \sum_{j=1}^m E_{(v_i,w_j)}\mathbf{w}_j^{l-1}\right),\\ \mathbf{w}_j^l &= f_l^W\left(\theta_l^W;\mathbf{w}_j^{l-1},\ \sum_{i=1}^n E_{(v_i,w_j)}\mathbf{v}_i^{l-1}\right),\end{aligned}\tag{3.3}$$

where the 0-step embeddings $\mathbf{v}_i^0$ and $\mathbf{w}_j^0$ are initialized as the input features $\mathbf{v}_i$ and $\mathbf{w}_j$, respectively.

Knowledge-based masking. After the L message passing steps, we want to predict the label for each node based on its current feature. Namely, we design mappings.

$$\mathbf{v}_i^L\to\mathbf{p}_{x,i}\in\Delta^3\ \text{ and }\ \mathbf{w}_j^L\to\mathbf{p}_{s,j}\in\Delta^3,\quad\forall i\in[n],j\in[m],$$

such that $\mathbf{p}_{x,i},\mathbf{p}_{s,j}$ are the probabilities defined in (3.1).

It is important that these resulting probabilities respect the feasibility of non-basic entries. Formally, for any variable x_i, if $\ell_i^x=-\infty$ or $u_i^x=+\infty$, then the corresponding probability should be zero, *i.e.*,

$$\mathbb{P}(x_i=\ell_i^x)=0\ \text{ or }\ \mathbb{P}(x_i=u_i^x)=0.$$

A similar requirement exists for constraint variables.

To satisfy these requirements, we adopt the knowledge-based masking technique [56], where the key idea is to mask out probability entries based on the problem knowledge. Specifically, we define

$$\mathbf{p}_{x,i}=\mathsf{softmax}(\mathbf{v}_i^L+\mathbf{h}_{x,i})\ \text{ and }\ \mathbf{p}_{s,j}=\mathsf{softmax}(\mathbf{w}_j^L+\mathbf{h}_{s,j}),$$

where the softmax mapping is defined by $\mathsf{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^{d}\exp(z_j)}$, $\forall i \in [d]$ and

$$\mathbf{h}_{x,i} = \begin{cases} [-\infty \;\; 0 \;\; 0]^\top & \text{if } \ell_i^x = -\infty \\ [0 \;\; 0 \;\; -\infty]^\top & \text{if } u_i^x = +\infty \\ [0 \;\; 0 \;\; 0]^\top & \text{otherwise} \end{cases} \quad \text{and} \quad \mathbf{h}_{s,j} = \begin{cases} [-\infty \;\; 0 \;\; 0]^\top & \text{if } \ell_j^s = -\infty \\ [0 \;\; 0 \;\; -\infty]^\top & \text{if } u_j^s = +\infty \\ [0 \;\; 0 \;\; 0]^\top & \text{otherwise.} \end{cases}$$

Training loss Finally, we define the loss function, where we use the cross-entropy function to measure the mismatch between the resulting probability and the ground truth label for every decision and constraint variable. Let $\theta = \{\theta_l^V, \theta_l^W \mid 1 \le l \le L\}$ denote the set of all learnable parameters. The loss function is defined as

$$\mathcal{L}_{\mathcal{D}}(\theta) = \frac{1}{K}\sum_{k=1}^{K} \ell\left(\theta; (P^k), (\boldsymbol{x}^k, \boldsymbol{s}^k)\right),$$

where the per-sample loss is

$$\ell\left(\theta; (P), (\boldsymbol{x}, \boldsymbol{s})\right) = \frac{1}{m+n}\left[\sum_{i=1}^{n} \ell_{\mathsf{CE}}(\mathbf{p}_{x,i}, \boldsymbol{y}_{x,i}) + \sum_{j=1}^{m} \ell_{\mathsf{CE}}(\mathbf{p}_{s,j}, \boldsymbol{y}_{s,j})\right].$$

Here, the cross-entropy function is defined as $\ell_{\mathsf{CE}}(\mathbf{p}, \boldsymbol{y}) = -\sum_{i=1}^{d} y_i \log(p_i)$, and the predicted probabilities $\{\mathbf{p}_{x,i}, \mathbf{p}_{s,j} \mid 1 \le i \le n, 1 \le j \le m\}$ are obtained from $f(\theta; (P))$.

Since the basis labels can be imbalanced, *e.g.*, only a few slack variables may be basic, we introduce label-dependent weights $\alpha(y_{x,i})$ and $\alpha(y_{s,j})$ into the training loss:

$$\ell(\theta; (P), (\boldsymbol{x}, \boldsymbol{s})) = \frac{1}{m+n}\left[\sum_{i=1}^{n} \alpha(\boldsymbol{y}_{x,i})\ \ell_{\mathsf{CE}}(\mathbf{p}_{x,i}, \boldsymbol{y}_{x,i}) + \sum_{j=1}^{m} \alpha(\boldsymbol{y}_{s,j})\ \ell_{\mathsf{CE}}(\mathbf{p}_{s,j}, \boldsymbol{y}_{s,j})\right],$$

where the weights are defined as the inverse frequency of each label:

$$\alpha(\boldsymbol{y}_{x,i}) = \frac{1}{\sum_{i' \in [n]} \mathbf{1}\{\boldsymbol{y}_{x,i'} = \boldsymbol{y}_{x,i}\}}, \qquad \alpha(\boldsymbol{y}_{s,j}) = \frac{1}{\sum_{j' \in [m]} \mathbf{1}\{\boldsymbol{y}_{s,j'} = \boldsymbol{y}_{s,j}\}}.$$

Here, $\mathbf{1}\{\cdot\}$ denotes the indicator function.

3.2.4 Basis Inference

In this section, we show how to infer a valid basis from the probabilities predicted by the learned mapping $f(\theta; \cdot)$. The overall procedure of the inference step is shown in Figure 3.3. Given an LP instance (P), recall that the predicted probabilities from $f(\theta; (P))$ are $\{\mathbf{p}_{x,i}, \mathbf{p}_{s,j} \mid i \in [n], j \in [m]\}$.

Basis generation First, we select basis as the indices corresponding to the top-m predicted values for $\mathbb{P}(\ell_i^x < x_i < u_i^x)$ and $\mathbb{P}(\ell_j^s < s_j < u_j^s)$, *i.e.*,

$$(\mathcal{B}_x, \mathcal{B}_s) \in \underset{|\mathcal{B}_x|+|\mathcal{B}_s|=m}{\operatorname{argmax}} \prod_{i \in \mathcal{B}_x} p_{x,i}[2] \prod_{j \in \mathcal{B}_s} p_{s,j}[2].$$

Basis adjustment However, $(\mathcal{B}_x, \mathcal{B}_s)$ from the last step may not be a valid basis, as the matrix $[\mathbf{A}_{\mathcal{B}_x} \quad -\mathbf{I}^m_{\mathcal{B}_s}]$ may be singular (Definition 1). Thus, we adjust $(\mathcal{B}_x, \mathcal{B}_s)$ to make it a valid basis. Our adjustment approach is inspired by a basis repair procedure described in [16]. We try to factor the matrix $[\mathbf{A}_{\mathcal{B}_x} \quad -\mathbf{I}^m_{\mathcal{B}_s}]$ [15]. Note that this factorization does not incur additional computational complexity, as it has to be done during the Simplex method. During the factorization, if we encounter a column whose pivot value is smaller than some fixed tolerance, then we will remove this column by eliminating the corresponding index from $(\mathcal{B}_x, \mathcal{B}_s)$. After the factorization, if the basis is incomplete, *i.e.*, $|\mathcal{B}_x| + |\mathcal{B}_s| < m$, then we will fill it by adding the non-selected indices according to the predicted probabilities. The factorization is then attempted again until a complete and successfully factored basis is produced.

Finally, we determine statuses for the non-basic entries according to predicted probabilities of reaching the upper bound or lower bound, *i.e.*, for any $i \in \mathcal{N}_x$ and $j \in \mathcal{N}_s$

$$x_i^{\mathcal{B}} = \begin{cases} \ell_i^x & \text{if } p_{x,i}[1] \geq p_{x,i}[3] \\ u_i^x & \text{otherwise} \end{cases} \quad \text{and} \quad s_j^{\mathcal{B}} = \begin{cases} \ell_j^s & \text{if } p_{s,j}[1] \geq p_{s,j}[3] \\ u_j^s & \text{otherwise.} \end{cases}$$

3.3 Numerical Experiments

In this section, we evaluate the performance of our proposed methodology as a warm-start strategy for the Simplex and column generation methods, as well as to investigate the impact of dataset diversity on the methodology and the potential for model transferability. To accomplish this, we conduct a series of experiments consisting of four main parts. First, in Section 3.3.1, we test the performance of the proposed methodology as a warm-start strategy for the Simplex method. Second, in Section 3.3.2, we repeat this experiment but using the column generation method. Third, in Section 3.3.3, we sought to determine the impact of dataset diversity on the proposed methodology by training the model on generated datasets with varying diversity and comparing the results. Finally, in Section 3.3.4, we evaluate the potential for model transferability by training the model on one dataset and then testing it on another dataset that comes from a different source.

Datasets We evaluate our method on a diverse collection of datasets: three publicly available (MIRP [121], LIBSVM [180, 8], and STOCH [24]), two privately sourced from large-

Algorithm 1: Smart Initial Basis Selection Algorithm for Linear Programs

```
1  Training Stage: Given historical LPs, train a basis prediction model.
   Input  : Past solved LPs set $\mathcal{D} = \{[(P^k), (\boldsymbol{x}^k, \boldsymbol{s}^k)]\}_{k=1}^K$
   Output: GNN model $f(\theta; \cdot)$
   ▷ Construct training dataset from past LPs
2  foreach $[(P^k), (\boldsymbol{x}^k, \boldsymbol{s}^k)] \in \mathcal{D}$ do
       ▷ Construct graph representation from LP data
3      Extract cost vector $\boldsymbol{c}$, bounds $\boldsymbol{\ell^x}, \boldsymbol{\ell^s}, \mathbf{u^x}, \mathbf{u^s}$, and constraint matrix $\mathbf{A}$;
4      Construct constraint node features $\{\mathbf{w}_j\}_{j=1}^m$ and variable node features $\{\mathbf{v}_i\}_{i=1}^n$;
5      Connect edge $E_{(w_j, v_i)}$ with weight $A_{ji}$ if $A_{ji} \neq 0$;
       ▷ Construct labels from optimal solution
6      Transform $(\boldsymbol{x}^k, \boldsymbol{s}^k)$ into one-hot labels $\{\boldsymbol{y}_{x,i}, \boldsymbol{y}_{s,j} \in \mathsf{OneHot}(3) \mid i \in [n], j \in [m]\}$;
7  Train $f(\theta; \cdot)$ with the constructed dataset: $\theta^* \leftarrow \arg\min_\theta \mathcal{L}_\mathcal{D}(\theta)$;
8  return $f(\theta; \cdot)$
9  Inference Stage: Given a testing LP, predict an initial basis.
   Input  : Testing LP $P^{\text{test}}$
   Output: Predicted basis $(\mathcal{B}'_x, \mathcal{B}'_s)$
   ▷ Basis generation
10 Infer basis probabilities $\{\mathbf{p}_{x,i}, \mathbf{p}_{s,j} \mid i \in [n], j \in [m]\}$ using $f(\theta; \cdot)$;
11 Select top-$m$ likely entries into $\mathcal{B}_x, \mathcal{B}_s$ according to $\mathbb{P}(\ell_i^x < x_i < u_i^x)$ and
   $\mathbb{P}(\ell_j^s < s_j < u_j^s)$;
   ▷ Basis adjustment
12 Adjust basis to make matrix non-singular;
13 Determine statuses for non-basic entries;
14 return $(\mathcal{B}'_x, \mathcal{B}'_s)$
15 Function GNN_Model($\theta, \{\mathbf{v}_i\}_{i=1}^n, \{\mathbf{w}_j\}_{j=1}^m$)
16     Initialize node features: $\mathbf{v}_i^0 \leftarrow \mathbf{v}_i$, $\mathbf{w}_j^0 \leftarrow \mathbf{w}_j$;
       ▷ Message passing for $L$ steps
17     for $l = 1$ to $L$ do
18         $\mathbf{v}_i^l \leftarrow f_l^V(\theta_l^V; \mathbf{v}_i^{l-1}, \sum_{j=1}^m E_{(v_i, w_j)} \mathbf{w}_j^{l-1})$;
19         $\mathbf{w}_j^l \leftarrow f_l^W(\theta_l^W; \mathbf{w}_j^{l-1}, \sum_{i=1}^n E_{(v_i, w_j)} \mathbf{v}_i^{l-1})$;
       ▷ Knowledge masking to ensure feasibility
20     $\mathbf{p}_{x,i} \leftarrow \mathsf{softmax}(\mathbf{v}_i^L + \mathbf{h}_{x,i})$;
21     $\mathbf{p}_{s,j} \leftarrow \mathsf{softmax}(\mathbf{w}_j^L + \mathbf{h}_{s,j})$;
22     return $\{\mathbf{p}_{x,i}, \mathbf{p}_{s,j} \mid i \in [n], j \in [m]\}$;
```

Dataset	#LPs	m=#constraints	n=#variables	density	$\frac{\text{\#basic variables}}{n}$ (%)	$\frac{\text{\#basic slacks}}{m}$ (%)
LIBSVM	100	$20.0K$	$20.0K$	5e-4$_{\pm\text{2e-19}}$	$31.3_{\pm\text{2e-3}}$	$68.7_{\pm\text{2e-3}}$
MIRP	28	$28.2K_{\pm 25.2K}$	$28.7K_{\pm 25.0K}$	2e-4$_{\pm\text{1e-4}}$	$40.0_{\pm 3.1}$	$58.6_{\pm 4.3}$
STOCH	100	$52.3K_{\pm 1.9K}$	$107.0K_{\pm 3.8K}$	5e-5	$48.9_{\pm\text{5e-4}}$	$0.0_{\pm\text{2e-4}}$
GEN	100	$1.0K$	$1.0K$	$0.1_{\pm\text{1e-3}}$	60	40
SC-1	525	$312.9K_{\pm 177.4K}$	$659.1K_{\pm 386.4K}$	3e-5$_{\pm\text{2e-4}}$	$38.8_{\pm 4.5}$	$20.1_{\pm 4.6}$
SC-2	190	$1.4M_{\pm 199.1K}$	$2.9M_{\pm 450.6K}$	2e-6$_{\pm\text{3e-7}}$	$36.7_{\pm 1.5}$	$22.6_{\pm 1.4}$

Table 3.2: The statistics of datasets.

scale supply chain problems (SC-1 and SC-2), and one synthetic dataset (GEN) generated using the LP generator of [19]. Dataset statistics are summarized in Table 3.2. All datasets are split into training and test sets with a 7:3 ratio.

For evaluation, we follow default configurations [121, 8, 24, 19], with minor adjustments to reflect scenarios where a series of similar LPs are solved. In LIBSVM, the Cod-RNA dataset is used, and training/test LPs are constructed by randomly sampling 20K points from each split. MIRP uses Group-1 LPs without modification. STOCH generates 2-stage stochastic supply chain LPs with 75–85 first-stage variables. In GEN, the diversity parameter λ (Section 3.3.3) is set to 10.

Candidate Optimization Solvers We evaluate the performance of the proposed methodology using two optimization solvers: HiGHS [82], a state-of-the-art open-source solver that offers both primal and dual Simplex methods, and an OptVerse solver [83] that additionally incorporates the column generation algorithm. To eliminate any potential impact from solver configurations, the presolve option is turned off, and default settings are used.

Implementation Our approach is implemented using Python 3.7, PyTorch 1.8, and the PyG framework [57]. The GNN model is trained on an NVIDIA V100 GPU (32 GB), and evaluations are conducted on a system equipped with an 8-core Intel Xeon E5-2690 v4 CPU and 64 GB memory, running Ubuntu 18.04 within Docker containers for solver execution. Our code is publicly available at the Huawei AI Gallery[1] . For the model architecture, we adopt the GNN proposed by [115] and customize the graph convolution operation. To efficiently process large, sparse constraint–variable bipartite graphs, each convolution layer is implemented as two sparse matrix multiplications—one for message passing from constraint to variable nodes and another in the reverse direction. By default, a 3-layer lightweight GNN is used for SC-1 and SC-2, and a 5-layer GNN for other datasets. The hidden dimension is 128, and the dropout rate is 0.1. We adopt standard hyperparameters without tuning,

[1]`https://developer.huaweicloud.com/develop/aigallery/notebook/detail?id=ce45dd10-44ce-43bb-89c8-1f3277f1132d`

Dataset	DEFAULT	CA	CA-MPC	CA-ANG	GNN(Ours)
Iterations					
LIBSVM	$14.9K_{\pm 9.5K}$	$14.9K_{\pm 9.5K}$	$21.0K_{\pm 4.8K}$	$15.2K_{\pm 1.1K}$	$\mathbf{9.1K}_{\pm \mathbf{3.1K}}$
MIRP	$40.3K_{\pm 23.3K}$	$34.8K_{\pm 20.2K}$	$36.7K_{\pm 20.8K}$	$39.6K_{\pm 22.7K}$	$\mathbf{25.9K}_{\pm \mathbf{16.9K}}$
STOCH	$75.3K_{\pm 4.3K}$	$52.5K_{\pm 4.8K}$	$48.7K_{\pm 5.2K}$	$53.3K_{\pm 1.7K}$	$\mathbf{31.8K}_{\pm \mathbf{14.3K}}$
GEN	$2.4K_{\pm 225.0}$	$2.4K_{\pm 225.0}$	$2.4K_{\pm 225.0}$	$2.4K_{\pm 225.0}$	$\mathbf{552.8}_{\pm \mathbf{642.9}}$
SC-1	$272.3K_{\pm 151.9K}$	$158.9K_{\pm 89.1K}$	$266.9K_{\pm 148.5K}$	$269.2K_{\pm 151.5K}$	$\mathbf{26.6K}_{\pm \mathbf{15.4K}}$
SC-2	$1.2M_{\pm 170.7K}$	$1.1M_{\pm 172.2K}$	$1.2M_{\pm 163.5K}$	$431.9K_{\pm 99.0K}$	$\mathbf{169.1K}_{\pm \mathbf{34.3K}}$
Time (s)					
LIBSVM	$16.6_{\pm 10.0}$	$16.7_{\pm 10.0}$	$27.9_{\pm 12.4}$	$28.3_{\pm 2.2}$	$\mathbf{11.0}_{\pm \mathbf{3.7}}$
MIRP	$22.1_{\pm 23.3}$	$21.4_{\pm 22.5}$	$18.6_{\pm 16.9}$	$21.6_{\pm 20.9}$	$\mathbf{15.4}_{\pm \mathbf{15.7}}$
STOCH	$44.6_{\pm 11.8}$	$61.3_{\pm 12.3}$	$51.3_{\pm 12.4}$	$53.2_{\pm 8.5}$	$\mathbf{42.7}_{\pm \mathbf{30.0}}$
GEN	$1.3_{\pm 0.2}$	$1.4_{\pm 0.2}$	$1.4_{\pm 0.3}$	$1.4_{\pm 0.3}$	$\mathbf{0.5}_{\pm \mathbf{0.5}}$
SC-1	$77.9_{\pm 68.4}$	$85.8_{\pm 80.3}$	$86.1_{\pm 79.5}$	$100.1_{\pm 94.0}$	$\mathbf{22.8}_{\pm \mathbf{23.5}}$
SC-2	$348.7_{\pm 101.0}$	$1.3K_{\pm 698.2}$	$382.8_{\pm 102.3}$	$338.7_{\pm 181.5}$	$\mathbf{87.3}_{\pm \mathbf{25.4}}$

Table 3.3: Performance comparison between the proposed and rule-based initial-basis strategy, with the **dual** Simplex method and the **OptVerse** solver.

using the Adam optimizer (learning rate 10^{-3}, weight decay 10^{-4}) with a decay factor of 0.1 every 200 epochs, trained for 800 epochs in total.

The training cost scales linearly with the number of constraints and variables, owing to the fact that each node aggregates information only from its neighbors, and the constraint matrices (i.e., connections) are typically sparse. Model training takes about 1–2 hours for small and medium datasets and 6–8 hours for the largest ones. Since our focus lies in achieving fast inference rather than minimizing training time, this cost is considered acceptable. Meanwhile, memory consumption is negligible for small graphs. For large graphs in SC-2 (up to 2 M nodes and 10 M edges), *neighborhood sampling* [76] is applied when the number of edges exceeds 10 M to control GPU memory usage.

3.3.1 Initial-Basis Strategy for the Simplex Algorithm

In this section, we evaluate the performance of the proposed methodology as a warm-start strategy for the Simplex method. The proposed method is compared to four initial-basis strategies:

1. **DEFAULT**: the initial basis contains all slack variables.
2. **CPLEX Crash (CA)** [15]: employs a heuristic approach to construct a basis with improved triangularity and fewer artificial variables.
3. **CA-MPC** [126]: constructs a sparse initial basis using triangulation and fill-reducing techniques, as implemented in the OptVerse solver.

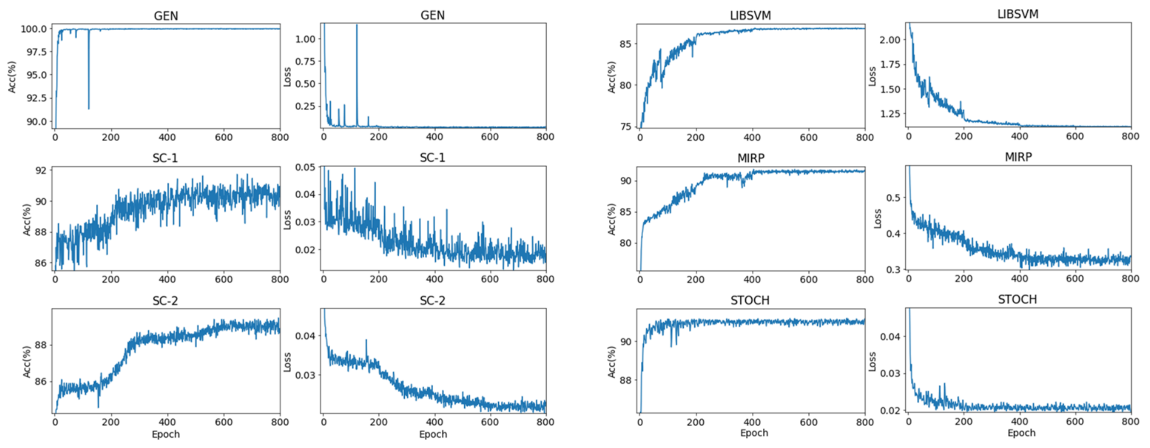


Figure 3.4: The convergence plots for training accuracy and loss

	Iterations			Time (s)		
Dataset	DEFAULT	CA	GNN(Ours)	DEFAULT	CA	GNN(Ours)
LIBSVM	$9.0K_{\pm 178.8}$	$9.0K_{\pm 64.2}$	**$5.2K_{\pm 1.5K}$**	$7.4_{\pm 0.2}$	$7.5_{\pm 0.1}$	**$4.6_{\pm 1.2}$**
MIRP	$29.9K_{\pm 17.0K}$	$25.9K_{\pm 14.5K}$	**$18.2K_{\pm 12.3K}$**	$17.8_{\pm 17.2}$	$17.6_{\pm 16.7}$	**$14.5_{\pm 14.2}$**
STOCH	$343.2K_{\pm 36.6K}$	$261.4K_{\pm 36.2K}$	**$165.1K_{\pm 42.6K}$**	$718.7_{\pm 112.5}$	$553.9_{\pm 102.9}$	**$251.6_{\pm 66.7}$**
GEN	$2.2K_{\pm 105.0}$	$2.2K_{\pm 103.4}$	**$80.4_{\pm 186.1}$**	$1.3_{\pm 0.1}$	$1.3_{\pm 0.1}$	**$0.2_{\pm 0.2}$**
SC-1	$262.6K_{\pm 147.0K}$	$207.5K_{\pm 111.0K}$	**$64.9K_{\pm 36.5K}$**	$21.3_{\pm 18.7}$	$62.9_{\pm 50.0}$	**$11.1_{\pm 11.2}$**
SC-2	$1.2M_{\pm 165.5K}$	$1.1M_{\pm 128.8K}$	**$214.3K_{\pm 40.4K}$**	$194.4_{\pm 58.9}$	$336.6_{\pm 103.6}$	**$65.0_{\pm 25.1}$**

Table 3.4: Evaluation of the performance of the initial-basis strategy for the **dual** Simplex method using the **HiGHS** solver.

4. **CA-ANG** [87]: heuristically builds an initial basis closer to the optimal vertex, also evaluated via the OptVerse solver.

Both the primal and dual Simplex methods were employed in the evaluation. The convergence plots, including training accuracy and loss, are shown in Figure 3.4. Performance results for the dual Simplex method are presented in Table 3.3 and Table 3.4, while Table 3.5 and Table 3.6 report the performance of our method using the primal Simplex algorithm. The tables compare the number of Simplex iterations and the total running time, which includes both the time required to determine the initial basis and the time for executing the Simplex method. Results are reported as $mean_{\pm std}$ over the test set. Overall, our proposed approach consistently reduces both iteration counts and running time across all datasets when using the primal Simplex method. In the following, we further analyze the performance of our method under the dual Simplex algorithm and provide additional insights.

We highlight the results in Table 3.3. The proposed initial-basis strategy consistently outperforms DEFAULT and the rule-based strategies (CA, CA-MPC, and CA-ANG) in terms of both the number of Simplex iterations and total running time across all datasets. This is because the heuristic-based strategies have three main drawbacks:

	Iterations			Time (s)		
Dataset	DEFAULT	CA	GNN (Ours)	DEFAULT	CA	GNN (Ours)
LIBSVM	$20.2K_{\pm 2.4K}$	$20.2K_{\pm 2.4K}$	$\mathbf{19.5K}_{\pm \mathbf{3.1K}}$	$18.2_{\pm 2.8}$	$18.5_{\pm 2.8}$	$\mathbf{18.1}_{\pm \mathbf{2.7}}$
STOCH	$56.2K_{\pm 30.1K}$	$19.3K_{\pm 8.4K}$	$\mathbf{15.4K}_{\pm \mathbf{9.5K}}$	$44.7_{\pm 36.1}$	$9.1_{\pm 7.3}$	$\mathbf{8.3}_{\pm \mathbf{7.0}}$
MIRP	$36.3K_{\pm 26.9K}$	$32.8K_{\pm 22.9K}$	$\mathbf{24.1K}_{\pm \mathbf{18.4K}}$	$24.5_{\pm 26.3}$	$20.4_{\pm 21.7}$	$\mathbf{15.7}_{\pm \mathbf{16.9}}$
SC-1	$359.6K_{\pm 210.4K}$	$211.2K_{\pm 126.1K}$	$\mathbf{34.9K}_{\pm \mathbf{23.1K}}$	$173.4_{\pm 205.1}$	$142.7_{\pm 164.6}$	$\mathbf{34.8}_{\pm \mathbf{65.2}}$
SC-2	$1.8M_{\pm 275.9K}$	$1.2M_{\pm 184.9K}$	$\mathbf{393.5K}_{\pm \mathbf{72.1K}}$	$1.4K_{\pm 627.7}$	$937.2_{\pm 449.0}$	$\mathbf{324.5}_{\pm \mathbf{133.1}}$

Table 3.5: Performance evaluation with **primal** Simplex algorithm in the **OptVerse** solver

	Iterations			Time (s)		
Dataset	DEFAULT	CA	GNN (Ours)	DEFAULT	CA	GNN (Ours)
GEN	$3.9K_{\pm 661.8}$	$3.9K_{\pm 661.8}$	$\mathbf{279.6}_{\pm \mathbf{444.2}}$	$1.8_{\pm 0.5}$	$1.9_{\pm 0.5}$	$\mathbf{0.2}_{\pm \mathbf{0.3}}$
LIBSVM	$9.1K_{\pm 726.2}$	$9.1K_{\pm 726.2}$	$\mathbf{6.1K}_{\pm \mathbf{794.1}}$	$6.0_{\pm 0.6}$	$6.0_{\pm 0.6}$	$\mathbf{3.7}_{\pm \mathbf{0.6}}$
STOCH	$332.5K_{\pm 39.6K}$	$318.5K_{\pm 25.9K}$	$\mathbf{314.8K}_{\pm \mathbf{68.5K}}$	$635.7_{\pm 169.4}$	$581.3_{\pm 101.2}$	$\mathbf{558.5}_{\pm \mathbf{70.1}}$
MIRP	$148.2K_{\pm 120.0K}$	$131.6K_{\pm 111.1K}$	$\mathbf{116.2K}_{\pm \mathbf{91.1K}}$	$111.4_{\pm 127.1}$	$98.1_{\pm 112.9}$	$\mathbf{82.1}_{\pm \mathbf{87.0}}$
SC-1	$387.4K_{\pm 199.5K}$	$213.6K_{\pm 106.6K}$	$\mathbf{100.8K}_{\pm \mathbf{86.2K}}$	$1.6K_{\pm 1.5K}$	$593.5_{\pm 598.9}$	$\mathbf{258.8}_{\pm \mathbf{613.7}}$
SC-2	$1.5M_{\pm 231.2K}$	$1.7M_{\pm 313.2K}$	$\mathbf{146.2K}_{\pm \mathbf{27.2K}}$	$17.9K_{\pm 4.6K}$	$23.2K_{\pm 7.5K}$	$\mathbf{2.0K}_{\pm \mathbf{651.5}}$

Table 3.6: Performance evaluation for **primal** Simplex algorithm in **HiGHS** solver

1. CA and CA-MPC are designed to achieve better numerical properties for the initial basis matrix, rather than being close to optimal. Consequently, only the first few iterations are accelerated, and the total number of iterations does not necessarily decrease.

2. CA-ANG aims to find an initial basis close to the optimal basis, but it only works when angular conjectures hold. Similarly, other rule-based methods are limited to specific problem structures: CA-CPLEX works only for problems with equality constraints, and CA-MPC prefers constraint matrices with more singleton columns.

3. They do not utilize information from previously solved LPs, which limits both their capacity and applicability.

The acceleration observed in the results is attributed to two aspects: (1). The inference time is negligible; (2). The predicted initial bases are close to optimal.

To verify this, the inference time and prediction performance of the proposed approach are presented in Table 3.7. The "Inference time (s)" column represents the additional time required during the inference stage, including GNN inference and basis adjustment. As shown in the results, the inference cost of the proposed method is insignificant, accounting for less than 10% of the total running time.

The performance of the GNN model in predicting optimal bases is evaluated using **accuracy**, **precision**, and **recall**, computed with our extended metric $\tilde{M}$. Accuracy measures the fraction of correctly predicted variables, while precision and recall capture the correct-

	Inference time (s)		Prediction performance		
Dataset	GNN inference	Basis adjustment	Accuracy (%)	Precision (%)	Recall (%)
LIBSVM	0.1$_{\pm 8e-3}$	0.1$_{\pm 2e-3}$	87.1$_{\pm 4e-2}$	84.6$_{\pm 4e-2}$	87.7$_{\pm 2e-2}$
MIRP	0.1$_{\pm 5e-2}$	3e-3$_{\pm 2e-3}$	88.9$_{\pm 1.8}$	78.4$_{\pm 1.6}$	80.8$_{\pm 5.4}$
STOCH	2e-2$_{\pm 7e-3}$	4e-3$_{\pm 2e-3}$	81.7$_{\pm 1.9}$	81.3$_{\pm 1.9}$	81.3$_{\pm 1.9}$
GEN	2e-2$_{\pm 6e-3}$	0.1$_{\pm 2e-2}$	99.9$_{\pm 0.1}$	99.9$_{\pm 0.1}$	99.9$_{\pm 0.1}$
SC-1	0.1$_{\pm 4e-2}$	0.3$_{\pm 0.2}$	93.0$_{\pm 2.1}$	89.0$_{\pm 5.4}$	84.6$_{\pm 6.5}$
SC-2	0.3$_{\pm 0.1}$	0.3$_{\pm 0.1}$	90.4$_{\pm 0.8}$	90.8$_{\pm 0.7}$	78.4$_{\pm 2.0}$

Table 3.7: Test performance of the GNN prediction model.

ness and completeness of the predictions. Unlike standard machine learning tasks, our basis generation ensures exactly m entries are predicted as basic. Metrics computed over all $m+n$ entries would yield identical precision and recall (and be equal to accuracy when $m = n$), failing to reflect true performance. High precision and recall together are required to indicate a reliable model, particularly when variable or constraint labels are imbalanced.

Formally, let $\{\hat{\boldsymbol{y}}_{x,i}^{(k)}\}_{i=1}^{n}$ and $\{\hat{\boldsymbol{y}}_{s,j}^{(k)}\}_{j=1}^{m}$ denote the predicted statuses of variables and constraints for the k-th LP, and $\{\boldsymbol{y}_{x,i}^{(k)}\}_{i=1}^{n}$ and $\{\boldsymbol{y}_{s,j}^{(k)}\}_{j=1}^{m}$ the corresponding ground-truth labels. The extended metric $\tilde{M}$, based on a standard ML metric M (e.g., accuracy, macro-precision, macro-recall), is defined as

$$\tilde{M} = \frac{1}{2K} \sum_{k=1}^{K} \Big[M\big(\{\hat{\boldsymbol{y}}_{x,i}^{(k)}\}_{i=1}^{n}, \{\boldsymbol{y}_{x,i}^{(k)}\}_{i=1}^{n}\big) + M\big(\{\hat{\boldsymbol{y}}_{s,j}^{(k)}\}_{j=1}^{m}, \{\boldsymbol{y}_{s,j}^{(k)}\}_{j=1}^{m}\big) \Big],$$

which averages separately over variables and constraints to provide a balanced assessment of model performance. Our GNN model achieves over 81% accuracy and 78% for both precision and recall across all datasets, indicating near-optimal bases.

3.3.2 Initial-RMP Strategy for the Column Generation Algorithm

In this section, we extend our proposed methodology to be a warm-start strategy for the column generation (CG) algorithm.

The column generation method, originally proposed by [42] and [67], is designed for linear programs (LPs) with a very large number of columns ($n \gg m$), where traditional Simplex iterations over all variables become computationally prohibitive. CG begins with a restricted master problem (RMP) that contains only a subset $\mathcal{F} \subseteq [n]$ of the full variable set and iteratively augments this set by adding columns identified via dual information. Formally, the RMP is defined as

	Iterations		Time (s)	
Dataset	DEFAULT	GNN	DEFAULT	GNN
LIBSVM	$34.3K_{\pm 45.1}$	$\mathbf{17.6K}_{\pm \mathbf{1.9K}}$	$24.9_{\pm 0.2}$	$\mathbf{19.0}_{\pm \mathbf{2.7}}$
MIRP	$59.2K_{\pm 46.0K}$	$\mathbf{31.2K}_{\pm \mathbf{25.9K}}$	$54.7_{\pm 65.2}$	$\mathbf{23.7}_{\pm \mathbf{28.0}}$
STOCH	$99.3K_{\pm 6.8K}$	$\mathbf{33.3K}_{\pm \mathbf{12.1K}}$	$29.7_{\pm 10.2}$	$\mathbf{23.3}_{\pm \mathbf{19.0}}$
GEN	$4.2K_{\pm 2.4K}$	$\mathbf{95.7}_{\pm \mathbf{120.9}}$	$1.4_{\pm 0.5}$	$\mathbf{0.5}_{\pm \mathbf{0.3}}$
SC-1	$345.2K_{\pm 194.0K}$	$\mathbf{68.1K}_{\pm \mathbf{41.4K}}$	$64.8_{\pm 56.3}$	$\mathbf{31.7}_{\pm \mathbf{35.0}}$
SC-2	$1.6M_{\pm 238.6K}$	$\mathbf{384.1K}_{\pm \mathbf{92.7K}}$	$721.6_{\pm 299.4}$	$\mathbf{463.5}_{\pm \mathbf{586.1}}$

Table 3.8: Evaluation of the performance of the initial-RMP strategy for the column generation method using the OptVerse solver.

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^{|\mathcal{F}|},\boldsymbol{s}\in\mathbb{R}^m} \quad & \boldsymbol{c}_{\mathcal{F}}^T\boldsymbol{x} \\ \text{s.t.} \quad & \mathbf{A}_{\mathcal{F}}\boldsymbol{x} = \boldsymbol{s} \\ & \boldsymbol{\ell}_{\mathcal{F}}^x \le \boldsymbol{x} \le \boldsymbol{u}_{\mathcal{F}}^x \\ & \boldsymbol{\ell}^s \le \boldsymbol{s} \le \boldsymbol{u}^s. \end{aligned} \tag{RMP}$$

If the predicted basis is already optimal, constructing an RMP based on it would allow the CG algorithm to converge immediately. This motivates the use of a GNN-predicted basis to accelerate CG. The initial subset $\mathcal{F}$ must be chosen carefully to ensure that (RMP) is feasible. Once the RMP is solved, the optimal dual solution is used to identify and add promising columns from $[n] \setminus \mathcal{F}$. The RMP is then updated and resolved, and this process repeats until convergence.

Since the GNN-predicted basis may yield a basic solution that is infeasible, we introduce an auxiliary LP with artificial slack variables for infeasible constraints. This auxiliary LP provides a feasible starting point and serves as the initial RMP for the CG algorithm.

We compare the proposed initial-RMP strategy with the DEFAULT approach, which heuristically constructs an initial feasible solution by fixing many variables at their bounds and forming an RMP from the remaining variables. Performance is evaluated in terms of the number of CG iterations and total runtime, as reported in Table 3.8. The results demonstrate that the initial-RMP strategy consistently reduces both CG iterations and running time across all datasets.

3.3.3 Impact of Dataset Diversity

In this section, we aim to evaluate the performance of our proposed approach on datasets with varying degrees of diversity. The efficacy of data-dependent machine learning techniques for solving LP problems is contingent on the similarity of the mapping from an LP instance to its optimal basis in a given dataset. To this end, we devise a problem-generation strategy that allows for the controllable manipulation of the diversity of the mapping.

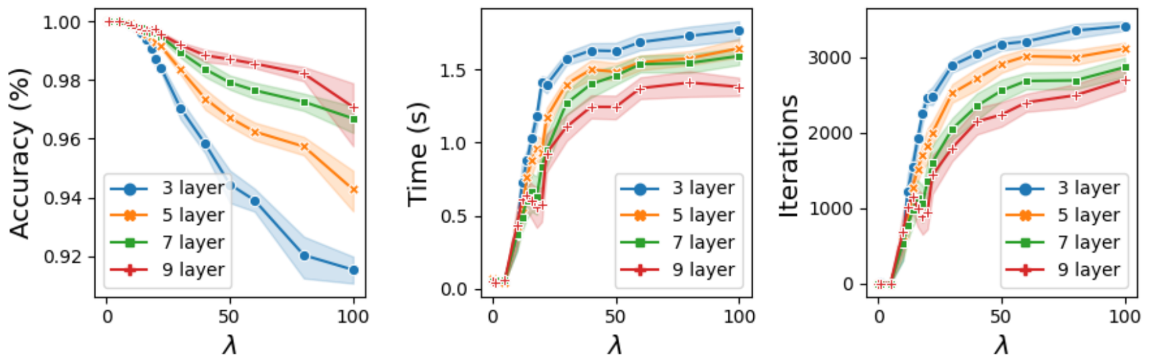


Figure 3.5: **Left:** Test accuracy versus λ. **Middle:** Total running time versus λ. **Right:** Simplex iterations versus λ.

Our problem-generation strategy is designed based on the LP-generation technique developed by [19]. Constraint matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is generated in the same way, where the maximum number of nonzeros in rows and columns will be less than thresholds τ_{row} and τ_{column}. Then we select basic entries according to the number of nonzeros in $\mathbf{A}$ and generate an optimal basic solution $(\boldsymbol{x}, \boldsymbol{s})$ through the following steps:

- $k = \lfloor \gamma m \rfloor$;
- Choose k indices $\mathcal{B}_x \subseteq [n]$ according to probabilities $\mathsf{softmax}_\lambda([nnz(\mathbf{A}_{:i})])$ without replacement; Here, $\mathsf{softmax}_\lambda(\boldsymbol{z})_i = \frac{\exp(z_i/\lambda)}{\sum_{j=1}^{d} \exp(z_j/\lambda)}$, $\forall i \in [d]$.
- Choose $m-k$ indices $\mathcal{B}_s \subseteq [m]$ according to probabilities $\mathsf{softmax}_\lambda([nnz(\mathbf{A}_{j:})])$ without replacement;
- Randomly fill in remaining values for $(\boldsymbol{x}, \boldsymbol{s})$.

Finally, following the same way, c and b are computed using complementary slackness, and LPs are generated.

In our LP-generating strategy, the scalar λ controls the similarity between the mapping from an LP instance to an optimal basis. When $\lambda = 1$ the $\mathsf{softmax}_\lambda$ coincides with $\mathsf{softmax}$, when $\lambda \to \infty$ the $\mathsf{softmax}_\lambda$ tends to uniform distribution, and when $\lambda \to 0$ the $\mathsf{softmax}_\lambda$ approximates onehot distribution. To conclude, as λ increases, the generated dataset becomes more diverse.

The evaluation results are presented in Figure 3.5, where we test the performance of GNN models with a varying number of layers. The left figure illustrates the relationship between test accuracy and the diversity of the dataset (λ). As depicted in the left figure, the test accuracy decreases as the diversity of the dataset increases, indicating that the diversity of the instances has a significant impact on the GNN prediction. The middle and right figures demonstrate the effect of dataset diversity on our initial-basis strategy for the

Iteration Ratio = GNN Iterations / DEFAULT Iterations						
Target / Source	LIBSVM	MIRP	STOCH	GEN	SC-1	SC-2
LIBSVM	**0.8$_{\pm 0.3}$**	1.0$_{\pm 0.1}$	0.8$_{\pm 3e-2}$	1.7$_{\pm 0.2}$	0.9$_{\pm 0.1}$	0.9$_{\pm 3e-2}$
MIRP	1.1$_{\pm 0.4}$	**0.6$_{\pm 0.1}$**	1.0$_{\pm 4e-2}$	1.4$_{\pm 0.9}$	1.0$_{\pm 3e-2}$	1.0$_{\pm 5e-3}$
STOCH	1.0$_{\pm 0.4}$	1.3$_{\pm 0.1}$	**0.4$_{\pm 0.2}$**	1.6$_{\pm 0.5}$	0.6$_{\pm 0.3}$	1.2$_{\pm 0.2}$
GEN	2.1$_{\pm 1.1}$	1.0$_{\pm 0.1}$	0.8$_{\pm 3e-2}$	**0.2$_{\pm 0.3}$**	1.2$_{\pm 0.1}$	1.3$_{\pm 0.1}$
SC-1	1.0$_{\pm 0.4}$	0.8$_{\pm 0.1}$	1.3$_{\pm 0.2}$	1.1$_{\pm 0.1}$	**0.1$_{\pm 2e-2}$**	**0.1$_{\pm 2e-2}$**
SC-2	1.0$_{\pm 0.4}$	0.8$_{\pm 0.1}$	0.8$_{\pm 0.1}$	1.0	0.3$_{\pm 0.1}$	**0.1$_{\pm 2e-2}$**

Time Ratio = GNN Time / DEFAULT Time						
Target / Source	LIBSVM	MIRP	STOCH	GEN	SC-1	SC-2
LIBSVM	**0.8$_{\pm 0.2}$**	1.1$_{\pm 0.2}$	2.1$_{\pm 0.4}$	1.8$_{\pm 0.3}$	1.8$_{\pm 0.5}$	2.5$_{\pm 0.4}$
MIRP	1.1$_{\pm 0.4}$	**0.7$_{\pm 0.1}$**	1.0$_{\pm 0.1}$	3.5$_{\pm 6.1}$	1.2$_{\pm 0.1}$	1.2$_{\pm 0.2}$
STOCH	1.0$_{\pm 0.4}$	1.6$_{\pm 0.4}$	**0.9$_{\pm 0.5}$**	0.7$_{\pm 0.3}$	1.7$_{\pm 0.9}$	3.9$_{\pm 2.4}$
GEN	2.0$_{\pm 1.0}$	1.1$_{\pm 0.3}$	2.1$_{\pm 0.3}$	**0.1$_{\pm 0.1}$**	3.1$_{\pm 1.0}$	7.4$_{\pm 4.3}$
SC-1	1.2$_{\pm 0.4}$	0.8$_{\pm 0.1}$	3.1$_{\pm 0.7}$	1.1$_{\pm 0.1}$	**0.3$_{\pm 0.1}$**	**0.3$_{\pm 3e-2}$**
SC-2	1.0$_{\pm 0.3}$	1.0$_{\pm 0.1}$	1.2$_{\pm 0.1}$	1.0$_{\pm 0.1}$	0.6$_{\pm 0.3}$	**0.3$_{\pm 4e-2}$**

Table 3.9: Performance of models trained on a source dataset transferring to a target dataset. The entries show the Iteration/Time ratio between utilizing the GNN model trained on a source dataset and adopting the DEFAULT strategy towards a target dataset. **Bold** entries mean that on the corresponding target dataset (column-wise), the fastest iterations/time is achieved.

Simplex method. The results show that both the running time and the number of Simplex iterations increase as the diversity of the dataset increases, which is consistent with the findings in the left figure, as the predicted bases are farther from the optimal ones.

3.3.4 Cross-dataset Evaluation

In this section, we evaluate the transferability of our proposed approach. The motivation for this evaluation is that, in practice, datasets may originate from different sources, such as airplane scheduling and product planning. It is unknown whether our approach could produce a general model transferable to various sources.

To test this, we train the GNN model on one dataset and then test it on another, with the two datasets having distinct sources. The results are presented in Table 3.9. It is observed that most models (except the one trained on SC-2) perform best on their corresponding test sets. The model trained on SC-1 performs well on the SC-2 test set, which may be because SC-1 and SC-2 have the same source (supply chain demand). For the remaining models,

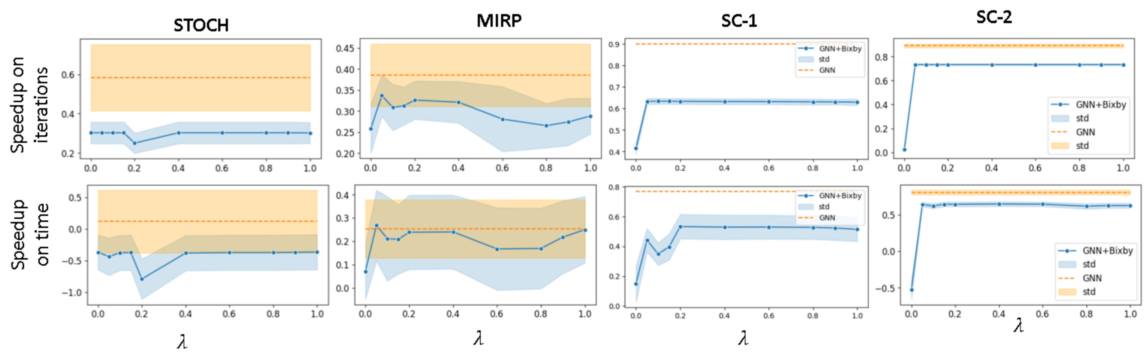


Figure 3.6: Speedup v.s. the weight of GNN prediction on Mirp, STOCH, SC-1, and SC-2 datasets.

their performance on test sets from other sources is poor, as they perform similarly to the DEFAULT initial-basis strategy.

In summary, the numerical results suggest that our proposed approach does not possess good model transferability. This may be attributed to the different problem structures of LPs from different sources, such as the block structures in the constraint matrix and the topology of the corresponding Simplex polyhedron.

3.3.5 Hybrid Classical and GNN-based Strategies

This experiment is motivated by two questions. First, Can rule-based classical warm-start strategies and data-driven GNN-based methods be fruitfully combined? Classical heuristics (e.g., CPLEX Crash) are valued for their robustness and decades of engineering; GNN-based approaches, in turn, offer adaptivity through learned problem structure. Yet, it is not obvious *a priori* whether these paradigms are complementary—or whether their underlying design principles may conflict. Second, classical and GNN-based methods appear to excel along *orthogonal dimensions*. Classical methods prioritize *numerical stability*, constructing bases with near-triangular structure to minimize factorization cost and ensure *fast early iterations*. In contrast, GNN-based methods prioritize *solution proximity*, predicting bases closer to optimality to reduce the *total number of iterations*. This suggests a potential complement: a hybrid strategy could retain cheap early iterations while converging in fewer steps.

To test this hypothesis, we design a way to inject GNN predictions into CPLEX Crash (CA) [15]. We first introduce the CA strategy. It constructs its initial basis in three steps:

1. Add slack variables for all one-sided inequality constraints into basis;

2. Sort variables by heuristic penalties $q \in \mathbb{R}^n$ proposed in [15], where smaller q_i indicates higher priority (i.e., the corresponding variable is heuristically “freer”);

3. Greedily select variables to preserve triangularity of the basis matrix $[A_{\mathcal{B}_x} \ -I^m_{\mathcal{B}_s}]$, and add them to basis.

We will inject GNN predictions into variable-prioritization step (Step 2) while leaving its structural safeguards (Steps 1 and 3) unchanged. Let $p = \left[\mathbb{P}(l^x_1 < x_1 < u^x_1), \ldots, \mathbb{P}(l^x_n < x_n < u^x_n)\right]$ denote the GNN-predicted probability of each variable being basic. Since q and $-p$ are on different scales, we apply z-score normalization and define a *combined penalty*:

$$q' = \lambda \cdot \text{zscore}(-p) + (1-\lambda) \cdot \text{zscore}(q), \tag{3.4}$$

where $\lambda \in [0, 1]$ controls the influence of the GNN signal:

- $\lambda = 0$: pure CPLEX Crash;
- $\lambda = 1$: GNN-guided variable ordering *within* the CA framework;
- $0 < \lambda < 1$: hybrid strategies.

We evaluate performance using *speedup*, defined as:

$$\text{Speedup} = \frac{\text{Iterations (or time) of DEFAULT} - \text{Iterations (or time) of X}}{\text{Iterations (or time) of DEFAULT}}, \tag{3.5}$$

where X $\in$ {CA ($\lambda = 0$), hybrid ($0 < \lambda < 1$), GNN-augmented CA ($\lambda = 1$)}. Datasets without equality constraints (GEN, LIBSVM)—where CA plays no role due to the absence of artificial variables—are excluded. Results for Mirp, STOCH, SC-1, and SC-2 are shown in Figure 3.6. The blue line shows the mean speedup of hybrid strategies (within the CA framework) across varying λ. The dashed orange line shows the mean speedup of the pure GNN warm-start, which bypasses Steps 1 and 3 of CA framework. The results yield two key insights:

1. Hybrid strategies with $\lambda > 0$ *consistently outperform* pure CA ($\lambda = 0$) across all benchmarks, confirming that GNN predictions improve upon CA's heuristic variable prioritization.
2. However, even when $\lambda = 1$—which fully replaces CA's penalty with GNN guidance—remains *inferior* to *pure GNN warm-start*.

Our hybrid design does not surpass the pure GNN warm-start. We attribute this to Steps 1 and 3 of CPLEX Crash, which restrict the feasible set of initial bases and may push the starting point farther from the optimum. Although these steps ensure fast early iterations via near-triangular structure—and thus confer numerical advantages—the resulting *solution-quality loss* outweighs these benefits when a high-fidelity GNN prediction is available. Rather than a simple hybridization, this result highlights the need for co-design: a deeper integration of learning and optimization components at a more fundamental level, where numerical stability and solution proximity are jointly optimized—not sequentially imposed.

3.4 Limitations and Future Works

Extension to Interior-Point Methods. Our current model predicts basis status (i.e., at the lower bound, basic, or at the upper bound) and is thus naturally aligned with simplex-type solvers. A interesting direction is to investigate whether the predicted basis information can also be used to warm-start interior-point methods (IPMs).

In principle, a predicted basic feasible solution may be converted into a strictly interior initialization by applying a feasibility-preserving perturbation (e.g., shifting boundary variables away from 0 and re-centering), thereby obtaining positive primal and slack variables. However, such a conversion does not guarantee proximity to the central path, which is typically more critical for IPM efficiency than proximity to a vertex solution. Therefore, although a basis initialization can in principle be converted into an interior-point initialization, such a transformation does not necessarily lead to significant speedup in the solving process. Careful empirical validation and additional algorithmic design (e.g., centering strategies) are required. This remains an interesting direction for future research.

Interpretability: What the GNNs Learned? Recent work has established theoretical foundations for learning to optimize and has analyzed the representational power of GNNs for modeling linear and mixed-integer optimization problems [30, 31, 128, 102]. However, it remains a question about what the GNNs Learned.

The bipartite LP graph encodes coefficients, bounds, and objective information, while message passing aggregates signals between constraints and variables. Empirically, the network behaves like a learned classifier that correlates local structural patterns (e.g., sparsity, coefficient magnitudes, and neighborhood interactions in a k-hops reception field) with optimal basis status [109].

To help understand what the GNN learns, consider an extreme yet intuitive example. In historical LP instances of a supply chain planning problem, suppose that all apple-related products consistently experience very high demand. In such cases, the optimal basis solution tends to allocate as much available material and machine capacity as possible to apple-related production. Consequently, constraints associated with these materials and machines are more likely to become binding, i.e., reach their capacity limits. Through repeated exposure to such patterns in the training data, the GNN may learn to associate high demand signals and structural connectivity with a higher probability of corresponding constraints being binding and related variables being basic.

Meanwhile, to better understand the learned signals in a scientifically rigorous manner, we may adopt post-hoc GNN interpretability methods [168, 166, 110] that attribute predictions to subsets of nodes, edges, or input features. Representative approaches include perturbation-based subgraph and feature masking techniques, such as GNNExplainer [166].

Such methods can provide insight into which constraints, coefficients, or bound-related features most strongly influence whether a variable is predicted to be basic or at one of its bounds. However, these interpretability tools are general-purpose methods developed for graph representation learning models and do not explicitly leverage the algebraic structure of linear programs. To the best of our knowledge, there are currently no interpretability methods specifically designed for LP/MILP bipartite graphs. Therefore, developing interpretation frameworks that incorporate optimization-specific properties remains an open and promising research direction.

Generalization across datasets. We observe that cross-dataset generalization can degrade when the training and test sets are from different application domains. There may be multiple reasons for this limitation.

First, the GNN architecture we used may itself be limited. For example, a 3-layer GNN has a receptive field of only three hops, meaning that it can utilize information only within a local subgraph and lacks the ability to effectively capture global structural information.

Although we experimented with increasing the number of layers and observed some performance improvement, training became significantly slower and convergence more difficult as the model depth increased. This may be because deeper message passing can suffer from the over-smoothing phenomenon [23, 165], where node representations become less discriminative as the number of layers grows. This issue has been extensively analyzed both theoretically and empirically in the GNN literature.

Therefore, exploring more expressive GNN architectures is a promising direction for future work. In particular, models that incorporate global attention mechanisms, higher-order interactions, or algebraically informed inductive biases may better capture global structural dependencies beyond local message passing. Such architectures could potentially move beyond learning local statistical correlations and instead approximate algebraic properties of LP instances, such as linear dependence, rank conditions, and global constraint interactions, which are closely related to basis selection.

Second, the performance degradation may be partially attributed to distribution shift [159, 100], both in numerical features and graph structure. Specifically, the training and test sets may differ in coefficient distributions, constraint-to-variable ratios, sparsity patterns, or other structural characteristics. In such cases, the model may rely on correlations that are stable within the training dataset but fail to generalize under a different formulation regime.

However, in practice, our goal is to develop a powerful and versatile model capable of accelerating a broad range of LP instances. Inspired by the scaling laws observed in the

LLM literature, a promising direction is to investigate whether larger models trained on more diverse and large-scale datasets can achieve improved robustness and generalization.

Accordingly, one potential avenue is to explore graph pre-training followed by large-scale supervised fine-tuning [77], with the objective of learning more transferable representations. However, as discussed previously, the GNN architecture itself may require further improvement. Without architectural advances, it may struggle to faithfully represent global algebraic properties of LPs, such as linear independence and rank conditions of constraint submatrices. As a result, the model may still lack mathematical reasoning capability and fail to truly understand or accurately predict optimal bases. Therefore, this remains a long-term research direction.

As a more immediate and practical step, a promising alternative is to leverage LLMs to generate human-interpretable heuristics or guidance strategies that assist program solving [163, 26], thereby complementing learned basis prediction models.

3.5 Conclusion

In this chapter, we proposed a novel graph neural network (GNN) method for smart initial basis selection in linear programming. Our approach represents LP instances as weighted bipartite graphs and learns a mapping from problem instances to high-quality basis selections using GNNs. The main contributions are: (1) a principled graph representation of LP problems that captures both structural and numerical characteristics while preserving permutation equivariance; (2) a GNN architecture tailored to basis prediction, equipped with knowledge-based masking to enforce feasibility constraints; (3) an inference procedure that couples probabilistic predictions with basis adjustment techniques to guarantee mathematical validity; and (4) comprehensive numerical experiments demonstrating substantial improvements over traditional rule-based initialization strategies. The method is particularly effective in settings where similar LP instances are solved repeatedly, such as industrial applications and column generation frameworks. Finally, we provide insights into hybridizing the learned model with classical heuristics, the transferability of the learned model, and the limitations of the proposed approach.

While this chapter exemplifies how learning on structured representations of LPs can enhance the solving stage of the OR pipeline, the next chapter moves one step upstream to the construction of LP formulations from natural language descriptions and explores how ML can further enhance this formulation stage of the OR process, as well as what kinds of evaluation metrics are required in this setting.

Chapter 4

Towards Human-Aligned Evaluation for the Natural Language to Linear Program Task

The Natural Language to Linear Program Task (NL2LP) is a type of mathematical word problem in which a natural language description must be translated into a formal linear program (LP). Unlike elementary arithmetic or algebra problems, solving an NL2LP instance requires identifying decision variables, constructing an objective function, and formulating constraints based on textual information. For instance, an NL2LP problem may describe a resource allocation or scheduling scenario, and the solver must derive the corresponding optimization model. An overview of this process is illustrated in Figure 4.1.

NL2LP tasks are particularly important because they closely resemble real-world decision-making in operations research, including transportation planning, production scheduling, and network design. Despite their practical significance, NL2LP remains relatively under-explored, partly due to its reasoning-intensive nature and the lack of evaluation metrics that reflect expert judgment.

With the advent of large language models (LLMs), non-experts can now automatically generate LP formulations from natural language descriptions. Consequently, it has become increasingly important to identify suitable LLMs and to develop evaluation metrics that align with expert opinion. In this work, we highlight the limitations of prior metrics and introduce a new metric designed to better capture expert-aligned performance.

4.1 Pitfalls of Prior Metrics

Prior research commonly evaluates NL2LP performance using two metrics: *Canonical Accuracy* and *Execution Accuracy*.

Canonical Accuracy [134] It evaluates prediction quality at the declaration level, where a declaration corresponds to either an optimization objective or a constraint. For a

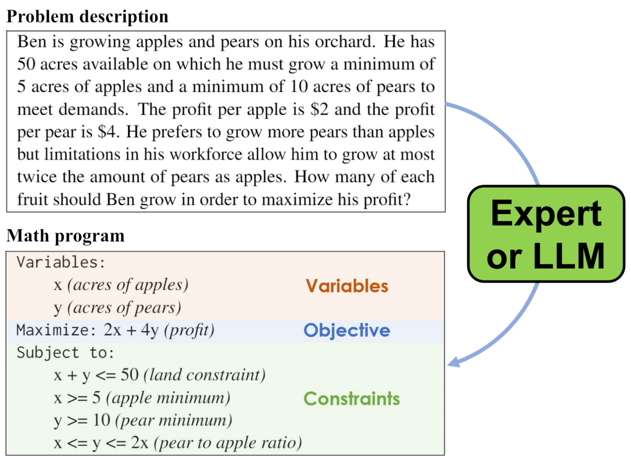

Problem description
Ben is growing apples and pears on his orchard. He has 50 acres available on which he must grow a minimum of 5 acres of apples and a minimum of 10 acres of pears to meet demands. The profit per apple is $2 and the profit per pear is $4. He prefers to grow more pears than apples but limitations in his workforce allow him to grow at most twice the amount of pears as apples. How many of each fruit should Ben grow in order to maximize his profit?
Math program
Variables:
x (acres of apples)
y (acres of pears)
Maximize: 2x + 4y (profit)
Subject to:
x + y <= 50 (land constraint)
x >= 5 (apple minimum)
y >= 10 (pear minimum)
x <= y <= 2x (pear to apple ratio)
Variables
Objective
Constraints
Expert or LLM

Figure 4.1: Illustration of a Natural Language to Linear Program (NL2LP) task, where the textbook or real-world problem is described in natural language, and then an expert or LLM identifies decision variables, formulates the objective function, and generates constraints to construct the corresponding linear program.

single NL2LP problem, it is computed as:

$$Acc_i = 1 - \frac{\min(FP_i + FN_i, D_i)}{D_i} \tag{4.1}$$

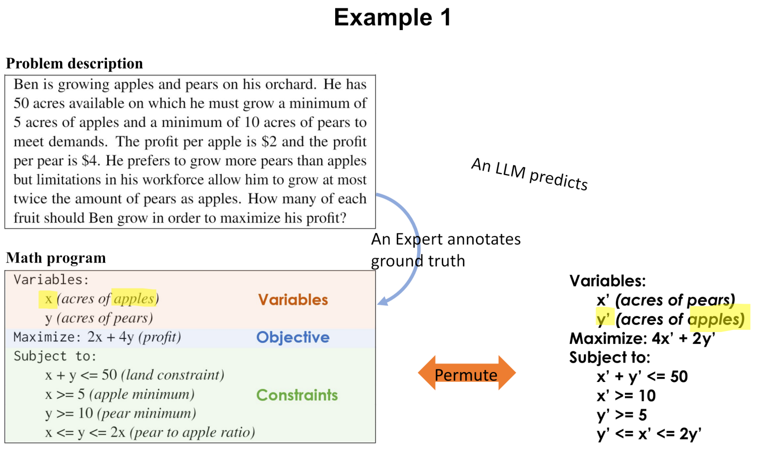
Here, D_i is the total number of declarations in the reference program. FP_i counts predicted declarations not matching any reference declarations, and FN_i counts reference declarations not matched by any predictions. The min operation ensures that the accuracy does not become negative or overly penalized when $FP_i + FN_i > D_i$.

A key limitation of this metric is its sensitivity to declaration order. For instance, the constraints $a \cdot X + b \cdot Y \leq c$ and $a \cdot Y + b \cdot X \leq c$ are mathematically equivalent, but canonical accuracy would treat them as different due to the variable ordering. Figure 4.2 illustrates such a scenario, where a predicted program is equivalent to the reference but receives a low canonical score because of variable permutation.

Execution Accuracy [127] Inspired by functional correctness evaluation in code generation [28], it assesses whether the predicted program produces the same optimal objective as the reference. Specifically, the LLM-generated program is converted into an MPS[1] file, which is then solved to obtain the optimal value. A match in optimal objectives between predicted and reference programs is counted as a correct prediction.

Despite its appeal, execution accuracy can be misleading. As shown in Figure 4.3 (Example 2), a predicted program may ignore several constraints yet still achieve the reference's

[1]An MPS (Mathematical Programming System) file is an industry-standard format for linear and mixed-integer programs [155].



Example 1
Problem description
Ben is growing apples and pears on his orchard. He has 50 acres available on which he must grow a minimum of 5 acres of apples and a minimum of 10 acres of pears to meet demands. The profit per apple is $2 and the profit per pear is $4. He prefers to grow more pears than apples but limitations in his workforce allow him to grow at most twice the amount of pears as apples. How many of each fruit should Ben grow in order to maximize his profit?
Math program
Variables:
x (acres of apples)
y (acres of pears)
Variables
Maximize: 2x + 4y (profit)
Objective
Subject to:
x + y <= 50 (land constraint)
x >= 5 (apple minimum)
y >= 10 (pear minimum)
x <= y <= 2x (pear to apple ratio)
Constraints
An Expert annotates ground truth
An LLM predicts
Permute
Variables:
x' (acres of pears)
y' (acres of apples)
Maximize: 4x' + 2y'
Subject to:
x' + y' <= 50
x' >= 10
y' >= 5
y' <= x' <= 2y'

Figure 4.2: An example illustrating the limitation of Canonical Accuracy: the predicted program merely swaps the order of variables compared to the reference, but canonical evaluation penalizes this change, producing a misleadingly low score.

optimal value. Moreover, two programs both deemed “infeasible” by the solver are considered identical, even if their underlying formulations differ substantially (Figure 4.3, Example 3).

4.2 Evaluation via Graph Edit Distance

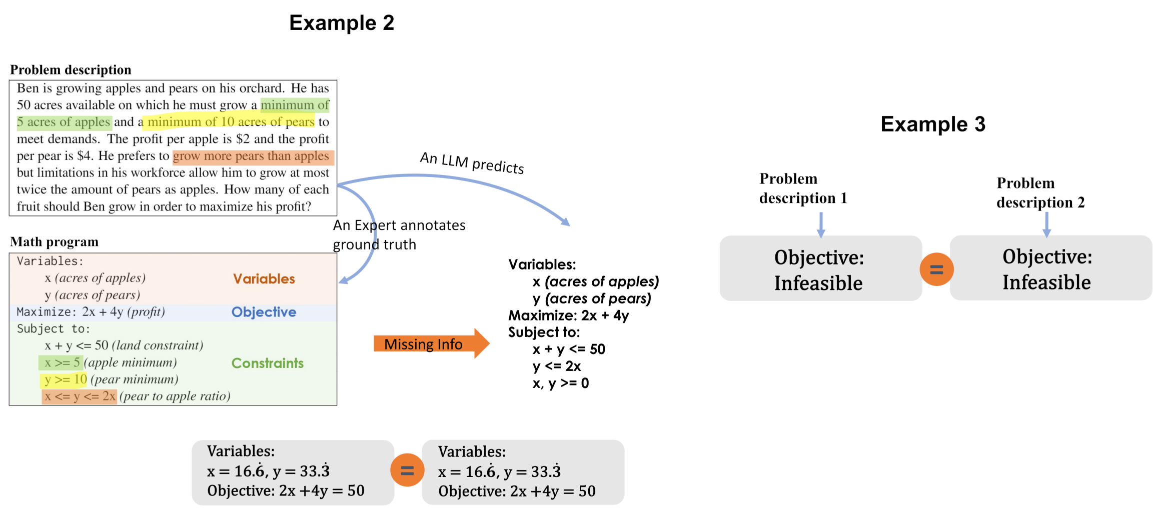
This section introduces a simple yet effective evaluation strategy grounded in graph edit distance (GED). This strategy tackles the pitfalls of prior metrics and aligns more closely with human judgment, as a smaller edit distance indicates fewer mistakes in the predicted program *w.r.t.* the reference program.

Generally, this evaluation strategy unfolds in three steps:

1. Converting the initial textual form of the predicted and reference programs into Linear Programs (LPs) in *general forms*.
2. Transforming the predicted and reference LPs into *bipartite graphs*.
3. Calculating the *graph edit distance* between the predicted and reference graphs.

LP in General Form We continue to use the formulation P in Section 3.1, and provide a quick recap of its simplified form below:

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^n} \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{\ell}^s \leq \mathbf{A}\boldsymbol{x} \leq \boldsymbol{u}^s \\ & \boldsymbol{\ell}^x \leq \boldsymbol{x} \leq \boldsymbol{u}^x, \end{aligned} \tag{P}$$



Ben is growing apples and pears on his orchard. He has 50 acres available on which he must grow a minimum of 5 acres of apples and a minimum of 10 acres of pears to meet demands. The profit per apple is \$2 and the profit per pear is \$4. He prefers to grow more pears than apples but limitations in his workforce allow him to grow at most twice the amount of pears as apples. How many of each fruit should Ben grow in order to maximize his profit?

```
Variables:
    x (acres of apples)
    y (acres of pears)
Maximize: 2x + 4y (profit)
Subject to:
    x + y <= 50 (land constraint)
    x >= 5 (apple minimum)
    y >= 10 (pear minimum)
    x <= y <= 2x (pear to apple ratio)
```

```
Variables:
    x (acres of apples)
    y (acres of pears)
Maximize: 2x + 4y
Subject to:
    x + y <= 50
    y <= 2x
    x, y >= 0
```

Variables: $x = 16.\dot{6}$, $y = 33.\dot{3}$
Objective: $2x + 4y = 50$

Variables: $x = 16.\dot{6}$, $y = 33.\dot{3}$
Objective: $2x + 4y = 50$

Figure 4.3: Examples demonstrating the limitations of Execution Accuracy. **Example 2**: the predicted program omits several constraints but still matches the reference optimal value; **Example 3**: two different programs are both classified as infeasible, leading to a misleading match.

For this step, we implement a robust rule-based parser to convert the initial textual math program into the LP in this general form.

Graph representation We continue to use the attributed bipartite graph representation shown in Figure 3.2 and concisely recap it. We denote it by $\mathcal{G} = (S \cup X, E)$ in this section. This graph consists of two disjoint vertex sets $S = \{s_i \mid i \in [m]\}$ and $X = \{x_j \mid j \in [n]\}$, and a collection $E = \{e_{ij} \mid i \in [m], j \in [n]\}$ of edges. Here, the notation $[\cdot]$ means a set of consecutive numbers. Vertex s_i corresponds to the i-th constraint $\ell_i^s \leq \mathbf{a}_i^\top \boldsymbol{x} \leq u_i^s$, with its attribute being $\mathrm{attr}(s_i) = [\ell_i^s, u_i^s]^\top$. The notation x_j is overloaded in the graph context to represent the vertex x_j that corresponds to the decision variable x_j. Its attribute $\mathrm{attr}(x_j) = [\ell_j^x, u_j^x, c_j]^\top$ contains the bounds (ℓ_j^x, u_j^x) and objective coefficient (c_j). The topology of $\mathcal{G}$ is determined by $\mathbf{A}$, *i.e.*, edge e_{ij} exists iff $A_{ij} \neq 0$. The attribute of this edge is simply the weight A_{ij}, *i.e.*, $\mathrm{attr}(e_{ij}) = [A_{ij}]$.

One significant advantage of this graph representation is that it introduces ***permutation invariance*** into the evaluation metric. This means that even if the constraints of the input LPs are permuted, or if the decision variables (along with the corresponding cost vector and columns of the matrix $\mathbf{A}$) are reordered, the resulting graph remains equivalent, underpinning the invariance of the overall evaluation metric.

Graph edit distance (GED) GED is the minimum cost required to transform one graph into another by a sequence of operations including inserting, deleting, and substituting vertices and/or edges (as shown in Fig 4.4). For generality, all these operations are viewed

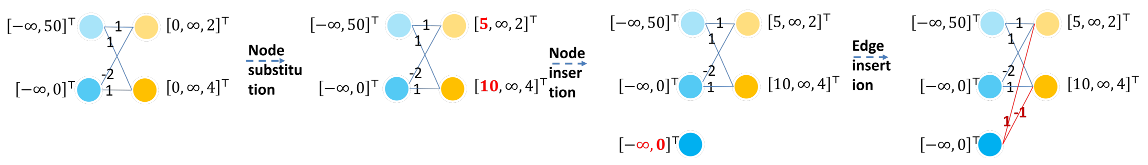


Figure 4.4: Exemplar graph edit path from the graph associated with the predicted program to the reference program in Figure 4.3 (Example 2). Blue and yellow vertices are respectively constraint and variable vertices.

as matching, *e.g.*, deleting a vertex is to match this vertex to an empty vertex, denoted by ϵ.

Any well-established Graph Edit Distance (GED) algorithm [1, 137, 62] can be employed once the costs of matching operations are specified. While various cost definitions exist, our proposed scheme follows a simple principle: **each operation on a vertex attribute incurs a unit cost of 1**.

Let $\mathcal{G}^p = (S^p \cup X^p, E^p)$ denote the predicted program graph and $\mathcal{G}^r = (S^r \cup X^r, E^r)$ the reference program graph. Based on the principle above, the vertex cost matrix $\mathbf{C}_v$ is formally defined as:

	$s^r_{i'} \in S^r$	$x^r_{i'} \in X^r$	ϵ
$s^p_i \in S^p$	① $\#\text{msm}(s^p_i, s^r_{i'})$	∞	④ $\#\text{attr}(s^p_i)$
$x^p_i \in X^p$	② ∞	$\#\text{msm}(x^p_i, x^r_{i'})$	$\#\text{attr}(x^p_i)$
ϵ	③ $\#\text{attr}(s^r_{i'})$	$\#\text{attr}(x^r_{i'})$	∞

Here, the rows' names correspond to vertices in the predicted graph, while the columns' names correspond to vertices in the reference graph. The special symbol ϵ represents a "null vertex," used for deletion (mapping a vertex in the predicted graph to ϵ) or insertion (mapping ϵ to a vertex in the reference graph).

We now explain the representative entries ①–④; the remaining entries are defined analogously.

- ① **Substitution between constraint vertices:** Substituting a constraint vertex s^p_i from the predicted graph with a constraint vertex $s^r_{i'}$ from the reference graph corresponds to editing the attributes of one vertex to match the other:

$$\mathbf{C}_v(s^p_i \rightarrow s^r_{i'}) = \#\text{msm}(s^p_i, s^r_{i'})$$

  where $\#\text{msm}(\cdot)$ denotes the number of mismatched attributes between two vertices.

- ② **Invalid substitution between different vertex types:** A constraint vertex cannot be converted into a variable vertex (and vice versa). Such substitutions are therefore assigned an infinite cost:

$$\mathbf{C}_v(s^p_i \rightarrow x^r_{i'}) = \infty$$

- ③ **Vertex deletion:** Deleting a vertex in the predicted graph incurs a cost equal to the number of its attributes. For example, deleting a constraint vertex s_i^p:

$$\mathbf{C}_v(s_i^p \to \epsilon) = \#\text{attr}(s_i^p)$$

- ④ **Vertex insertion:** Conversely, inserting a vertex into the predicted graph can be viewed as converting a null vertex into the inserted one:

$$\mathbf{C}_v(\epsilon \to s_i^r) = \#\text{attr}(s_i^r)$$

Similarly, the edge cost matrix $\mathbf{C}_e$ is defined as:

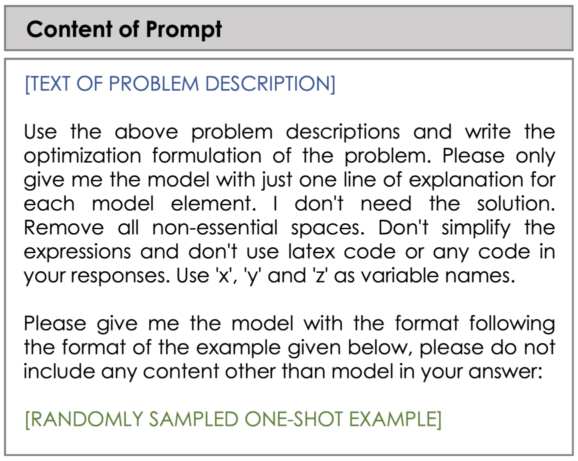
	$e_{ij}^r \in E^r$	ϵ
$e_{ij}^p \in E^p$	$\#\text{msm}(e_{ij}^p, e_{ij}^r)$	$\#\text{attr}(e_{ij}^p)$
ϵ	$\#\text{attr}(e_{ij}^r)$	∞

While $\text{GED}(\mathcal{G}^p, \mathcal{G}^r)$ can measure the similarity between predicted and reference programs, it is sensitive to graph size. A larger graph, representing a predicted program with more variables and constraints, is more prone to errors, thereby leading to larger GED *w.r.t.* the reference program. To address this issue, we further normalize $\text{GED}(\mathcal{G}^p, \mathcal{G}^r)$ by the graph size, as $\text{NGED}(\mathcal{G}^p, \mathcal{G}^r) = \frac{\text{GED}(\mathcal{G}^p, \mathcal{G}^r)}{\max(|\mathcal{G}^p|, |\mathcal{G}^r|)}$, where $|\mathcal{G}| = \sum_{e \in E} \#\text{attr}(e) + \sum_{v \in S \cup X} \#\text{attr}(v)$. Ultimately, NGED forms the core of our proposed evaluation metric for NL2LP. We adopted the exact algorithm proposed by [1], which is a branch-and-bound method with a tailored pruning and branching strategy. The worst-case complexity of comparing two bipartite graphs with n_1 and n_2 nodes is $O\big((n_1 n_2)^{n_1+n_2}\big)$. In practice, however, the predicted and ground-truth graphs often share similar characteristics, such as easily identifiable identical nodes, which allows the algorithm to scale reasonably well—for example, it can process graphs with 15 nodes within 60 seconds. A notable limitation is its inability to handle larger graphs efficiently. Nevertheless, as mentioned in Section 4.2, framing the evaluation as a graph edit distance problem allows us to leverage various existing algorithms. One potential approach is to use an approximate algorithm, such as that in [136], with cubic complexity $O(\max\{n_1, n_2\}^3)$. A more detailed exploration of such methods is left for future work.

4.3 Experiments and Analysis

4.3.1 Experimental Setup

Datasets We use the NL4OPT benchmark [133], the first large-scale dataset for NL2LP tasks, in our experiments. The dataset contains 713 training, 99 validation, and 289 test instances. Each instance consists of a natural language problem description paired with a



Content of Prompt

[TEXT OF PROBLEM DESCRIPTION]

Use the above problem descriptions and write the optimization formulation of the problem. Please only give me the model with just one line of explanation for each model element. I don't need the solution. Remove all non-essential spaces. Don't simplify the expressions and don't use latex code or any code in your responses. Use 'x', 'y' and 'z' as variable names.

Please give me the model with the format following the format of the example given below, please do not include any content other than model in your answer:

[RANDOMLY SAMPLED ONE-SHOT EXAMPLE]

Figure 4.5: The prompt templates we applied for four Llama-based language models in Section 4.3.1. The randomly sampled one-shot example is not added for Llama-2-SFT (13B).

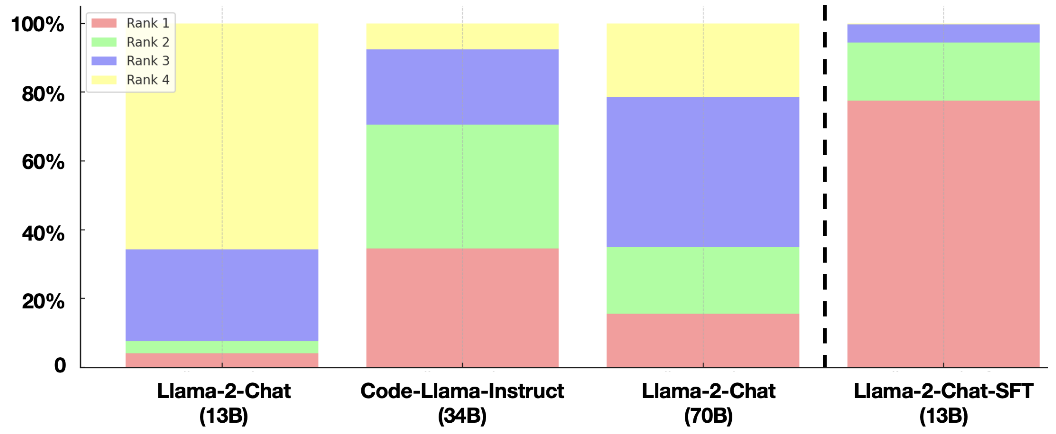


Figure 4.6: Ranking distributions of human judgments on the NL4OPT test set.

ground-truth mathematical program, which has been verified by domain experts as part of the NL4OPT benchmark.

Language Models To assess the effectiveness of evaluation metrics comprehensively, we consider four LLMs, all rooted in the foundational architecture of the widely explored, open-sourced Llama family [149, 138] but with different settings to obtain diverse NL2LP modeling. Specifically, we include three Llama-based models: (1) **Llama-2-Chat (13B)**, (2) **Code-Llama-Instruct (34B)**, and (3) **Llama-2-Chat (70B)**. Additionally, we also fine-tune Llama-2-Chat (13B) with the training set of NL4OPT and name it **Llama-2-SFT (13B)**. Except for Llama-2-SFT (13B), all three other LLMs are under the one-shot in-context learning (ICL) setting, where a validation datapoint is randomly selected and utilized as the one-shot example for all inferences. Figure 4.5 shows the complete prompting formulation that we use identically across all four LLMs in our experiments (Section 4.3.1).

Language Models	**Execution**(↑)	**Canonical**(↑)	**Ours**(↓)
Llama-2-Chat (13B)	0.07 (4)	0.24 (4)	0.52 (4)
Code-Llama-Instruct (34B)	0.35 (2)	0.54 (2)	0.25 (2)
Llama-2-Chat (70B)	0.21 (3)	0.31 (3)	0.41 (3)
Llama-2-Chat-SFT (13B)	0.53 (1)	0.64 (1)	0.14 (1)

Table 4.1: Evaluation scores on the test set via 3 metrics for 4 models. ↑ means larger is better; ↓ means lower is better. Model ranks are in brackets.

Self-Evaluation As a qualitative sanity check, the authors conducted a self-evaluation to examine whether the proposed automatic evaluation metric aligns with human judgment. Since the authors are either experts in Operations Research or have received targeted training to acquire the necessary background knowledge, this self-evaluation can be viewed as a form of preliminary human evaluation. Accordingly, we also refer to it as human evaluation in the following. For each test sample, the authors were provided with the reference program and four anonymized programs generated by the four LLMs. The authors compared each predicted program against the reference program and produced a relative ranking based on perceived deviation from the reference (e.g., $LLM_1 = LLM_2 > LLM_4 > LLM_3$), allowing ties when two predictions appeared equally similar. Figure 4.6 shows that the four LLMs exhibit distinct performance patterns under this human evaluation. Based on the authors' rankings, the overall performance order is *llama-13b-sft* > *code-llama-34b* > *llama-70b* > *llama-13b*.

4.3.2 Experimental Results

Performance of LLMs Table 4.1 presents the evaluation results obtained from our proposed graph-based metric alongside two baseline metrics (i.e., execution accuracy and canonical accuracy in Section 4.1), averaged across 289 test samples. The rankings of LLMs based on these three metrics are consistent with human judgment shown in Figure 4.6, indicating that all three evaluation metrics can effectively assess language models' capability to solve NL2LP to some extent. However, this does not suggest that they align equally well with human judgment. For the majority of test samples, which are either distinctly easy or challenging for specific LLMs, the discrepancies between their predicted and reference programs can be easily quantified by all metrics.

Correlation with Self-Evaluation To more comprehensively measure the alignment between human evaluation and three automatic evaluation metrics for NL2LP, we delve deeper by looking into the ranking match for each test sample. Specifically, we define two types of matching rates: coarse-grained rate (**C-Match**) and fine-grained rate (**F-Match**). Given two ranking lists obtained by human judgment and the automatic metric, we call it "lists exactly match" if these two ranking lists are identical. The C-Match measures

Metrics	C-Match	F-Match
Execution	9 / 289	716 / 1734
Canonical	64 / 289	1336 / 1734
Ours	**178 / 289**	**1641 / 1734**

Table 4.2: Ranking match rate between automatic evaluation metrics and human judgments.

the percentage of instances where the human and automatic ranking lists exactly match. On the other hand, the F-Match decomposes ranking lists into individual ranking pairs and then calculates the match rate at the pair level. As shown in Table 4.2, our proposed evaluation metric consistently achieves the highest match rate with human evaluations at both granularities. This highlights the enhanced reliability and alignment with human judgment of our proposal, especially when conducting evaluation in a pairwise manner (comparing merely two models LLM_1 and LLM_2).

4.4 Limitations and Future Works

Fine-Grained Distinction. Although the proposed metric achieves higher alignment with expert rankings compared to existing alternatives, it is important to note that fine-grained distinction among LP formulations is inherently limited by the variability of human judgments [36, 92].

In practice, different evaluators may prioritize different aspects of a formulation, such as modeling elegance, numerical stability, structural simplicity, or solver performance. As a result, even among domain experts, complete agreement on subtle ranking differences is not always observed. This intrinsic subjectivity makes defining a perfectly consistent fine-grained ground truth fundamentally challenging.

Therefore, the alignment rate should be interpreted not as an absolute upper bound on metric quality, but rather as a relative indicator under evaluator-dependent variability.

Therefore, one promising direction for future research is to develop a learning-based evaluation metric that leverages human preference data. Instead of relying on predefined structural distances or execution-based comparisons, such a metric could be trained from pairwise or listwise human rankings of LP formulations [22, 108]. By learning a preference model over formulation graphs, the evaluation function may capture nuanced criteria that are difficult to formalize explicitly, such as modeling clarity or structural coherence.

This approach is analogous to preference learning or reward modeling techniques used in large language model alignment. By aggregating judgments from multiple experts and modeling inter-evaluator variability, a learned metric may provide more fine-grained and adaptive distinctions than rule-based metrics.

We leave the systematic collection of human preference data and the design of such a learning-based evaluator as an interesting direction for future work.

Reliability Evaluation A limitation of the proposed evaluation in this chapter is that it primarily assesses *single-shot accuracy* and *relative quality* among alternative formulations (e.g., which formulation is more human-aligned), while largely overlooking the model's *reliability* and *output variability.* In practice, LLM-based formulation generation is inherently stochastic: a model may occasionally produce an excellent formulation by chance, which can inflate the observed score if reliability is not accounted for. This motivates a future research direction to explore whether and how output variability could serve as a measure of model reliability.

Variability has long been studied in the machine learning literature under the lens of uncertainty quantification and calibration, where predictive variance and confidence miscalibration are well-known phenomena [72]. Classical approaches such as Monte Carlo dropout [59] or ensemble-based uncertainty estimation [96] aim to approximate predictive uncertainty for point-prediction models.

However, large language models introduce additional dimensions of variability beyond traditional predictive variance. Unlike conventional classifiers, LLMs generate structured outputs in natural language. Variability may arise not only from differences in final answers, but also from alternative surface expressions of the same answer, different reasoning trajectories, or distinct intermediate solution paths. These forms of variation are often measurable and, to some extent, controllable through training pipelines such as supervised fine-tuning or preference-based optimization (e.g., RLHF/DPO), which reshape the distribution of generated outputs.

More challenging, however, is a deeper source of variability that is difficult to disentangle: whether the input query is merely phrased in an unfamiliar way that "surprises" the model, or whether it genuinely lies outside the model's effective knowledge boundary. Existing uncertainty estimation techniques primarily capture output dispersion but do not reliably distinguish between these two causes. From a reliability perspective, the latter—knowledge boundary violation—is arguably the more meaningful notion of model confidence. Developing principled methods to identify when a query falls outside the model's knowledge coverage remains an open and challenging research direction.

Difficulty-Aware Evaluation. A limitation of the current study is that the benchmark mainly consists of relatively simple, textbook-style LP problems, where full formulations with numerical coefficients are explicitly specified. While such instances provide a clean evaluation setting, they do not fully capture the spectrum of modeling complexity encountered in practice. As discussed in the survey [53], OR problems can be broadly categorized into multiple levels: textbook-level LP problems, where complete formulations with explicit numerical coefficients are provided, and higher-level modeling tasks that involve abstract reasoning, implicit assumption inference, or parameterized symbolic formulations.

This observation raises the broader question of how "difficulty" should be defined for LLM-based modeling. Rather than being a single scalar notion, difficulty may be inherently multi-dimensional. Beyond the abstraction level of the problem statement, it may depend on (i) the degree of implicit modeling assumptions and external knowledge required—i.e., how much domain knowledge is necessary but not explicitly specified in the prompt to construct a valid model—and (ii) the minimum reasoning complexity needed to derive the final formulation, such as the number of essential inference steps (or sub-decisions) required to define variables, objectives, and constraints. Developing a difficulty-aware capability profiling benchmark along these dimensions is an interesting direction for future research.

4.5 Conclusion

In this chapter, we have addressed a critical challenge in the evaluation of the Natural Language to Linear Program (NL2LP) task: the lack of metrics that faithfully reflect expert-level, human-aligned judgment. While existing metrics—Canonical Accuracy and Execution Accuracy—offer computationally tractable evaluation, we demonstrated through concrete counterexamples that they suffer from structural brittleness and functional insensitivity, respectively. These shortcomings can lead to misleading performance assessments, particularly when evaluating large language models (LLMs) in optimization-aware applications.

To bridge this gap, we proposed a graph-based evaluation metric grounded in graph edit distance (GED). By first converting LPs into a permutation-invariant bipartite graph representation and then measuring the normalized GED between predicted and reference graphs, our metric directly reflects the number and severity of semantic modeling errors. Experiments on the NL4OPT dataset with four Llama-based models showed that our metric not only produces sensible aggregate rankings but also achieves substantially higher agreement with expert rankings at both coarse-grained and fine-grained levels, indicating a closer alignment with human judgment. Moreover, our proposed metric provides interpretable diagnostic signals: high scores naturally point to where and how a predicted formulation diverges from the reference (e.g., omitted constraints, misassigned coefficients), offering actionable feedback for model refinement.

Looking ahead, our evaluation framework can be further improved in the future. First, the graph-based framework could be extended beyond pure LPs to mixed-integer and non-linear programs, enabling human-aligned evaluation for a broader class of optimization models. Second, approximate GED algorithms could be leveraged to scale evaluation to larger, real-world instances without sacrificing too much fidelity. Third, the metric could be refined by incorporating additional semantic information about the two programs, for example, by comparing the natural-language explanations of their constraints or objectives.

Meanwhile, our proposed metric can further boost the LLM field. By providing structured, expert-auditable signals, our metric can serve as a foundation for more trustworthy NL2LP benchmarks and for feedback-driven training schemes, ultimately guiding the development of LLMs that are not only syntactically competent but also mathematically reliable.

This chapter completes the "ML enhances OR" part of the thesis by focusing on how to *evaluate* LP formulations generated from natural language in a way that aligns with expert judgment, thereby strengthening the modeling stage of the optimization pipeline. In the next chapter, we turn to the complementary perspective, where *OR enhances ML*: we treat LLM serving systems themselves as the objects of optimization and study how ideas from OR can be used to enhance LLM serving on NPU clusters.

Chapter 5

Joint Design of Algorithms and Systems for Multi-SLO Serving and Fast Scaling

Modern large language model (LLM) serving systems have become increasingly critical for a wide range of applications, from interactive chatbots to large-scale document summarization [69, 178, 2]. These applications often involve requests with highly variable input and output lengths, as well as diverse service-level objectives (SLOs) for latency requirements. As illustrated in Figure 5.1, real-time chat applications require very low latency to ensure a responsive user experience, whereas batch generation tasks, such as document summarization, can tolerate longer startup times but demand higher throughput. This heterogeneity motivates the design of serving systems and scheduling algorithms capable of handling diverse workloads while balancing the needs of latency-sensitive and latency-tolerant requests and optimizing resource efficiency.

5.1 Background and Problem Definition

LLMs & LLM Service Large Language Models (LLMs) are deep neural networks trained on massive text corpora to perform diverse natural language processing tasks, including text generation, summarization, and question answering. Examples include Alibaba's *Qwen* [10]

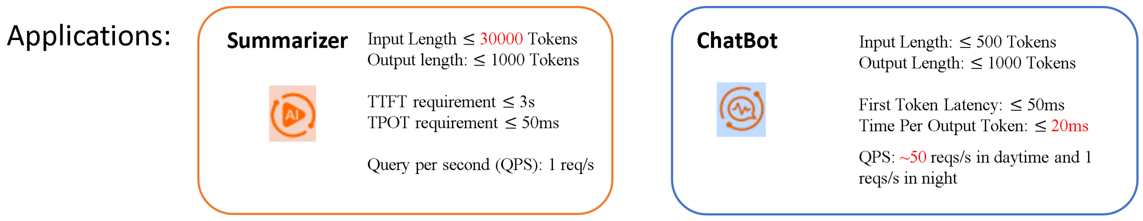

Applications:
Summarizer
Input Length ≤ 30000 Tokens
Output length: ≤ 1000 Tokens
TTFT requirement ≤ 3s
TPOT requirement ≤ 50ms
Query per second (QPS): 1 req/s
ChatBot
Input Length: ≤ 500 Tokens
Output Length: ≤ 1000 Tokens
First Token Latency: ≤ 50ms
Time Per Output Token: ≤ 20ms
QPS: ~50 reqs/s in daytime and 1 reqs/s in night

Figure 5.1: Diverse LLM application scenarios with varying lengths, latency requirements, and load intensity.

and Meta's *LLaMA* [149], representing state-of-the-art open-source models widely used in research and industry. LLM inference, where a single instance executing the LLM to serve a single request, typically consists of two computationally distinct phases:

1. *Prefill phase:* Generates the first token using full-sequence attention. This phase is *latency-sensitive* and compute-intensive.

2. *Decode phase:* Generates subsequent tokens incrementally using cached key–value states. This phase is *throughput-oriented* and benefits from batching and sparsity techniques [94].

An *LLM service* follows a client–server architecture: clients submit text requests, and a cluster host one or more serving *instances* on accelerators (e.g., GPUs or NPUs). Multiple requests may concurrently arrive at the same server, and the service typically groups them into *batches* for prefill or decode execution to improve hardware efficiency.

The latency of such prefill and decode batches can be analytically modeled. DistServe [177] models prefill latency with a quadratic regression in Appendix A.2 of this paper, but both our NPU profiling and DistServe's released implementation show that the quadratic term is consistently negligible. This is because the typical prompt length of current benchmark is around 2k tokens, which lies within the locally linear operating region of the latency curve. In this regime, the system is primarily memory-bound rather than compute-bound, and device throughput remains stable. Intuitively, the prefill stage is dominated by dense matrix multiplications that are highly parallelizable. As long as hardware resources are not saturated, the computational cost scales approximately proportionally with the number of prompt tokens. Consequently, the quadratic coefficient reported in prior work has minimal practical impact in our operating range.

To keep our subsequent design simple and consistent, we therefore adopt linear performance models for both stages. For a prefill batch containing n_{token} prompt tokens, we approximate

$$E_p = a + b\, n_{\text{token}}. \tag{5.1}$$

Similarly, following the linear formulation used in DistServe [177] (Appendix A.3), we model the per-iteration decode latency as

$$E_d = a' + b'_1\, n_{\text{token}} + b'_2\, B, \tag{5.2}$$

where n_{token} denotes the total number of active tokens (including both prefilled and decoded tokens) in the current decode step, i.e., the accumulated KV-cache length for the batch, and B is the batch size.

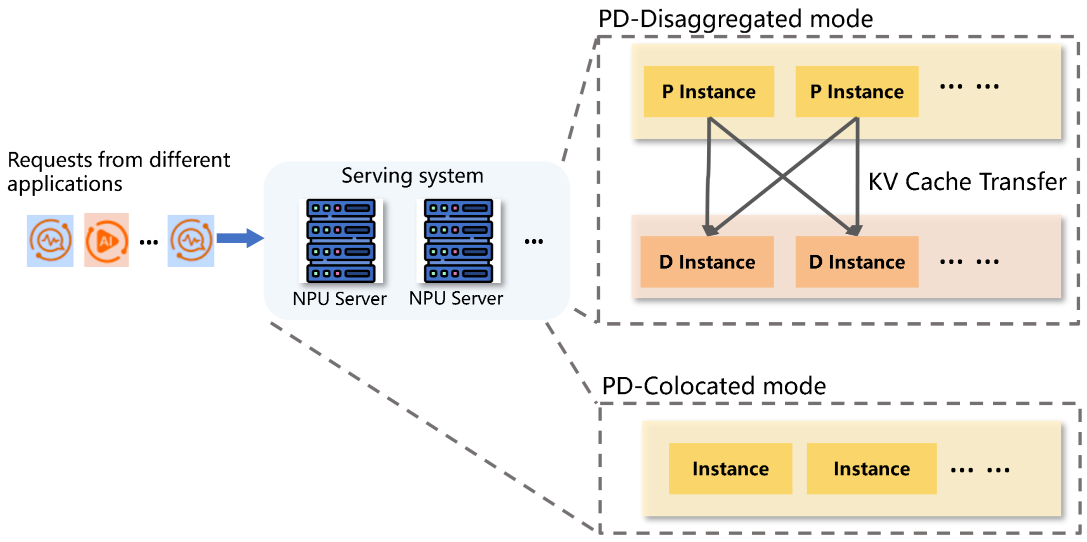


Figure 5.2: The LLM serving system could be deployed in colocated or disaggregated execution modes.

Multi-SLO serving. Many real-world applications impose heterogeneous latency requirements across requests. Users may specify two key latency-related service-level objectives (SLOs):

1. *Time-to-First-Token (TTFT):* The maximum allowed latency before receiving the first output token.

2. *Time-per-Output-Token (TPOT):* The maximum allowed average latency per generated token.

Serving such multi-SLO workloads is challenging because actions that optimize one metric can easily hurt another. Without explicit multi-SLO awareness, a system may over-provision resources for low-priority requests or starve latency-critical ones, leading to degraded user experience or inefficient accelerator utilization [94, 171, 29].

Colocated vs. Disaggregated Execution Modes As shown in Figure 5.2, LLM serving systems can adopt either colocated or disaggregated execution modes, each with distinct trade-offs.

In colocated mode [94], prefill and decode run on the same devices, avoiding inter-node communication and achieving low per-request latency. However, colocated deployments can suffer from *interference*: prefill-heavy requests may repeatedly preempt ongoing decoding, pushing TPOT beyond its SLO. Hardware utilization can remain suboptimal because the prefill–decode resource ratio is not controllable nor SLO-aware, and the system suffers from fragmented execution, frequent context switches, and even GPU–CPU memory swaps. Despite this under-utilization, users may still experience unsatisfactory quality of service due to TPOT violations.

Disaggregated mode [177] separates prefill and decode onto different nodes or device pools, enabling independent scaling and improving cluster-level resource utilization. This separation also provides greater flexibility and cost efficiency for large-scale deployments. However, it introduces communication overhead: KV caches must be transferred from prefill to decode instances before token generation begins. For latency-sensitive requests, this additional transfer can violate strict SLOs. Furthermore, naive dispatching in disaggregated clusters can cause unpredictable queuing and contention, leading to SLO violations in production environments.

Recent analysis shows that the relative advantage of aggregation versus disaggregation depends strongly on the strictness of TTFT and TPOT requirements, with neither mode consistently outperforming the other across all settings [154]. Overall, neither mode universally dominates. Both approaches are adopted across open-source and commercial systems, and practical LLM serving designs should remain compatible with colocated and disaggregated execution to support diverse deployment environments.

Ascend NPU Ascend Neural Processing Units (NPUs) are domain-specific accelerators designed for large-scale matrix and tensor operations common in deep learning, often delivering higher performance and energy efficiency than general-purpose CPUs or GPUs [105]. Built on the Da Vinci architecture [106], each AI core integrates a cube tensor engine, vector unit, and scalar unit, all optimized for neural network computation—unlike GPUs, which were originally crafted for graphics and then adapted for general compute. The accompanying CANN software stack [182] supports popular ML frameworks (e.g., MindSpore, PyTorch) and compiles workloads into optimized operator graphs for the hardware. Ascend NPUs can be deployed either in standalone servers or as part of large-scale clusters, such as the CloudMatrix platform, which provides high interconnect speed and bandwidth [84, 179]. As ASIC-based specialized accelerators, NPUs generally offer advantages over GPUs in terms of hardware cost and energy efficiency, potentially leading to lower total cost of ownership in production deployments. However, these intrinsic hardware benefits are orthogonal to our contributions and are not explicitly reflected in our results. Although our evaluation is conducted on NPUs, the proposed scheduling framework is architecture-agnostic and can be applied to both NPUs and GPUs, as it operates at the system level and does not rely on device-specific microarchitectural features.

5.1.1 Challenges

Multi-SLO Scheduling Challenges To handle increasing load, large-scale LLM serving systems are deployed across clusters of multiple instances. However, simply placing a standard load balancer (e.g., Round-Robin or Least-Connections) is insufficient for SLO-aware serving. This naive approach fails to address the state-dependent nature of LLM serving and lacks the mechanisms to prioritize requests based on their remaining SLO budget.

Another way to serve multi-SLO requests is by dedicating instances to different request types or latency classes. While this avoids interference across classes, it introduces several new issues: (1) resource under-utilization, as some instances may remain lightly loaded while others accumulate long queues; (2) operational inflexibility, the deployment must be reconfigured when new latency classes are introduced; and (3) increased provisioning cost, since maintaining separate capacity for every request class results in over-provisioning during low-demand periods. Furthermore, this static separation does not solve the core problem: within a single instance, request urgency changes over time, and the scheduler must continuously decide.

A second challenge arises from the nature of batching. Continuous batching improves local device throughput, but its view is strictly local. Scheduler within each instance treats the stream of incoming requests as fixed; it can only decide how to batch what it receives, not which requests it should receive in the first place. It lacks the global context required to make trade-offs between multiple instances. For example, a latency-sensitive request may be queued behind long-context requests on one instance, even though another instance in the cluster has available compute capacity. Without a global scheduler that understands the current queue depth, memory fragmentation, and request types across all instances, the cluster suffers from Head-of-Line (HOL) blocking, where high-priority requests get stuck behind long-running ones on a selected instance.

Elastic Scaling Challenges LLM workloads exhibit significant variability in request rate, lengths, priorities, and latency SLOs. In large-scale cloud deployments, clusters must handle dynamic request patterns and fluctuating workloads. Therefore, the challenges of multi-instance scheduling are compounded by the need for elastic scaling.

Cold-Start Penalty: Scaling is not instantaneous. New instances incur significant delays because the newly provisioned instances need to instantiate the inference engine and load model weights from storage.

Cost vs. SLO Attainment Tradeoff: Scaling decisions must balance the risk of SLO violations against the cost of idle resources. Over-provisioning masks scheduling inefficiencies but increases operational costs. Conversely, conservative scaling reactions risk violating latency-critical SLOs. A robust system must jointly address *micro-level scheduling* (routing requests to the best instance) with *macro-level scaling* (detecting when current instances will no longer suffice).

Existing LLM serving systems do not fully address the operational complexities of large-scale cloud environments. Their reliance on naive round-robin dispatching and static scaling fails to account for dynamic queuing, multi-stage execution, and heterogeneous SLOs, leading to unpredictable latency, inefficient utilization, and higher operational costs. These limitations motivate the design of HyperFlexis, a unified LLM serving system for production-scale deployments.

5.1.2 Problem Definition

In this section, we first define the multi-SLO serving scenario and the SLO-aware scheduling problem. Because the scheduling dynamics are highly event-driven and difficult to capture in a clean mathematical program, we then identify core principles that guide the system and algorithm designs.

The serving system receives a continuous stream of requests $R = \{r_1, r_2, \dots, r_m, \dots\}$. Each request r_m arrives at certain timestamp and specifies its prompt, maximum output length, and heterogeneous SLOs, including TTFT and TPOT. Meanwhile, the system should support elastic scaling, adding instances on demand up to a maximum of N, constrained by available resources.

At every event boundary, either a new request arrival or the completion of a prefill or decode iteration by any instance, the scheduler should make two types of decision: (i) *dispatching*: whether and how to assign pending requests to instances, and (ii) *scaling*: whether to scale out or scale in, and which instances to deactivate when scaling in. These decisions must balance two conflicting objectives: minimizing infrastructure cost while maximizing user satisfaction. Specifically, infrastructure cost is given by

$$\text{Cost} = \sum_{n=1}^{N} (\text{active time of instance } n) \cdot \text{UnitCost}, \tag{5.3}$$

where UnitCost denotes the rental price per-second for an NPU instance. Meanwhile, user satisfaction is measured by SLO attainment, defined as the fraction of requests that meet both the TTFT and TPOT requirements:

$$\text{Attainment} = \frac{1}{M} \sum_{m=1}^{M} \mathbb{I}_{\text{TTFT}}(r_m)\, \mathbb{I}_{\text{TPOT}}(r_m). \tag{5.4}$$

Apart from SLO attainment, end-to-end (E2E) completion time for a request r_m provides a complementary view of user-perceived latency, defined as $\text{E2E}(r_m) = T_{\text{complete}}(r_m) - T_{\text{arrival}}(r_m)$.

As discussed in Section 5.1.1, cost and satisfaction metrics reveal an inherent tradeoff. An effective scheduler must navigate this tension by admitting requests only when their SLOs can be met while avoiding unnecessary instance activity. Although this problem resembles a constrained scheduling and allocation task, a full end-to-end optimization is not tractable in practice. Several factors drive this complexity. First, decisions are event-driven—triggered by request arrivals, completions, or the completion of an instance's iteration—rather than occurring on a fixed rolling horizon [151], resulting in an irregular decision timeline. Second, it is difficult to accommodate future uncertainty, as output lengths are unknown at arrival and workload arrivals follow highly stochastic patterns. Third, batching forms a circular dependency: given the current queue delays and SLO requirements, the scheduler chooses a batch size; this batch size determines the per-iteration latency, which then feeds back into future queue delays. This closed feedback loop has no simple analytic form.

These challenges make an explicit formulation infeasible. We therefore distill a set of core guiding principles to drive both the algorithmic and system design: (1) At *micro-level dispatching*, the system prioritizes protecting in-flight SLOs, ensuring that admitted tasks can continue progressing toward their SLO requirements before any additional load is accepted. Meanwhile, new requests are admitted greedily, whenever it is safe, to maximize hardware utilization and thereby reduce cost. (2) At *macro-level scaling*, Elastic scaling serves as a fallback when no safe placement exists, enabling the system to adjust resources dynamically while preserving SLO compliance. These principles are instantiated naturally in both colocated and disaggregated deployment modes, providing a practical framework for building an SLO-aware serving system that balances user satisfaction and cost efficiency in dynamic workloads.

5.2 System and Algorithm designs of HyperFlexis

Assumptions To make the scheduling problem analytically tractable while retaining practical relevance, we adopt the following assumptions.

1. *Homogeneous NPU Devices.* All NPUs are assumed to be homogeneous in terms of computational throughput, memory capacity, and interconnect bandwidth. This allows the scheduler to treat devices as interchangeable resources and focus on workload allocation decisions without device-specific optimization. Extending the proposed scheduler to heterogeneous accelerator clusters is an interesting and practically relevant direction for future research [135, 172].

2. *Restricted Preemption.* Full migration of active decoding requests across devices is not considered [173]. Instead, we allow two restricted forms of preemption, where prefill computation may preempt decoding computation: (i) compute-only preemption, where compute resources are reallocated while KV states remain in place; and (ii) KV offloading preemption, where KV cache may be temporarily offloaded to host memory. Cross-device state migration is excluded due to its substantial transfer overhead and implementation complexity. We disable migration-based preemption to keep the problem definition simple and focused on core scheduling dynamics. Accordingly, migration decisions are not incorporated into the current model. Extending the scheduler to jointly optimize migration and preemption is an interesting direction for future

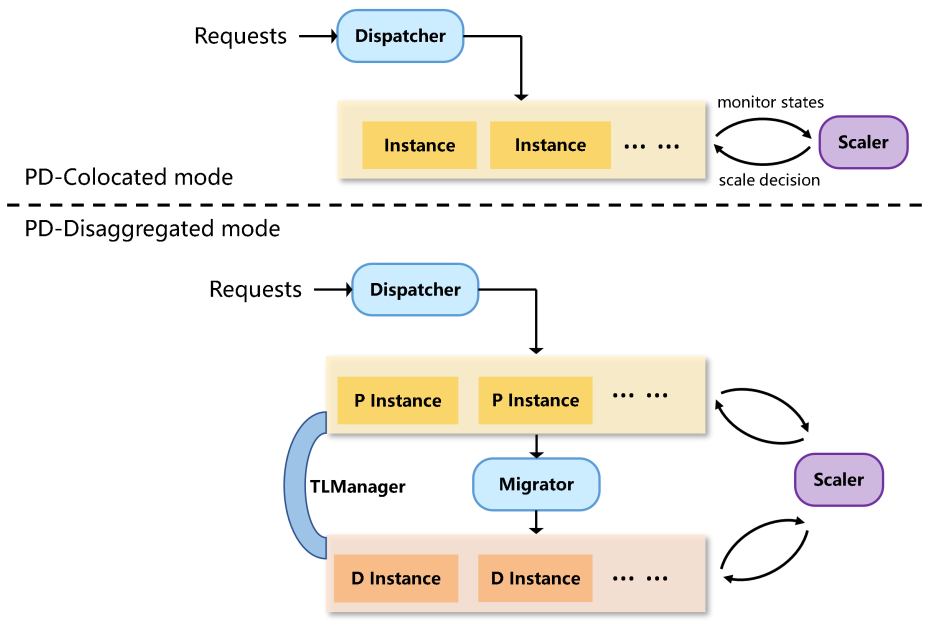


Figure 5.3: Overview of HyperFlexis's LLM serving architecture.

work, as it would enable additional system capabilities such as workload consolidation and memory defragmentation.

3. *Fast Autoscaling.* We assume the availability of a fast autoscaling mechanism, which is implemented in our system, such that scaling operates at a timescale faster than workload fluctuations. This prevents transient capacity shortages that would otherwise lead to SLO violations regardless of the scheduling policy. Under this assumption, we can focus on short-term scheduling decisions without conflating them with long-term capacity provisioning effects.

4. *Latency Estimation via Profiling.* The execution latency of each scheduling step is assumed to be approximated by an offline profiled performance function that maps batch size and sequence length to step latency. This abstraction enables analytical reasoning without modeling low-level hardware execution details.

Architecture As shown in Figure 5.3, HyperFlexis comprises five primary components: the *Dispatcher*, *Scaler*, *Monitor*, *TransferLink Manager (TLManager)*, and *Migrator*. The Dispatcher handles request scheduling, while the Scaler adjusts the number of active instances in response to demand. Both components rely on runtime statistics collected by the Monitor to guide their decisions. In PD-disaggregated mode, the Migrator forwards prefilled requests to decode instances, and the TLManager maintains the instance-linking topology that must be pre-established for KV-cache transfers.

Next, we describe how the macro- and micro-level principles are instantiated in these components. Specifically, the micro-level dispatching principle is embodied in the logic of the Dispatcher modules under both execution modes and in the Migrator for PD-disaggregated mode, while the macro-level scaling principle is realized by the Scaler.

5.2.1 SLO-Aware Dispatching Mechanism

Our scheduling policy follows a strict priority order: we first protect in-flight requests on each instance so they can meet their SLOs, and only then admit new arrivals—dispatching urgent ones immediately whenever this safety condition is satisfied. Admitting a new request can interfere with ongoing executions by contending for compute or memory bandwidth, increasing latency and risking deadline violations. To prevent such interference, we introduce the notion of maturity time, which marks when an instance has progressed far enough on its current workload to safely absorb additional demand. Before this point, adding new requests could jeopardize existing SLOs; after it, the instance can accommodate extra load without harmful slowdowns. The dispatcher tracks each instance's maturity time and admits new requests only when the corresponding safety condition is met, thereby preserving SLOs while still starting urgent work as early as possible. When no instance is currently safe to admit more load, requests are held in the queue and deferred until scaling provisions additional capacity, meaning that scaling acts as the fallback mechanism.

We next introduce our dispatching logic at high level. This logic is shared by the Dispatcher in both execution modes and by the Migrator in the PD-disaggregated mode. At a high level, the dispatcher repeatedly performs the following steps in a coroutine, illustrated in Figure 5.4 (a).

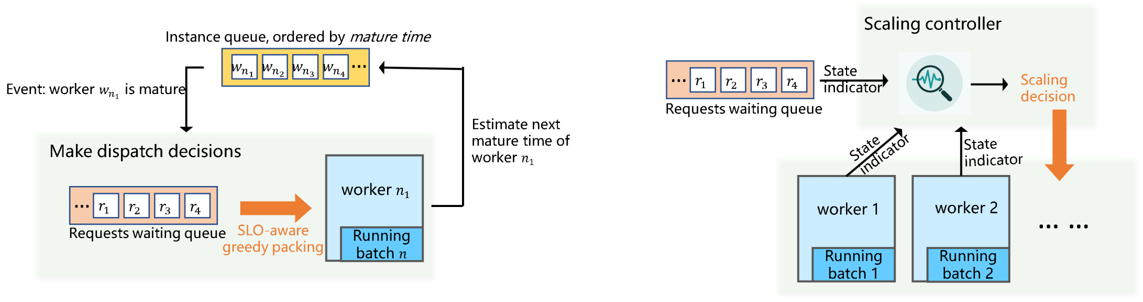
1. Select the instance with the closest maturity time—the earliest future point at which admitting new requests will not jeopardize the SLOs of its in-flight batch.

2. Compute the instance's current safe token budget.

3. Scan the global request queue and select the most urgent requests that fit within this budget.

4. Dispatch these requests and update the instance's next maturity time according to the newly predicted workload.

In the following, we tailor the control policy to the colocated and disaggregated deployment modes, which exhibit distinct interference patterns: in colocated mode, new prefill work directly contends with ongoing decodes, creating cross-stage interference; in disaggregated mode, the stages are isolated and interference occurs within each stage, as new arrivals are added to the next batch, making it larger and slower for all requests in that batch.



(a) Algorithm steps for dispatching module

(b) Algorithm steps for scaling module

Figure 5.4: The high-level algorithm steps instantiated from the micro-level dispatching and macro-level scaling principles.

Colocated Mode In PD-colocated deployment mode, prefill and decode share the same devices. This coupling creates a critical challenge: a newly arriving request triggers a prefill phase that can immediately interfere with ongoing decode work, elongating its token generation latency and violating TPOT targets. The dispatcher's primary responsibility in this mode is therefore to *protect ongoing decode requests* from such interference, while admitting urgent new requests before violating their TTFTs.

To achieve this, the dispatcher maintains centralized visibility of all instances and regulates when each instance may accept new prefills by computing a *maturity time*. Determining this maturity time requires estimating how many new requests an instance can safely accommodate. We capture this with a *token budget*, which represents the maximum amount of prefill-induced interference the instance can absorb while still keeping all ongoing decode requests within their TPOT targets. Intuitively, this budget reflects the slack accumulated through previous decode progress and the ability to amortize small early interruptions over the upcoming uninterrupted decode window before maturity. Once the token budget is known, the dispatcher can determine how many—and which—new requests can be admitted without violating SLO constraints.

Specifically, in Step 2, the token budget determines the maximum number of additional prefill tokens an instance can absorb without violating SLOs. This budget is derived from the instance's instantaneous state, specifically the *waiting time until maturity* and the TTFT requirement. This waiting time—computed in Step 4 of the previous iteration—comprises up to two components: the remaining time to complete the current prefill iteration (R_p, nonzero if the instance is still in the prefill stage), and the remaining uninterrupted decode window (R_d) required to ensure in-flight decode requests meet their TPOT constraints. If the candidate batch is with n_{budget} prompt tokens, the estimated prefill latency is $E_p(n_{\text{budget}}) = a + b\, n_{\text{budget}}$. The TTFT constraint requires $R_p + R_d + E_p(n_{\text{budget}}) \leq \text{TTFT}$. Solving for

n_{budget} yields the instance's safe token budget:

$$n_{\text{budget}} \leq \frac{\text{TTFT} - R_p - R_d - a}{b}. \tag{5.5}$$

In Step 3, the dispatcher scans the global request queue and admits the most urgent requests whose total prompt tokens do not exceed the token budget. This yields the actual prefill batch for the instance, with a total of n_{actual} tokens.

In Step 4, launching the batch incurs prefill latency $E_p(n_{\text{actual}})$, which interrupts decoding. Any accumulated decode slack S_{acc}—time saved from prior decode iterations running faster than TPOT—can offset this interference. The net delay to amortize is thus $\max\{0,\, E_p(n_{\text{actual}}) - S_{\text{acc}}\}$. To preserve TPOT, the instance must execute enough uninterrupted decode iterations to absorb this residual delay. With per-iteration decode latency E_d, the slack per iteration is $\text{TPOT} - E_d$, yielding the required number of iterations:

$$k = \left\lceil \frac{\max\{0,\, E_p(n_{\text{actual}}) - S_{\text{acc}}\}}{\text{TPOT} - E_d} \right\rceil. \tag{5.6}$$

The instance's new maturity time is then

$$T_{\text{mat}}^{\text{new}} = t_{\text{now}} + E_p(n_{\text{actual}}) + k \cdot E_d\ , \tag{5.7}$$

after which it becomes eligible for new requests.

Disaggregated Mode In the PD-disaggregated deployment mode, prefill and decode run on separate instance pools, so the high-level algorithm from Figure 5.4 (a) is instantiated twice: once by the *Dispatcher* for the P stage and once by the *Migrator* for the D stage. Both components follow the same four-step structure, with only Step 2 and Step 4 differing from the colocated mode due to a different interference model. We therefore focus on explaining these two steps for each stage.

P-stage Dispatcher. Prefill batches are non-interruptible under static batching. Consequently, the *Dispatcher* uses a simplified maturity definition: an instance becomes mature only when its current prefill batch completes. The different steps are:

- *Step 2 (token budget).* Starting from an empty instance, its safe token budget is determined solely by the strictest TTFT in the priority-ordered request queue. The intuition is that lower-priority requests have looser TTFT requirements, so satisfying the strictest TTFT automatically ensures that remaining requests meet their TTFT constraints.

- *Step 4 (maturity time).* After dispatching a new batch, the instance's next maturity time is set to the estimated completion time of this prefill batch, since it cannot be interrupted mid-way.

Algorithm 2: SLO-aware Dispatcher for Colocated Mode

```
1  Attribute Request Priority Queue $Q_R$: requests ordered by urgency (TTFT target,
     and then TPOT target).
2  Attribute Instance Priority Queue $Q_W$: instances ordered by maturity time.
3  Function OnRequestArrive(request $r$):
4      Insert $r$ into $Q_R$ according to priority;
5  Function Init():
6      Start Dispatcher() as a coroutine in the event loop;
7  Function Dispatcher():
8      while True do
              // Step 1: Select instance with earliest maturity time
9          $w \leftarrow Q_W$.pop() Synchronize the state of instance $w$ (time until mature,
             in-flight decode requests, accumulated slack $S_{\text{acc}}$, etc.);
              // Step 2: Compute safe token budget
10         $R_p \leftarrow$ remaining prefill time if $w$ is in prefill; else 0;
11         $R_d \leftarrow$ remaining uninterrupted decode window needed to meet TPOT;
12         $n_{\text{budget}} \leftarrow$ token budget per Eq. (5.5) using $(w, R_p, R_d)$, capped by $w$'s
             free-token memory capacity;
              // Step 3: Select requests within token budget
13         $\mathcal{R} \leftarrow \emptyset$;
14         for request $r \in Q_R$ in priority order do
15             if $tokens(r) + tokens(\mathcal{R}) \leq n_{budget}$ then
16                 Add $r$ to $\mathcal{R}$;
              // Step 4: Dispatch and update maturity time
17         if $\mathcal{R} \neq \emptyset$ then
18             $n_{\text{actual}} \leftarrow$ tokens($\mathcal{R}$);
19             $k \leftarrow$ computed from Eq. (5.6) with $(w, n_{\text{actual}}, S_{\text{acc}})$;
20             $w$.mature_time $\leftarrow$ computed from Eq. (5.7) with (now, $w, n_{\text{actual}}, k$);
21             Dispatch $\mathcal{R}$ to $w$;
22         $Q_W$.push($w$);
23         await until next instance matures;
```

D-stage Migrator. Under continuous batching, decode progress is iterative, and each iteration boundary provides a natural decision point, enabling fine-grained control. The *Migrator* makes the dispatching decision. The different steps are:

- *Step 2 (token budget).* The safe token budget depends primarily on the state of the decoding instance. When the instance already has an active decode batch, its effective TPOT constraint is determined by the strictest TPOT among the in-flight requests, and the budget is computed to ensure that admitting additional tokens will not violate this constraint. If the instance is temporarily idle, it instead uses the strictest TPOT from the priority-ordered queue of prefilled requests to determine its token budget.
- *Step 4 (maturity time).* After dispatching, maturity is updated to the end of the next decoding iteration, enabling a decision at every iteration boundary.

Simple strategies such as round-robin or dispatching to the least-loaded instance do not consider instance states or request-level SLO requirements, and therefore may admit requests even when the resulting latency cannot meet SLO constraints. In contrast, our dispatching logic is SLO-aware and avoids these limitations.

5.2.2 Scaling Mechanism

Dynamic scaling is essential for maintaining TTFT/TPOT attainment under fluctuating load. Insufficient resources lead to queueing delays and propagated SLO violations, whereas excessive provisioning increases cost. This section presents our design of *algorithmic* and *system-level* designs that enable responsive and cost-efficient scaling. The algorithm determines when and how to resize prefill and decode capacity, and the system mechanisms ensure that scaling decisions can be executed quickly and reliably.

Algorithmic Design At a high level, the scaling logic follows the abstract procedure shown in Figure 5.4 (b). Every τ seconds, the *Scaler* inspects the state of the global request queue and the state of all active instances, and derives three indicators to assist scaling decision.

The first indicator captures *system load* over a *long* (i.e., 10 s) time horizon. The Scaler maintains a fixed time window $W = 10\,\text{s}$ of enqueue and dequeue timestamps from the global request queue to derive smoothed arrival and completion rates, λ_{in} and λ_{out}. To avoid spurious scaling actions during startup or when only a small number of events are available, the Scaler suppresses rate estimates until the window contains sufficient samples. The rate-based load estimator is then defined as

$$f_1 = \frac{\lambda_{\text{out}}}{\lambda_{\text{in}}},$$

where $f_1 < 1$ indicates that completions lag behind arrivals (pressure to scale out), and $f_1 > 1$ indicates that processing capacity exceeds demand. This signal captures load over a longer time horizon.

Second, we use a complementary *SLO-aware* system load indicator that reacts more quickly to *short-term* fluctuations. The Scaler examines the current request queue and computes the normalized waiting time of each pending request relative to its SLO target. The instantaneous SLO pressure is defined as

$$f_2 \;=\; \frac{1}{N} \sum_{r \in \text{queue}} \frac{\text{waiting\_time}(r)}{\text{SLO}_r},$$

where N is the number of queued requests. Larger values of f_2 indicate more urgents requests in queue and increasing risk of SLO violations.

The third indicator captures the memory load of each instance. For every instance n, the Scaler maintains a coarse utilization estimate based on its KV-cache usage:

$$f_3 \;=\; \frac{\text{used\_blocks}_n}{\text{total\_blocks}_n},$$

where used_blocks_n accounts for KV-cache blocks held by in-flight requests and those reserved for pending requests in the instance's local queue.

These three indicators drive a threshold-based scaling policy. f_1 serves as the primary rate signal: when $f_1 < \epsilon_{\text{out}}$, the system interprets sustained overload and activates an additional instance. Conversely, when $f_1 > \epsilon_{\text{in}}$, and more than one instance is active, the Scaler scales in by selecting the instance with the smallest f_3, as it carries the least KV-cache commitment. Meanwhile, the SLO-aware signal f_2 provides fast reaction to bursts: if $f_2 > \epsilon_{\text{wait}}$, the system proactively scales out even when f_1 remains moderate, in order to satisfy user SLOs under short-term fluctuations. For these three hyper-parameters, by default, we use a smoothing window of $W = 10\,\text{s}$, a rate-based threshold pair of $\epsilon_{\text{out}} = 0.7$ and $\epsilon_{\text{in}} = 1.1$, and an SLO safeguard threshold of $\epsilon_{\text{wait}} = \frac{1}{4}$ (i.e., $\frac{1}{4}$ of the SLO target).

In PD-disaggregated mode, prefill and decode stages experience different workload patterns, so the Scaler evaluates the load of each stage independently. As a result, the two instance pools may diverge in demand—for example, the prefill pool may scale in while the decode pool scales out—a situation that does not arise in the colocated case. Naively terminating and relaunching instances in such scenarios would introduce unnecessary churn. Instead, the Scaler makes a *role switching decision*, for example, a prefill instance can be switched into a decode instance by reassigning its NPU resources to the decode pool.

System Design We design the system to support fast scale-out, enabling new instances to be instantiated with minimal cold-start latency.

Each NPU server maintains a warm-up pool of lightweight instances that initialize the runtime but defer model weight loading. Every instance embeds a *WeightManager* that tracks the device memory layout of model weights, enabling any running instance to serve as a source for weight transfer. During scale-out, the Scaler selects one running instance as the source, and triggers device-to-device `send_weight`/`receive_weight` operations. Tensors are transferred in a device-aligned manner, and the target instance joins the running pool immediately after transfer completion. If the transfer fails, the system falls back to disk loading to preserve correctness. This design bypasses the disk-loading path and substantially reduces cold-start latency.

For PD-disaggregated mode, the *TLManager* manages the communication-group topology required for enabling KV-cache transfer between prefill and decode instances. As part of the *role-switch mechanism*, when the Scaler reassigns an instance from prefill to decode (or vice versa), the TLManager proactively establishes the necessary link groups before the instance begins serving. This ensures that any subsequent KV-cache movement can proceed without waiting for on-demand connectivity setup, allowing the newly activated instance to take traffic immediately. In contrast, prior systems such as Mooncake [130] construct these communication links only at the moment a prefilled request needs its KV-cache forwarded to a decode instance, incurring additional delay.

5.3 Experimental Setup

We evaluate HyperFlexis on different models, realistic hardware constraints, and representative multi-SLO workloads. Unless otherwise noted, all methods are tested under identical hardware, software, and workload configurations to ensure a fair comparison. Each experiment is repeated three times, with the mean shown as a point and the standard deviation represented as error bars in the plots.

Environment. We conducted all experiments on a cluster of four servers. Each server is equipped with eight Ascend NPUs [105], each providing 64 GB of memory. Additionally, each server contains four HiSilicon Kunpeng-920 CPUs [161], with 48 cores per CPU (192 cores in total), running in 64-bit ARMv8-A mode (aarch64). Our implementation is built upon PyTorch 2.1.0 [9] with the Ascend backend, Python 3.9.9, CANN 7.6 [104], and vLLM-Ascend [152].

Serving Models. We evaluate our system using three representative open-source LLMs: Qwen7B, Qwen32B [10], and LLaMA70B [149]. These models span a wide range of deployment scales, from lightweight setups to large-scale deployments. All experiments are conducted with FP16 precision, the standard configuration for inference serving as it bal-

Task	SLO (s)		Lengths	
	TTFT	TPOT	Input	Output
`medical_qa` [37]	0.7	0.5	32.57 ± 10.32	38.92 ± 16.83
`tldr_content_gen` [174]	1	0.7	44.38 ± 6.58	96.04 ± 35.03
`tldr_headline_gen` [174]	2	0.9	121.82 ± 35.04	13.59 ± 6.55
`wikisql` [174]	20	1	643.22 ± 337.01	27.82 ± 4.84
`gsm8k` [35]	0.7	0.2	51.44 ± 15.78	90.13 ± 26.73
`sharegpt`[175]	2	0.5	259.19 ± 324.88	207.79 ± 234.99

Table 5.1: **Summary statistics of the benchmark tasks.** The first four tasks form the 4-task multi-SLO set, and the last two form the 2-task set. Input/output lengths are reported as mean ± standard deviation over 300 requests.

ances efficiency and accuracy. We employ tensor parallelism (TP) [143] to distribute weights across devices, using TP=1 for Qwen7B, TP=2 for Qwen32B, and TP=8 for LLaMA70B.

Benchmark Workloads and Datasets. By default, we evaluate on multi-task scenarios with multiple SLO requirements. We use two distinct multi-SLO task sets. The first set contains four diverse tasks with heterogeneous SLO requirements, summarized in Table 5.1. This set includes: (1) the `medical_qa` dataset [37]; (2) the `tldr_content_gen` and `tldr_headline_gen` datasets [174]; and (3) the `wikisql` dataset [174, 176]. The second set comprises two tasks: `gsm8k` [35], and `sharegpt` [175], which capture high-complexity reasoning and conversational workloads. We draw 300 samples per task. Requests for each task follow a Poisson arrival distribution. A fixed random seed ensures reproducibility.

Baselines. We compare HyperFlexis to following baseline scheduling approaches.

(1) RR under the Llumnix framework [173]. The open-source Llumnix runtime adopts a default Round Robin (RR) dispatcher to distribute requests across instances, thereby enabling multi-instance serving—i.e., multiple model instances collectively handling incoming requests. This baseline provides simple load balancing but is completely SLO-agnostic.

(2) Simulated Annealing (SA) [79] is a stochastic single-instance scheduler designed for multi-SLO workloads. SA operates on an instance's local queue and uses simulated annealing to rearrange request ordering and batch formation, aiming to *maximize SLO attainment while minimizing average end-to-end latency*. However, the algorithm operates only on a single queue within a single instance. In our experiment, we extend it to multi-instance serving by adding an external load balancer (e.g., RR) to distribute requests across instances, after which each instance independently applies SA to its own local queue.

(3) SCORPIO [147] is another single-instance multi-SLO scheduler. Given an instance's current load, SCORPIO predicts whether admitting a request would cause either its own SLO or the SLOs of existing requests in the batch schedule to be violated. Requests that

Method	**Multi-instance**	**Multi-SLO**	**Scaling**	**Disagg.**
RR (in Llumnix)	✓	✗ (SLO-agnostic)	✗	✓
SA	✗ (if using RR, ✓)	✓	✗	✗
SCORPIO	✗ (if using RR, ✓)	✓	✗	✗
ALADDIN	✓	✗ (single-SLO)	✓	✗
HyperFlexis (ours)	✓	✓	✓	✓

Table 5.2: Comparison of baselines across multi-instance support, multi-SLO capability, scaling, and compatibility with disaggregated deployment mode (denoted by "disagg.").

cannot satisfy their SLOs under any feasible schedule are proactively rejected. Like SA, SCORPIO provides no native multi-instance coordination. Similarly, we extend it via an external RR dispatcher.

(4) ALADDIN [118] is a cluster-level scheduler that jointly handles request placement and resource scaling. It models scheduling as a two-dimensional bin-packing problem, where each bin is subject to two constraints: a memory capacity determined by the instance's memory budget, and a time capacity that enforces SLO-compliant latency. Because the time-side capacity must remain a fixed constant for the packing formulation to hold, Aladdin is limited to single-SLO scenarios in which all requests share the same SLO target. Aladdin adopts a heuristic bin-packing routine that packs as many requests as possible into each bin while satisfying both memory and latency constraints. Compared with SA and SCORPIO, Aladdin naturally supports multi-instance operation and cluster-level scaling decisions.

Our HyperFlexis provides full support for features summarized in Table 5.2. In particular, it elastically scales the number of active instances and is compatible with both disaggregated and co-located deployment modes, offering a broader applicability than prior baselines. We use HyperFlexis to denote our SLO-aware dispatching, and HyperFlexis-Scaling to denote SLO-aware dispatching with scaling.

Metrics. We evaluate performance using three metrics (Section 5.1.2): (1) **SLO attainment**, the fraction of requests meeting both TTFT and TPOT; (2) **End-to-end latency**, defined as the time from request arrival to completion, reported as the mean over all requests; (3) **Resource cost** (Equation 5.3), measured as cumulative active instance-time (one unit = one instance active for 50 ms), enabling fair comparison between methods with and without scaling.

5.4 Performance Evaluation

5.4.1 Performance under PD Colocated Mode

Figure 5.5 compares the performance of HyperFlexis against RR and SCORPIO. The top row reports **SLO attainment**. On the 2-task benchmark, SLO-aware dispatching enables

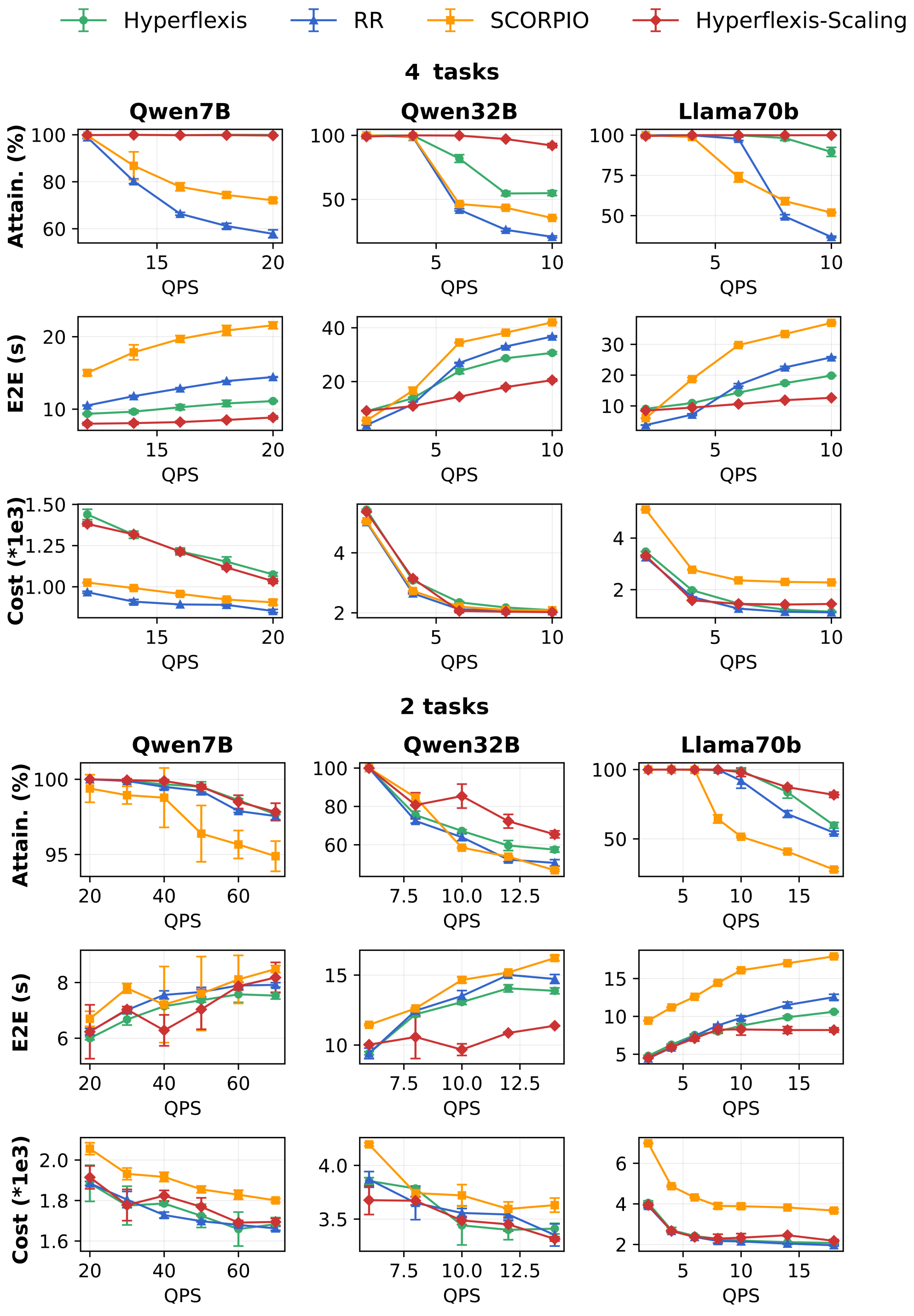


Figure 5.5: **Multi-task performance on 2-task and 4-task workloads.** Metrics include SLO attainment (top row, higher is better), end-to-end latency (middle row, lower is better), and cost (bottom row, lower is better) for HyperFlexis, RR, SCORPIO, and HyperFlexis-Scaling. By default we use two instances, with up to four for scaling.

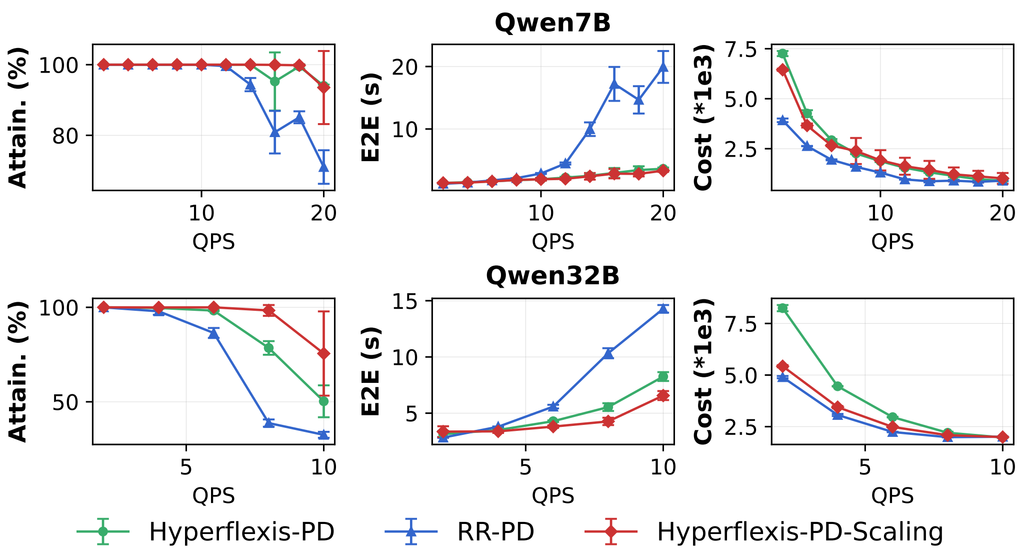


Figure 5.6: **Performance comparison under PD disaggregated mode.** Results for 4-task workloads using Qwen7B and Qwen32B.

HyperFlexis to improve attainment by up to 1.23× over RR and 2.14× over SCORPIO. With scaling enabled, HyperFlexis-Scaling further raises these gains to 1.5× and 2.93×, respectively. On the more heterogeneous 4-task benchmark, contention and queuing effects intensify, amplifying the benefits of SLO-aware scheduling. HyperFlexis achieves up to 2.60× improvement over RR and 1.73× over SCORPIO, and HyperFlexis-Scaling further increases these improvements to 4.44× and 2.59×.

The middle row reports **end-to-end latency**. While all methods show similar latency at low QPS, our advantages become clear as load increases, maintaining responsiveness under heavy traffic. On the 4-task benchmark, HyperFlexis-Scaling achieves up to 50.96% and 65.82% latency reduction over RR and SCORPIO, confirming that our SLO-aware policies effectively mitigate queuing and contention.

The bottom row compares **cost efficiency**. When compared to RR, our methods HyperFlexis and HyperFlexis-Scaling achieve similar cost, with occasional reductions of up to 4.99%. In contrast, relative to SCORPIO, our methods deliver substantial cost savings, indicating that the improved user satisfaction of our methods does not come at the expense of higher costs. Notably, for Llama70B, we observe up to 49.81% cost reduction over SCORPIO.

5.4.2 Performance under PD Disaggregated Mode

Figure 5.6 shows the comparison under the PD Disaggregated Mode for a 4-task workload. For HyperFlexis-PD-Scaling, we initially allocate 4 instances (2 Prefill and 2 Decode instances) and allow scaling up to 8 instances on demand. Note that the experiment for

	Qwen7B	Qwen32B	LLaMA70B
Fast Scaling	**0.89 ± 0.01**	**2.05 ± 0.02**	**1.16 ± 0.05**
CPU Offloading	2.73 ± 0.17	19.41 ± 3.14	11.50 ± 1.86
Disk loading	4.14 ± 0.26	28.84 ± 2.23	22.58 ± 0.82

Table 5.3: **Scaling time for different models.** Results for Qwen7B, Qwen32B, and LLaMA70B using different methods, measured in seconds.

Qwen32B (TP=2) runs across nodes, highlighting the system's scalability and flexibility. The RR-PD baseline means that request dispatching in both the Prefill and Decode stages follows a round-robin policy. As shown in Figure 5.6, both HyperFlexis-PD and HyperFlexis-PD-Scaling achieve higher attainment (up to 2.54×) and lower request latency (up to 31.82% reduction) compared to RR-PD. With auto-scaling enabled, the cost of HyperFlexis-PD-Scaling is only slightly higher than RR-PD for Qwen7B and roughly equivalent to RR-PD for Qwen32B.

5.4.3 Fast Scaling Evaluation

We evaluated three scaling strategies for HyperFlexis LLM serving: (1) loading weights from disk when scaling, (2) offloading weights to CPU when scaling out and reloading from CPU when scaling in, and (3) loading weights from an already running instance, which we term *Fast Scaling*. As shown in Table 5.3, Fast Scaling significantly reduces scaling time, achieving up to a 9.88× speedup over CPU offloading and 19.39× speedup over disk-based loading. This demonstrates that reusing weights resident in peer device memory can dramatically accelerate dynamic scaling without incurring the overhead of disk or CPU transfers.

5.4.4 Single-Task Performance Evaluation

HyperFlexis is designed for multi-task, multi-SLO workloads, we want to verify that it is still working on single-task deployments that remain common in production. We evaluate on the WikiSQL dataset using Qwen7B and Qwen32B, and compare HyperFlexis against two state-of-the-art single-task, single-SLO baselines, Aladdin and SA, under varying QPS levels. For this experiment, the TTFT and TPOT SLOs are set to 0.7 s and 0.5 s, respectively.

As shown in Figure 5.7, all systems maintain near-perfect SLO attainment under light load. As QPS increases, however, SA's attainment drops sharply once the system approaches saturation, while HyperFlexis sustains higher attainment, particularly at the knee point for Qwen7B (40 QPS).

Figure 5.7 also reports end-to-end latency. Under higher loads, however, HyperFlexis delivers more stable performance, avoiding the sharp latency spikes observed with SA and Aladdin (e.g., at 12 QPS on Qwen32B). These results confirm that HyperFlexis's schedul-

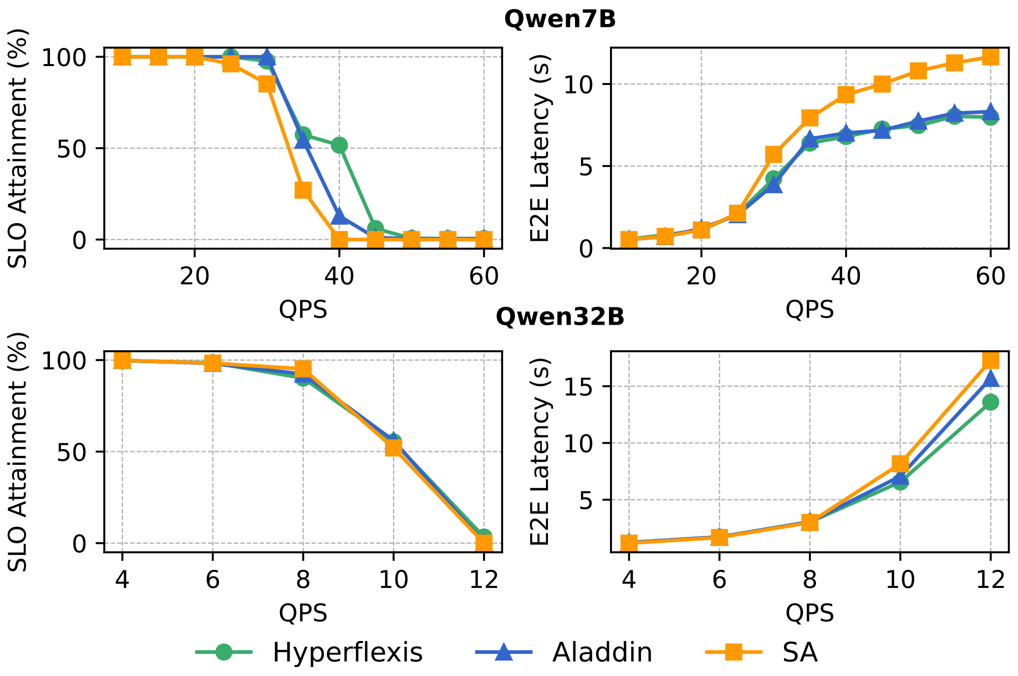


Figure 5.7: **Single-task performance comparison.** Results for Qwen7B and Qwen32B on the WikiSQL dataset compared with Aladdin [118] and SA [79]. The TTFT and TPOT SLOs are set to 0.7 s and 0.5 s, respectively.

ing and instance-aware queuing mechanisms improve responsiveness and reduce congestion even in single-task settings.

Overall, these results show that although HyperFlexis is designed to handle heterogeneous multi-SLO workloads, it remains competitive with, and in some cases outperforms, state-of-the-art methods Aladdin and SA in single-task settings.

5.4.5 Monitor and Scaling Interval

To understand the sensitivity of HyperFlexis's performance to key system parameters, we conduct an ablation study varying both the monitoring interval and the scaling interval as shown in Figure 5.8.

Effect of Monitor Interval. We vary the monitor interval across 50 ms, 1 s, and 5 s. As shown in the top row of Figure 5.8, performance is largely robust to the choice of interval. However, slight trends are observable: (1) SLO attainment is slightly reduced with a 5 s monitor interval compared to 50 ms and 1 s, reflecting delayed feedback that slows adaptation to system status changes; (2) end-to-end latency is marginally higher with the 50 ms interval due to the slight overhead of frequent monitoring, especially under heavy load, though the impact remains minimal. Overall, HyperFlexis is robust across a wide

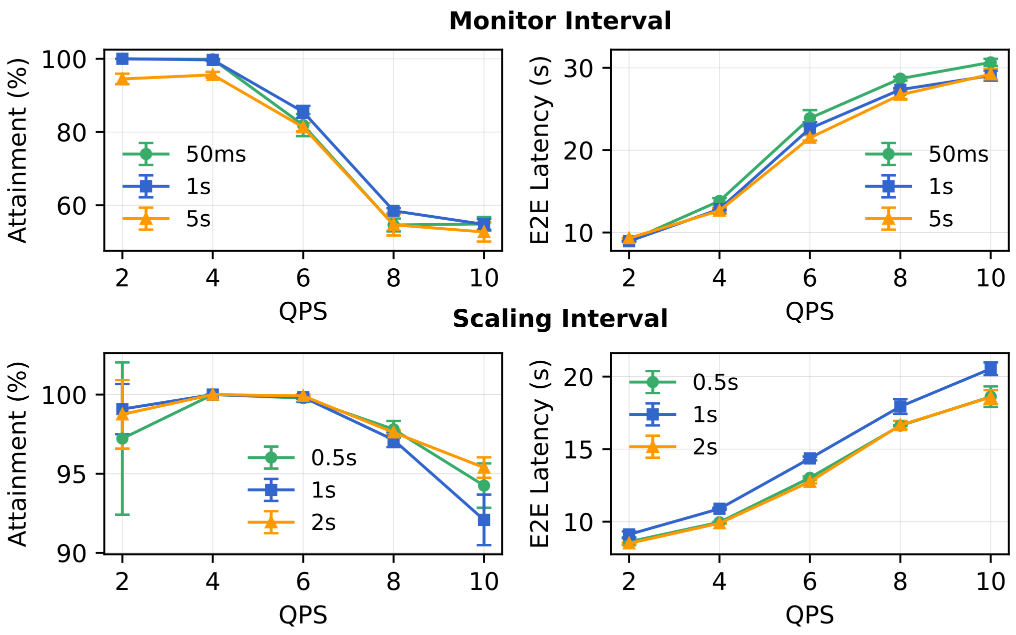


Figure 5.8: **Impact of monitor and scaling intervals.** Results for Qwen32B on the 4-task benchmark are shown with two instances (top) and up to four instances (bottom). Overall, HyperFlexis's performance is largely insensitive to changes in the scaling and monitor intervals.

range of monitor intervals, and choosing a moderate interval (e.g., 1 s) provides a good balance between responsiveness and overhead.

Effect of Scaling Interval. We vary the scaling interval across 0.5 s, 1 s, and 2 s to evaluate how the frequency of scaling decisions affects performance (Figure 5.8). SLO attainment shows minimal differences across intervals for most QPS values. At high QPS, shorter intervals exhibit slightly lower attainment, as the overhead of frequent scaling outweighs the benefits, particularly under heavy load when time is critical.

These results suggest that HyperFlexis's performance is largely insensitive to the scaling interval within the tested range. While very short or intermediate intervals may introduce minor variability in attainment or latency, overall system robustness is preserved.

5.5 Conclusion

This chapter presented HyperFlexis, which designs both scheduling algorithms and system mechanisms for multi-SLO LLM serving and fast scaling on Ascend NPUs.

On the algorithmic side, we developed an SLO-aware dispatching policy that admits and places requests according to TTFT/TPOT targets and instance states, and couples it with elasticity-aware decisions that trigger scaling when an emerging SLO risk is detected. These decisions are made in real time through an event-driven, maturity-time-based mechanism, enabling HyperFlexis to react promptly to workload changes rather than relying solely on coarse, periodic control.

On the system side, we introduced a Fast Scaling mechanism that reuses weights from running instances over device-to-device links, achieving rapid scale-out with minimal cold-start overhead compared to CPU- or disk-based loading. Meanwhile, HyperFlexis is compatible with existing deployment modes, supporting both colocated and prefill/decode-disaggregated execution modes.

Experiments on Qwen and LLaMA models, under diverse multi-task and single-task workloads, show that HyperFlexis improves SLO attainment, reduces end-to-end latency, and matches or lowers resource cost relative to round-robin, SCORPIO, Aladdin, and simulated-annealing-based schedulers.

A natural and optional extension is to relax the assumption that applications provide explicit TTFT and TPOT targets. In many real deployments, operators only specify coarse priority labels (e.g., "high" vs. "low" importance), while concrete latency budgets are either unknown or difficult to set. A straightforward extension is to translate these priority labels into actionable SLO targets so that HyperFlexis remains applicable. Such a priority-to-SLO layer could monitor recent performance histories—capturing both system cost and user satisfaction—and recommend suitable TTFT/TPOT targets. It could tighten SLOs for low-priority traffic when the system is underutilized, and gradually relax them as high-priority load increases. This would reduce configuration effort for operators and broaden the applicability of HyperFlexis to environments where only priority information, rather than explicit latency contracts, is available.

Overall, this chapter illustrates how OR ideas—SLO-aware scheduling, capacity planning, and cost–performance trade-off modeling—can enhance modern ML systems at serving time. It complements earlier chapters that use ML to assist optimization modeling and solving, representing the opposite direction of interaction between ML and OR. In the next chapter, we synthesize insights across all contributions of this thesis and discuss broader opportunities for deepening the synergies between OR and ML.

Chapter 6

Conclusion and Future Directions

Operations Research (OR) and Machine Learning (ML) are two powerful tools widely employed for solving real-world problems. While OR provides rigorous mathematical formulations, optimization principles, and decision-theoretic frameworks, ML offers adaptivity, scalability, and the capacity to learn complex patterns from large datasets. This thesis explores several synergies between OR and ML, enhancing methods and tools for tackling a variety of practical challenges. The key takeaway from this work is that ML can significantly enhance various steps in the OR process, while OR can guide the application and optimization of ML techniques, creating solutions that are more efficient, automated, fair, and scalable.

In this chapter, we summarize the key contributions of this thesis, which span representative cases of synergies between OR and ML, including: (1) ML enhancing solution methods, (2) ML enhancing mathematical program formulation, and (3) OR enhancing ML model serving. We then outline several directions for future research. These include extending the above forms of synergy, exploring new interactions across other stages of the OR process and the ML lifecycle, and investigating integrated OR–ML approaches in which learning and optimization are jointly designed as a unified decision-making framework.

6.1 Summary of the Thesis

This thesis investigates three cases where OR and ML mutually enhance each other.

- **Smart Initial Basis Selection.** We introduced a learning-based approach to accelerate the solution process of linear programming by predicting efficient initial bases using graph neural networks. Classical solvers often rely on heuristic or default strategies for basis initialization, which can result in suboptimal performance. By learning from the graph structure of correlated linear programs, the proposed model achieved high prediction accuracy (around 81%) and led to significant speedups across different problem instances. The method demonstrates how machine learning can directly enhance the key algorithmic component in the LP solving method.

- **Evaluation Metric for NL2LP task.** LLMs have emerged as powerful tools for assisting or even automating mathematical programming. However, existing evaluation metrics fail to capture expert judgment and the structural correctness of generated models—that is, whether the formulation faithfully respects the ground-truth relationships among objectives, variables, and constraints. To address this gap, we proposed a graph-based edit distance metric tailored to linear programming formulations. The metric evaluates both semantic and structural fidelity while remaining permutation-invariant to the ordering of constraints and variables, thus aligning more closely with human expert evaluations. It enhances the reliability and interpretability of benchmarking in NL2LP tasks.
- **System and Scheduling Algorithm Designs for LLM Serving:** This chapter develops both the serving system components and the scheduling algorithms for LLM requests with heterogeneous service-level objectives (SLOs). The system supports heterogeneous SLOs and fast scale-out with minimal cold-start time through modular execution components and runtime interfaces that allow the scheduler to control batching, dispatching, and resource sharing in real time. On top of this system, we developed heuristic and greedy scheduling strategies to balance latency and system cost across workloads. Experiments showed substantial gains in efficiency and service quality compared with standard baselines. This work exemplifies an optimization-oriented design philosophy inspired by operations research, treating large-scale LLM serving as a dynamic decision-making problem for scalable and efficient deployment.

Collectively, these chapters demonstrate how OR and ML can be mutually beneficial, unlocking new capabilities for automated, efficient, and scalable real-world problem-solving.

6.2 Impact of the Emergence of LLMs

The rapid rise of LLMs has reshaped the landscape of many existing research directions. This section discusses how these developments influence the conclusions of this thesis and whether the core insights remain valid. Overall, the main conclusion of this thesis remains unchanged. Since LLMs are a subfield within machine learning, the central argument—that ML can enhance OR processes, and OR can in turn improve the ML lifecycle—continues to hold. In fact, the emergence of LLMs reinforces and extends this synergy, creating new opportunities for integration between OR and ML across all stages of model design, training, and deployment. In the following, we discuss the impact of LLMs' emergence on the conclusions of each chapter.

LLMs and Smart Initial Basis Selection In Chapter 3, we introduced a GNN-based strategy for predicting the initial basis in linear programming. Despite the rise of LLMs, our GNN strategy remains highly effective. LLMs excel at processing text and code but are not inherently suited for numerical optimization or reasoning over graph-structured data. Therefore, the graph representation and GNN-based approach—which naturally captures the structure of LP problems—continue to play a critical role in solver acceleration and combinatorial optimization [55, 129].

Nevertheless, LLMs open new possibilities for hybrid approaches. Specifically, LLMs can provide semantic understanding or meta-level guidance to complement GNN reasoning. For example, in a supply chain planning problem, an LLM could interpret that a recent surge in apple demand implies production should reach resource limits for apple-related constraints. Incorporating such contextual knowledge into the graph representation could help the GNN predict a closer-to-optimal initial basis, especially in low-data or few-shot training settings, leading to faster convergence and improved solution quality [81].

LLMs and Human-Aligned LP Evaluation The work in Chapter 4 was conducted after the emergence of large language models (LLMs), which have demonstrated strong capabilities in reasoning and generating mathematical programs from natural language [107, 153]. Motivated by the lack of human-aligned evaluation metrics for this generation task, we proposed a new metric that better reflects user intent and logical correctness [122]. This metric has the potential to guide future natural language–to–math programming models to produce formulations that more faithfully capture user intent, and it could also serve as a reward signal in reinforcement learning processes.

Moreover, the metric itself can be further enhanced by leveraging LLMs as evaluators, since they are well known for their semantic understanding capabilities [98]. As an extension of our proposed metric, when comparing generated programs with ground truth, constraints and variables can be annotated with natural language explanations, and an LLM (or a simpler ML model) can act as a scorer to assess semantic similarity between the two explanations.

An open challenge remains in defining semantic equivalence [47]. A single problem may have multiple valid formulations—for example, one student may solve a cutting-stock problem using mixed-integer programming, column generation, or constraint programming formulations. All are correct yet differ structurally. In such cases, our metric may fail to recognize semantic equivalence. Future work could explore LLM-based evaluators and address this issue by reasoning about alternative yet valid formulations.

Overall, LLMs remain promising for enhancing or even automating mathematical programming tasks, and a reliable evaluation metric remains essential for assessing LLM performance on these specific tasks. Moreover, LLMs also open new opportunities for developing better evaluation metrics [107, 98].

LLMs and Serving System The work of Chapter 5 was conducted in the era of LLMs; therefore, rather than asking what impact LLMs might bring, our focus is on how OR techniques remain—and will continue to be—essential for improving LLM serving efficiency. With the rapid growth of LLM applications, challenges such as high inference latency, GPU scheduling, and energy-efficient resource allocation have become increasingly critical. OR, with its mature foundations in scheduling and multi-objective optimization, provides systematic and provably efficient approaches to address these issues [27]. Recent advances further show that combining reinforcement learning and optimization can enable adaptive GPU resource allocation for distributed LLM inference [48]. Complementarily, large-scale scheduling studies in GPU datacenters [164] emphasize the ongoing importance of algorithmic design for efficient serving. Looking forward, the intersection of OR and LLM serving opens promising research directions, where classical optimization principles (e.g., mixed-integer programming, queueing models) can guide scalable, adaptive, and multi-SLO serving systems for next-generation LLM infrastructures.

6.3 Future Directions

There remains substantial potential for advancing the synergy between OR and ML. Building on the themes explored in this thesis, future research directions can be naturally organized into three categories: (i) ML for OR; (ii) OR for ML; and (iii) integrated OR–ML approaches, where optimization and learning are tightly coupled into unified decision-making pipelines.

6.3.1 ML for OR

Neural Combinatorial Optimization Neural combinatorial optimization refers to using neural networks to help solve combinatorial optimization problems, such as routing, scheduling, and resource allocation. Traditional methods like branch-and-bound or dynamic programming often become expensive as the problem size grows. Neural approaches aim to learn heuristic or approximate strategies that can produce good solutions quickly, often by training on many example problems. This direction falls under the type of synergy where ML enhances traditional solution methods.

GNNs have been widely explored for this task. The message-passing mechanism of GNNs naturally captures local dependencies among decision variables. However, this raises a research question: Are local interactions sufficient for solving complex combinatorial optimization problems? As a potential future direction, architectures incorporating global attention—such as graph transformers—offer the ability to model long-range dependencies and global constraints, which are common in complex optimization problems like facility location and production scheduling.

Beyond architectural advances, the emergence of LLMs provides a broader insight: simple architectures, when over-parameterized and trained on massive, diverse datasets, can exhibit emergent reasoning capabilities. Translating this insight to the context of GNNs suggests a promising direction—scaling up lightweight or shallow GNNs on large, heterogeneous optimization instances to develop general-purpose "foundation models" for combinatorial optimization.

Meanwhile, scalability and efficiency remain critical challenges in large-scale real-world applications. Techniques such as graph sampling, subgraph clustering, and hierarchical training can effectively reduce computational overhead while maintaining predictive quality. Integrating these efficiency strategies with large-scale training may enable neural combinatorial optimization to become both powerful and practical for industrial-scale problems.

LLM-Assisted Programming Recent literature has explored the potential of LLMs and LLM-based agents in automating mathematical programming, particularly in translating natural language problem descriptions into formal optimization models. These systems often incorporate auxiliary information from solver outputs—such as tentative solutions or error messages—as well as human feedback, to iteratively refine the generated formulations. For example, OR-LLM-Agent [85] runs GPT-generated code in a sandboxed environment: if Python errors or infeasibility are detected, the system prompts the LLM to self-repair the code and re-execute it. If the solver still fails to find a feasible solution, a "self-verification" step asks the LLM to re-examine the mathematical model and regenerate the formulation.

However, current approaches primarily focus on semantic-level edits—modifying the code or formulation structure based on the problem description and solver diagnostics. Beyond this design-oriented refinement, richer forms of feedback could be considered. For example, higher-level semantic signals such as user preferences or domain-specific knowledge remain underexplored. Moreover, incorporating numerical feedback presents a particularly promising extension, for instance, by integrating feasibility relaxation into the LLM-based modeling loop. Feasibility relaxation is a well-established concept in traditional operations research, where solvers address infeasible problems by identifying a minimally relaxed version—allowing selected constraints to be violated at minimal cost—to restore feasibility. This mechanism is especially important in practical applications involving soft constraints, where certain requirements can be flexibly adjusted rather than strictly enforced. It also contributes to interpretability and user trust, as it provides transparent explanations of infeasibility and quantifies the degree of violation required to regain feasibility.

Combining these aspects, future LLM-assisted modeling systems could better handle feasibility issues in automatically generated models by balancing constraint relaxation, domain knowledge, and user preferences. Such capabilities would bring us closer to end-to-end assistants capable of formulating solvable and realistic optimization problems in complex, user-defined contexts.

New OR–ML Synergies Beyond Canonical Pipelines The OR process and the ML lifecycle introduced in Chapter 2 are conceptually simplified abstractions. In practice, real-world workflows are increasingly iterative, adaptive, and non-sequential.

This observation naturally raises the question of whether ML can be used to enhance additional substeps of the OR process beyond those traditionally considered. In our survey [54], we conclude that ML techniques can also be applied to other stages, such as model parameter generation, solution explanation, and the interactions between OR steps.

More broadly, these developments motivate the exploration of improved and more flexible pipelines. For example, with the integration of LLMs, the OR process may become increasingly dynamic: users can iteratively modify constraints, adapt formulations to new scenarios, or debug infeasibilities through natural language interaction. Such interactions enable on-demand transitions among the stages of modeling, solving, and evaluation, opening the door to new forms of synergy between OR and ML that go beyond canonical, linear pipelines.

6.3.2 OR for ML

OR Enhancing ML Model Deployment There has been work [94, 182, 123, 171] from systems and OR perspectives to improve ML model deployment, focusing on aspects such as model serving efficiency and model selection under accuracy–efficiency trade-offs. However, deployment goes beyond serving optimization—it also encompasses failure recovery, rollback mechanisms, model versioning, and A/B testing to ensure reliability and adaptability in dynamic environments. A promising yet underexplored research direction is applying OR methods to monitoring and maintenance in the deployment stage. For example,

1. *Data drift detection as optimization:* Formulate continuous monitoring as a resource allocation or change-detection optimization problem, deciding when and where to sample data for drift tests under limited monitoring budgets.

2. *Model versioning and A/B test optimization:* Use multi-armed bandit or online optimization techniques to dynamically route traffic among multiple model versions, balancing exploration (testing new models) and exploitation (serving stable production models). It is worth exploring adaptive OR heuristics to determine optimal user-group assignments for statistically reliable inference with minimal experimental overhead and potential degradation of user experience.

3. *Robust rollback and failure recovery:* Design OR-based decision frameworks for optimal rollback timing under uncertainty, minimizing downtime and service disruption.

These directions illustrate how OR, beyond improving efficiency, can provide optimization-based principles to enhance the reliability, adaptability, and continual improvement of deployed ML systems.

New OR–ML Synergies Beyond Canonical Pipelines The ML lifecycle introduced in Chapter 2 are conceptually simplified, whereas a real-world lifecycle may involve a broader range of stages. Examples include:

1. *Model selection*: identifying a model that best balances accuracy and efficiency trade-offs across a large design space [66];
2. *Model unlearning*: removing the influence of specific data points to comply with privacy regulations [68, 18];
3. *Interpretability refinement*: improving transparency and alignment with human reasoning [46];
4. *Contribution valuation*: quantifying each participant's data value in multi-party settings (e.g., federated learning) for fairness and incentivization [52].

6.3.3 Integrated OR–ML

Beyond the complementary paradigms of ML for OR and OR for ML, an intriguing direction is the development of integrated OR–ML methodologies, where learning and optimization are jointly designed as a single, end-to-end pipeline. This end-to-end perspective closely reflects real-world decision-making processes. There already exist several interesting works along this direction.

Predict then optimize It is the simplest form of integrating ML and OR: it concatenates a learning module with an optimization module. In this setting, the two components remain conceptually separated: an ML model first predicts uncertain or unknown parameters in an optimization problem, and these predictions are then treated as fixed inputs to an OR solver to compute the final decision. This paradigm is widely used in practice. For instance, in supply chain and logistics, demand forecasts learned from historical sales and contextual signals are fed into inventory, replenishment, or distribution optimization models to balance service levels and cost [13]. In transportation and mobility systems, ML models can estimate traffic conditions or travel times from spatiotemporal data, after which routing and dispatch decisions are computed via classical optimization, e.g., for dynamic vehicle routing [11]. Overall, predict-then-optimize provides a simple and practical baseline for more tightly integrated OR–ML methodologies.

A natural extension of this paradigm aims to make learning and optimization more tightly integrated. The idea is to recognize that prediction errors in the estimated parameters are inevitable. Crucially, not all prediction errors are equally important, for example, some parameters may be difficult to predict accurately, yet their errors may have little impact on the final decision. From a decision-making perspective, the ultimate objective is therefore not to minimize prediction error, but to minimize the *decision error* induced by these predictions. To achieve this, the optimization problem is placed inside the learning loop, allowing gradients of a decision-aware loss to propagate through the optimization solver back to the prediction model. This can be achieved via implicit differentiation or sensitivity analysis of the optimization problem. For instance, OptNet can embed a quadratic program as a neural network layer and backpropagates gradients through the corresponding KKT conditions [6]. In parallel, the Smart Predict–then–Optimize (SPO) framework proposes loss functions that directly measure decision error and derives tractable surrogate losses that can be optimized using standard learning pipelines [50].

Learning constraints from data In many applications, the main bottleneck is not only unknown parameters but also unknown or partially specified constraints. A representative approach is OptiCL [111] (Mixed-Integer Optimization with Constraint Learning), which learns unknown constraints and/or objectives from data as predictive models (e.g., linear models, trees, or ReLU networks) and then embeds these learned models into a mixed-integer formulation. Meanwhile, Fajemisin et. al. [51] frames constraint learning as a feasibility modeling problem: unknown constraints are learned by training a feasibility classifier or a surrogate constraint function from feasible and infeasible examples, and then iteratively refined through a solve–verify–update loop in a data-driven constraint modeling framework. On the tooling side, OMLT offers an open-source interface for incorporating trained surrogate models into large-scale optimization problems using algebraic modeling languages such as Pyomo [25].

Differentiable Optimization Layers A recent and influential line of research [3, 6] embeds *parameterized* optimization problems directly as layers within neural network architectures, thereby enabling the network to perform *constrained* decision-making rather than only unconstrained function approximation. As a toy example, if we train a neural network to solve mini-Sudoku puzzles (4×4) using only input–output pairs, it cannot guarantee that the solutions obey the Sudoku rules. By contrast, with differentiable optimization layers, these rules can be explicitly encoded into the output of a network layer. In this paradigm, the output of the layer is defined implicitly as the optimal solution to an optimization problem, with parameters that may depend on upstream network representations; gradients are computed via sensitivity analysis or implicit differentiation of the optimality conditions, allowing the entire architecture to be trained end-to-end using gradient descent. Embedding optimization layers is especially valuable in applications where feasibility, constraints, or structured outputs are intrinsic to the task—for example, in control and robotics, where actions must respect physical and safety constraints; in resource allocation and signal processing, where optimal trade-offs matter; and in structured prediction problems, where outputs must satisfy combinatorial or convex feasibility conditions.

Bibliography

[1] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. An exact graph edit distance algorithm for solving pattern recognition problems. In *Proceedings of the International Conference on Pattern Recognition Applications and Methods*, pages 271–278. SciTePress, 2015.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and Z. Kolter. Differentiable convex optimization layers. In *Advances in Neural Information Processing Systems*, 2019.

[4] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with Sarathi-serve. *Proceedings of 18th USENIX Symposium on Operating Systems Design and Implementation, 2024, Santa Clara*, 2024.

[5] Sohaib Ahmad, Hui Guan, Brian D. Friedman, Thomas Williams, Ramesh K. Sitaraman, and Thomas Woo. Proteus: A high-throughput inference-serving system with accuracy scaling. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, page 318–334, 2024.

[6] Brandon Amos and J. Zico Kolter. OptNet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning (ICML)*, pages 136–145, 2017.

[7] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, 2016.

[8] David Applegate, Mateo Díaz, Oliver Hinder, Haihao Lu, Miles Lubin, Brendan O'Donoghue, and Warren Schudy. Practical large-scale linear programming using primal-dual hybrid gradient. In *Proceedings of the 34th Conference on Neural Information Processing Systems*, pages 20243–20257, 2021.

[9] Ascend Community. Ascend pytorch adapter (`torch_npu`). `https://github.com/Ascend/pytorch`.

[10] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[11] Léo Baty, Kai Jungel, Patrick S. Klein, Axel Parmentier, and Maximilian Schiffer. Combinatorial optimization-enriched machine learning to solve the dynamic vehicle routing problem with time windows. *Transportation Science*, 58(4):708–725, 2024.

[12] Amir Beck. *First-Order Methods in Optimization*. Society for Industrial and Applied Mathematics (SIAM), 2017.

[13] Dimitris Bertsimas and Nathan Kallus. From predictive to prescriptive analytics. *Management Science*, 66(3):1025–1044, 2020.

[14] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*, volume 6. Athena Scientific, 1997.

[15] Robert E Bixby. Implementing the simplex method: The initial basis. *INFORMS Journal on Computing*, 4(3):267–284, 1992.

[16] Robert E Bixby and Matthew J Saltzman. Recovering an optimal LP basis from an interior point solution. *Operations Research Letters*, 15(4):169–178, 1994.

[17] R. Bödi and K. Herr. Symmetries in integer programs. *arXiv preprint arXiv:0908.3331*, 2009.

[18] Léo Bourtoule, Varun Chandrasekaran, Christopher A. Choquette-Choo, Haonan Jia, Alex Travers, Benjamin Zhang, David Lie, and Nicolas Papernot. Machine unlearning. In *IEEE Symposium on Security and Privacy (SP)*, pages 141–159, 2021.

[19] Simon Bowly, Kate Smith-Miles, Davaatseren Baatar, and Hans Mittelmann. Generation techniques for linear programming instances with controllable properties. *Mathematical Programming Computation*, 12(3):389–415, 2020.

[20] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

[21] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[22] Christopher Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 89–96, 2005.

[23] Chen Cai and Yusu Wang. A note on over-smoothing for graph neural networks. *arXiv:2006.13318*, 2020.

[24] Jordi Castro and Paula de la Lama-Zubirán. A new interior-point approach for large separable convex quadratic two-stage stochastic problems. *Optimization Methods and Software*, pages 1–29, 2020.

[25] Francesco Ceccon, Jeroen Jalving, John Haddad, Alexander Thebelt, Calvin Tsay, and Ruth Misener. OMLT: Optimization and machine learning toolkit. *Journal of Machine Learning Research*, 23(349):1–8, 2022.

[26] İbrahim Oğuz Cetinkaya, İ. Esra Büyüktahtakın, Parshin Shojaee, and Chandan K. Reddy. Discovering heuristics with large language models (llms) for mixed-integer programs: Single-machine scheduling. Technical report, Optimization Online, 2025.

[27] Robert Chab, Fei Li, and Sanjeev Setia. Algorithmic techniques for gpu scheduling: A comprehensive survey. *Algorithms*, 18(7):385, 2025.

[28] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[29] Siyuan Chen, Zhipeng Jia, Samira Khan, Arvind Krishnamurthy, and Phillip B Gibbons. SLOs-serve: Optimized serving of multi-SLO LLMs. *arXiv preprint arXiv:2504.08784*, 2025.

[30] Xiaohan Chen, Jialin Liu, and Wotao Yin. Learning to optimize: A tutorial for continuous and mixed-integer optimization. *arXiv:2405.15251*, 2024.

[31] Ziang Chen, Jialin Liu, Xinshang Wang, Jianfeng Lu, and Wotao Yin. On representing mixed-integer linear programs by graph neural networks. In *International Conference on Learning Representations (ICLR)*, 2023.

[32] Ke Cheng, Zhi Wang, Wen Hu, Tiannuo Yang, Jianguo Li, and Sheng Zhang. SCOOT: SLO-oriented performance tuning for LLM inference engines. In *Proceedings of the ACM Web Conference (TheWebConf 2025)*, 2025.

[33] Seungbeom Choi, Jeonghoe Goo, Eunjoo Jeon, Mingyu Yang, and Minsung Jang. Elis: Efficient LLM iterative scheduling system with response length predictor. *arXiv preprint arXiv:2505.09142*, 2025.

[34] Vasek Chvatal. *Linear programming.* W.H. Freeman, 1983.

[35] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[36] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[37] CMeKG_tools Contributors. Cmekg_tools: Chinese medical knowledge graph tools. `https://github.com/king-yyf/CMeKG_tools`, 2023.

[38] IBM ILOG Cplex. V12. 1: User's manual for CPLEX. *International Business Machines Corporation*, 46(53):157, 2009.

[39] Weihao Cui, Yukang Chen, Han Zhao, Ziyi Xu, Quan Chen, Xusheng Chen, Yangjie Zhou, Shixuan Sun, and Minyi Guo. Optimizing slo-oriented llm serving with pd-multiplexing. *arXiv preprint arXiv:2504.14489*, 2025.

[40] George B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1963.

[41] George B. Dantzig and Mukund N. Thapa. *Linear Programming 2: Theory and Extensions.* Springer-Verlag, 2003.

[42] George B Dantzig and Philip Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, 1960.

[43] Jonas De Bock, Kristof Coussement, Arnaud de Caigny, Roman Słowiński, Bart Baesens, Rob N. Boute, Tsan-Ming Choi, Dursun Delen, Martin Kraus, Stefan Lessmann, Sebastian Maldonado, David Martens, María Óskarsdóttir, Carlos Vairetti, Wouter Verbeke, and Richard Weber. Explainable AI for operational research: A defining framework, methods, applications, and a research agenda. *European Journal of Operational Research*, 317(2):249–272, 2024.

[44] Daswin De Silva and Damminda Alahakoon. An artificial intelligence life cycle: From conception to production. *Patterns*, 3(6):100489, 2022.

[45] Jian-Ya Ding, Chao Zhang, Lei Shen, Shengyin Li, Bing Wang, Yinghui Xu, and Le Song. Accelerating primal solution findings for mixed integer programs based on solution prediction. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 34, pages 1452–1459, 2020.

[46] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[47] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, Yan Liu, Yuhao Zhou, Yueming Wu, Rui Zheng, Ming Wen, Jingang Wang, Xunliang Cai, Xipeng Qiu, Qi Zhang, and Xuanjing Huang. Program semantic inequivalence game with large language models. *arXiv preprint arXiv:2505.03818*, 2025.

[48] Chengze Du, Zhiwei Yu, Heng Xu, Haojie Wang, Bo Liu, and Jialong Li. Temporal-aware gpu resource allocation for distributed llm inference via reinforcement learning. *arXiv preprint arXiv:2507.10259*, 2025.

[49] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[50] Adam N. Elmachtoub and Paul Grigas. Smart "predict, then optimize". *Management Science*, 68(1):9–26, 2022.

[51] Adejuyigbe O. Fajemisin, Donato Maragno, and Dick den Hertog. Optimization with constraint learning: A framework and survey. *European Journal of Operational Research*, 314(1):1–14, 2024.

[52] Zhenan Fan, Haifeng Fang, Xinglu Wang, Zirui Zhou, Jie Pei, Michael Friedlander, and Yong Zhang. Fair and efficient contribution valuation for vertical federated learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

[53] Zhenan Fan, Bissan Ghaddar, Xinglu Wang, Linzi Xing, Yong Zhang, and Zirui Zhou. Artificial intelligence for operations research: Revolutionizing the operations research process. *arXiv preprint*, 2024.

[54] Zhenan Fan, Bissan Ghaddar, Xinglu Wang, Linzi Xing, Yong Zhang, and Zirui Zhou. Artificial intelligence for optimization: Unleashing the potential of parameter generation, model formulation, and solution methods. *European Journal of Operational Research*, 2025.

[55] Zhenan Fan, Xinglu Wang, Oleg Yakovenko, Aravind A. Sivas, Olle Ren, Yong Zhang, and Zirui Zhou. Smart initial basis selection for linear programs. In *Proceedings of the International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pages 9650–9664, 2023.

[56] Zhenan Fan, Zirui Zhou, Jian Pei, Michael P Friedlander, Jiajie Hu, Chengliang Li, and Yong Zhang. Knowledge-injected federated learning. *arXiv preprint arXiv:2208.07530*, 2022.

[57] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[58] John J Forrest and Donald Goldfarb. Steepest-edge simplex algorithms for linear programming. *Mathematical Programming*, 57(1):341–374, 1992.

[59] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, pages 1050–1059, 2016.

[60] I. L. Galabova and J. A. J. Hall. The 'Idiot' crash quadratic penalty algorithm for linear programming and its application to linearizations of quadratic assignment problems. *Optimization Methods and Software*, 35(3):488–501, 2020.

[61] Shihong Gao, Xin Zhang, Yanyan Shen, and Lei Chen. Apt-serve: Adaptive request scheduling on hybrid cache for scalable LLM inference serving. *Proceedings of the ACM on Management of Data*, 3(3), June 2025.

[62] X. Gao, B. Xiao, D. Tao, et al. A survey of graph edit distance. *Pattern Analysis and Applications*, 13:113–129, 2010.

[63] Saul I. Gass and Sasirekha Vinjamuri. Cycling in linear programming problems. *Computers & Operations Research*, 31(2):303–311, 2004.

[64] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[65] Jared Lee Gearhart, Kristin Lynn Adair, Justin David Durfee, Katherine A Jones, Nathaniel Martin, and Richard Joseph Detry. Comparison of open-source linear programming solvers. Technical report, Sandia National Lab., 2013.

[66] Mohsen Gholami, Mohammad Akbari, Xinglu Wang, Behnam Kamranian, and Yong Zhang. ETran: Energy-based transferability estimation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 18613–18622, 2023.

[67] Paul C Gilmore and Ralph E Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859, 1961.

[68] Antonio Ginart, Melody Guan, Gregory Valiant, and James Zou. Making ai forget you: Data deletion in machine learning. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[69] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[70] Igor Griva, Stephen G. Nash, and Ariela Sofer. *Linear and Nonlinear Optimization*, volume 108. Society for Industrial and Applied Mathematics (SIAM), 2009.

[71] Jianfeng Gu, Puxuan Wang, Isaac Nunezand, Kai Huang, and Michael Gerndt. Hasgpu: Efficient hybrid auto-scaling with fine-grained gpu allocation for slo-aware serverless inferences. *arXiv preprint arXiv:2505.01968*, 2025.

[72] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 1321–1330, 2017.

[73] Prateek Gupta, Maxime Gasse, Elias B. Khalil, Pawan Mudigonda, Andrea Lodi, and Yoshua Bengio. Hybrid models for learning to branch. In *Advances in Neural Information Processing Systems*, volume 33, pages 18087–18097, 2020.

[74] Prateek Gupta, Elias B Khalil, Didier Chetélat, Maxime Gasse, Yoshua Bengio, Andrea Lodi, and M Pawan Kumar. Lookback for learning to branch. *arXiv:2206.14987*, 2022.

[75] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[76] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[77] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, 2017.

[78] Conor F. Hayes, Felipe Leno da Silva, Jiachen Yang, T. Nathan Mundhenk, Chak Shing Lee, Jacob F. Pettit, Claudio Santiago, Sookyung Kim, Joanne T. Kim, Ignacio Aravena Solis, Ruben Glatt, Andre R. Goncalves, Alexander Ladd, Ahmet Can Solak, Thomas Desautels, Daniel Faissol, Brenden G. Petersen, and Mikel Landajuela. Deep symbolic optimization: Reinforcement learning for symbolic mathematics. *arXiv preprint arXiv:2505.10762*, 2025.

[79] Jinqi Huang, Yi Xiong, Xuebing Yu, Wenjie Huang, Entong Li, Li Zeng, and Xin Chen. Slo-aware scheduling for large language model inferences. *arXiv preprint arXiv:2504.14966*, 2025.

[80] Mengyu Huang, Yuxing Zhong, Huiwen Yang, Jiazheng Wang, Fan Zhang, Bo Bai, and Ling Shi. Simplex initialization: A survey of techniques and trends. *arXiv preprint arXiv:2111.03376*, 2021.

[81] Sen Huang, Kaixiang Yang, Sheng Qi, and Rui Wang. When large language model meets optimization. *arXiv preprint arXiv:2405.10098*, 2024.

[82] I. Huangfu and J. A. Julian Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018.

[83] Huawei Technologies Co., Ltd. Optverse solver, 2021.

[84] Huawei Technologies Co., Ltd. Ascend community hardware documentation. `https://www.hiascend.com/en/document?tag=hardware`, 2025.

[85] Jiali Jiang, Yuzhou Wang, Ji He, Jiayi Yan, Yujie Li, Yining Li, and Hongyuan Zha. OR-LLM-Agent: Chain-of-experts reasoning for solving operations research problems via large language models. *arXiv preprint arXiv:2401.02595*, 2025.

[86] Han Jung, Jinho Kim, and Yong-Hyuk Park. Learning to tune admm for distributed optimal power flow. *Electric Power Systems Research*, 212:108364, 2022.

[87] Hélcio Vieira Junior and Marcos Pereira Estellita Lins. An improved initial basis for the simplex algorithm. *Computers and Operations Research*, 32(8):1983–1993, 2005.

[88] Bahman Karimi, Seyed Mohammad Taghi Fatemi Ghomi, and J. M. Wilson. The capacitated lot sizing problem: a review of models and algorithms. *The International Journal of Management Science*, 31(5):365–378, 2003.

[89] Elias B. Khalil, Pierre Le Bodic, Le Song, George L. Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *AAAI Conference on Artificial Intelligence*, 2016.

[90] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[91] Victor Klee and George J Minty. How good is the simplex algorithm. *Inequalities*, 3(3):159–175, 1972.

[92] Klaus Krippendorff. *Content Analysis: An Introduction to Its Methodology.* SAGE Publications, Thousand Oaks, CA, 3 edition, 2013.

[93] Arun Kumar and M. Tamer Özsu. Data management in machine learning: Challenges, techniques. *arXiv preprint arXiv:1706.05869*, 2017.

[94] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[95] Walid Labassi, Elias B. Khalil, Quentin Cappart, Didier Chételat, Andrea Lodi, and Louis-Martin Rousseau. Learning to search in mixed-integer programming. *INFORMS Journal on Computing*, 34(2):1038–1053, 2022.

[96] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 6402–6413, 2017.

[97] Ailsa H. Land and Alison G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[98] Thanh Le-Cong, Bach Le, and Toby Murray. Can llms reason about program semantics? a comprehensive evaluation of llms on formal specification inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL 2025)*, 2025.

[99] Anqi Li, Bonan Li, Congying Han, and Tiande Guo. Rethinking optimal pivoting paths of simplex method. *arXiv preprint arXiv:2210.02945*, 2022.

[100] Haoyang Li, Yuan Wang, Shuo Zhang, et al. Out-of-distribution generalization on graphs: A survey. *arXiv:2202.07987*, 2022.

[101] Ke Li and Jitendra Malik. Learning to optimize. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[102] Qian Li, Tian Ding, Linxin Yang, Minghui Ouyang, Qingjiang Shi, and Ruoyu Sun. On the power of small-size graph neural networks for linear programming. In *Advances in Neural Information Processing Systems*, 2024.

[103] Xijun Li, Qingyu Qu, Fangzhou Zhu, Jia Zeng, Mingxuan Yuan, Kun Mao, and Jie Wang. Learning to reformulate for linear programming. *arXiv:2201.06216*, 2022.

[104] Xiaoyao Liang. *Ascend AI Processor Architecture and Programming: Principles and Applications of CANN*. Elsevier, 2020.

[105] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture*, pages 789–801. IEEE, 2021.

[106] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium*, pages 1–44. IEEE Computer Society, 2019.

[107] Hanmeng Liu, Zhiyang Teng, Ruoxi Ning, Yiran Ding, Xiulai Li, Xiaozhang Liu, and Yue Zhang. Glore: Evaluating logical reasoning of large language models. *arXiv preprint arXiv:2310.09107*, 2023.

[108] Tie-Yan Liu. *Learning to Rank for Information Retrieval*. Springer, 2011.

[109] Anqi Lu and Junchi Yan. Learning initial basis selection for linear programming via duality-inspired tripartite graph representation and comprehensive supervision. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 40831–40847, 13–19 Jul 2025.

[110] Dongsheng Luo, Wei Cheng, Wenchao Yu, Bo Zong, Jingchao Ni, Haifeng Chen, and Xiang Zhang. Parameterized explainer for graph neural network. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[111] Donato Maragno, Holly Wiberg, Dimitris Bertsimas, S. İlker Birbil, Dick den Hertog, and Adejuyigbe O. Fajemisin. Mixed-integer optimization with constraint learning. *arXiv preprint*, 2021.

[112] François Margot. Symmetry in integer linear programming. *Mathematical Programming*, 128(1-2):1–46, 2010.

[113] István Maros. *Computational techniques of the simplex method*, volume 61. Springer Science & Business Media, 2002.

[114] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, page 1112–1127, 2024.

[115] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI)*, pages 4602–4609, 2019.

[116] Mahdi Mostajabdaveh, Timothy T. Yu, Samarendra Chandan Bindu Dash, Rindranirina Ramamonjison, Jabo Serge Byusa, Giuseppe Carenini, Zirui Zhou, and Yong Zhang. Evaluating llm reasoning in the operations research domain with ORQA. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2025.

[117] Mahdi Mostajabdaveh, Timothy T. Yu, Rindranirina Ramamonjison, Giuseppe Carenini, Zirui Zhou, and Yong Zhang. Optimization modeling and verification from problem specifications using a multi-agent multi-stage llm framework. *Information Systems and Operational Research*, 0(0):1–19, 2024.

[118] Chengyi Nie, Rodrigo Fonseca, and Zhenhua Liu. Aladdin: Joint placement and scaling for SLO-aware llm serving. *arXiv preprint arXiv:2405.06856*, 2024.

[119] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744, 2022.

[120] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity.* Courier Corporation, 1998.

[121] Dimitri J. Papageorgiou, George L. Nemhauser, Joel Sokol, Myun-Seok Cheon, and Ahmet B. Keha. MIRPLib–a library of maritime inventory routing problem instances: Survey, core model, and benchmark results. *European Journal of Operational Research*, 235(2):350–366, 2014.

[122] Mihir Parmar, Nisarg Patel, Neeraj Varshney, Mutsumi Nakamura, Man Luo, Santosh Mashetty, Arindam Mitra, and Chitta Baral. Logicbench: Towards systematic evaluation of logical reasoning ability of large language models. *arXiv preprint arXiv:2404.15522*, 2024.

[123] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Shengkun Cui, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. One queue is all you need: Resolving head-of-line blocking in large language model serving. *arXiv preprint arXiv:2407.00047*, 2024.

[124] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. Queue management for SLO-oriented large language model serving. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, page 18–35, 2024.

[125] PAN Ping-Qi. *Linear programming computation.* Springer, 2014.

[126] Nikolaos Ploskas, Nikolaos V Sahinidis, and Nikolaos Samaras. A triangulation and fill-reducing initialization procedure for the simplex algorithm. *Mathematical Programming Computation*, 13:491–508, 2021.

[127] Ganesh Prasath and Shirish Karande. Synthesis of mathematical programs from natural language specifications. *arXiv preprint arXiv:2304.03287*, 2023.

[128] Chendi Qian, Didier Chételat, and Christopher Morris. Exploring the power of graph neural networks in solving linear optimization problems. In *Proceedings of The 27th International Conference on Artificial Intelligence and Statistics*, volume 238 of *Proceedings of Machine Learning Research*, pages 1432–1440, 02–04 May 2024.

[129] Chendi Qian, Didier Chételat, and Christopher Morris. Exploring the power of graph neural networks in solving linear optimization problems. In *Proceedings of the 27th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 238, pages 1432–1440, 2024.

[130] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

[131] Qingyu Qu, Xijun Li, Yunfan Zhou, Jia Zeng, Mingxuan Yuan, Jie Wang, Jinhu Lv, Kexin Liu, and Kun Mao. An improved reinforcement learning algorithm for learning to branch. *arXiv:2201.06213*, 2022.

[132] Jayant Rajgopal. Principles and applications of operations research. *Maynard's Industrial Engineering Handbook*, pages 11–27, 2004.

[133] Rindra Ramamonjison, Haley Li, Timothy Yu, Shiqi He, Vishnu Rengan, Amin Banitalebi-dehkordi, Zirui Zhou, and Yong Zhang. Augmenting operations research with auto-formulation of optimization models from problem descriptions. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 29–62, December 2022.

[134] Rindranirina Ramamonjison, Timothy T Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. NL4Opt competition: Formulating optimization problems based on their natural language descriptions. In *Advances in Neural Information Processing Systems*, pages 189–203, 2023.

[135] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed inference: Enabling efficient inference of transformer models at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2022.

[136] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing*, 27(7):950–959, 2009.

[137] Kaspar Riesen, Miquel A. Ferrer, and H. Bunke. Approximate graph edit distance in quadratic time. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2020.

[138] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel

Synnaeve. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[139] Chaoyi Ruan, Yinhe Chen, Dongqi Tian, Yandong Shi, Yongji Wu, Jialin Li, and Cheng Li. Dynaserve: Unified and elastic execution for dynamic disaggregated llm serving. *arXiv preprint arXiv:2504.09285*, 2025.

[140] Kasitinart Sangngern and Aua aree Boonperm. A new initial basis for the simplex method combined with the nonfeasible basis method. *Journal of Physics: Conference Series*, 1593(1):012002, 2020.

[141] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[142] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single GPU. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 31094–31116, 2023.

[143] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[144] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. USHER: Holistic interference avoidance for resource optimized ML inference. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 947–964, July 2024.

[145] Gursimran Singh, Xinglu Wang, Yifan Hu, Timothy Yu, Linzi Xing, Wei Jiang, Zhefeng Wang, Xiaolong Bai, Yi Li, Ying Xiong, Yong Zhang, and Zhenan Fan. Efficiently serving large multimodal models using epd disaggregation. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025.

[146] Varun Suriyanarayana, Onur Tavaslıoğlu, Ankit B Patel, and Andrew J Schaefer. Reinforcement learning of simplex pivot rules: a proof of concept. *Optimization Letters*, pages 1–13, 2022.

[147] Yinghao Tang, Tingfeng Lan, Xiuqi Huang, Hui Lu, and Wei Chen. Scorpio: Serving the right requests at the right time for heterogeneous slos in llm inference. *arXiv preprint arXiv:2505.23022*, 2025.

[148] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford Alpaca: An instruction-following LLaMA model. GitHub repository: `https://github.com/tatsu-lab/stanford_alpaca`, 2023.

[149] Hugo Touvron, Louis Martin, Kevin Stone, and other contributors. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[150] Paul Tseng. An incremental gradient(-projection) method with momentum term and adaptive stepsize rule. *SIAM Journal on Optimization*, 8(2):506–531, 1998.

[151] Pascal Van Hentenryck, Hadrien Bentz, and Laurent Michel. A rolling-horizon approach for multi-period optimization. *Annals of Operations Research*, 70:287–311, 1996.

[152] vLLM Community. vllm ascend plugin. `https://github.com/vllm-project/vllm-ascend`, 2024.

[153] Yuxuan Wan, Wenxuan Wang, Yiliu Yang, Youliang Yuan, Jen-tse Huang, Pinjia He, Wenxiang Jiao, and Michael R. Lyu. Logicasker: Evaluating and improving the logical reasoning ability of large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP 2024)*, 2024.

[154] Chao Wang, Pengfei Zuo, Zhangyu Chen, Yunkai Liang, Zhou Yu, and Ming-Chang Yang. Prefill–decode aggregation or disaggregation? unifying both for goodput-optimized llm serving. *arXiv preprint arXiv:2508.01989*, 2025.

[155] Wikipedia. MPS (format) – Wikipedia, The Free Encyclopedia, 2021.

[156] Wayne L. Winston and Jeffrey B. Goldberg. *Operations Research: Applications and Algorithms*. Thomson Brooks/Cole, 4th edition, 2004.

[157] Stephen J. Wright. *Primal-Dual Interior-Point Methods*. Society for Industrial and Applied Mathematics (SIAM), 1997.

[158] Jingfeng Wu, Yiyuan He, Minxian Xu, Xitong Gao, Kejiang Ye, and Chengzhong Xu. Unlock the potential of fine-grained llm serving via dynamic module scaling. *arXiv preprint arXiv:2507.18006*, 2025.

[159] Man Wu, Xin Zheng, Qin Zhang, Xiao Shen, Xiong Luo, Xingquan Zhu, and Shirui Pan. Graph learning under distribution shifts: A comprehensive survey on domain adaptation, out-of-distribution, and continual learning. *arXiv:2402.16374*, 2024.

[160] Yu Wu, Tongxuan Liu, Yuting Zeng, Siyu Wu, Jun Xiong, Xianzhe Dong, Hailong Yang, Ke Zhang, and Jing Li. Arrow: Adaptive scheduling mechanisms for disaggregated llm inference architecture. *arXiv preprint arXiv:2505.11916*, 2025.

[161] Jing Xia, Chuanning Cheng, Xiping Zhou, Yuxing Hu, and Peter Chun. Kunpeng 920: The first 7-nm chiplet-based 64-core arm soc for cloud services. *IEEE Micro*, 41(5):67–75, 2021.

[162] Linzi Xing, Xinglu Wang, Yuxi Feng, Zhenan Fan, Jing Xiong, Zhijiang Guo, Xiaojin Fu, Rindra Ramamonjison, Mahdi Mostajabdaveh, Xiongwei Han, Zirui Zhou, and Yong Zhang. Towards human-aligned evaluation for linear programming word problems. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING)*, pages 16347–16359, 2024.

[163] Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. *arXiv:2402.01145*, 2024.

[164] Z. Ye, Wei Chen, Liang Zhang, and Qiang Liu. Deep learning workload scheduling in gpu datacenters. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 102–111, 2024.

[165] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform bad for graph representation? *arXiv:2106.05234*, 2021.

[166] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks. In *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*, pages 9244–9255, 2019.

[167] Zahra Yousefijamarani, Xinglu Wang, Qian Wang, Morgan Lindsay Heisler, Taha Shabani, Niloofar Gholipour, Parham Yassini, Hong Chang, Kan Chen, Qiantao Zhang, Xiaolong Bai, Jiannan Wang, Ying Xiong, Yong Zhang, and Zhenan Fan. Hyperflexis: Joint design of algorithms and systems for multi-slo serving and fast scaling. *arXiv preprint arXiv:2508.15919*, 2025.

[168] Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. On explainability of graph neural networks via subgraph explorations. In *International Conference on Machine Learning (ICML)*, 2021.

[169] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Scott A. Hong, Andy Konwinski, Scott Murching, Herman Narkhede, Rohan Nishtala, Reynold S. Park, et al. A system to accelerate the machine learning lifecycle. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing*, 2020.

[170] Zhi Zeng, Hao Liu, Jian Xu, and Zongjie Wu. Reinforcement learning for adaptive penalty parameter in admm. *IEEE Transactions on Smart Grid*, 13(2):1451–1462, 2022.

[171] Wei Zhang, Zhiyu Wu, Yi Mu, Banruo Liu, Myungjin Lee, and Fan Lai. Tempo: Application-aware LLM serving with mixed SLO requirements. *arXiv preprint arXiv:2504.20068*, 2025.

[172] Yifan Zhang et al. Helix: Distributed serving of large language models via max-flow scheduling. In *Proceedings of Machine Learning and Systems (MLSys)*, 2024.

[173] Hanyu Zhao, Haoran Yang, Ziqi Song, Yuanzhi Li, Wenbo Zhang, Bo Li, Xing Liu, Xuanzhe Liu, Yang Zhou, Yong Li, et al. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 1–18, 2024.

[174] Justin Zhao, Timothy Wang, Wael Abid, Geoffrey Angus, Arnav Garg, Jeffery Kinnison, Piero Molino, Travis Addair, and Devvret Rishi. Lora land: 310 fine-tuned LLMs that rival gpt-4 — a technical report. `https://predibase.com/blog/lora-land-fine-tuned-open-source-llms-that-outperform-gpt-4`, 2024.

[175] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *arXiv preprint arXiv:2306.05685*, 2023.

[176] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.

[177] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, 2024.

[178] Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. LIMA: Less is more for alignment. *arXiv preprint arXiv:2305.11206*, 2023.

[179] Yuhang Zhou, Zhibin Wang, Guyue Liu, Shipeng Li, Xi Lin, Zibo Wang, Yongzhong Wang, Fuchun Wei, Jingyi Zhang, Zhiheng Hu, et al. Squeezing operator performance potential for the ascend architecture. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1156–1171, 2025.

[180] Ji Zhu, Saharon Rosset, Robert Tibshirani, and Trevor Hastie. 1-norm support vector machines. In *Proceedings of the 16th Annual Conference on Neural Information Processing Systems*, 2003.

[181] Kan Zhu, Haiyang Shi, Le Xu, Jiaxin Shan, Arvind Krishnamurthy, Baris Kasikci, and Liguang Xie. Polyserve: Efficient multi-SLO serving at scale. *arXiv preprint arXiv:2507.17769*, 2025.

[182] Pengfei Zuo, Huimin Lin, Junbo Deng, Nan Zou, Xingkun Yang, Yingyu Diao, Weifeng Gao, Ke Xu, Zhangyu Chen, Shirui Lu, et al. Serving large language models on huawei CloudMatrix384. *arXiv preprint arXiv:2506.12708*, 2025.