Abstract of "Machine Learning Methods for Combinatorial Optimization" by Abdelrahman Hosny Ibrahim, Ph.D., Brown University, October 2023.

Combinatorial optimization problems are fundamental in many real-world applications, where the goal is to find the optimal or near-optimal solution to a problem subject to constraints. However, most real-world optimization problems are computationally intractable, especially ones with discrete domains for the objective function. Traditional methods that solve optimization problems typically rely on handcrafted heuristics for computationally-intensive decisions during the solving process and overlook patterns in the input data for problems that are being solved repeatedly. This leads to sub-optimal solutions and longer runtimes. To improve the solving process and the solution optimality, this dissertation introduces machine learning methods within the solving procedure of existing algorithms. The premise is that machine learning methods can offer an efficient and effective way to trim the search space of candidate solutions by exploiting the patterns and structures in the problem instance data, hence reducing the time required to find a (sub)optimal solution.

This thesis makes the following contributions toward the advancement of machine learning methods for combinatorial optimization. First, we address the inefficiencies inherent in making local optimal choices within the solver's loop of a combinatorial algorithm. We propose a novel model-based reinforcement learning (RL) methodology, designed to guide the solver towards decisions in local optimizations that maximize future rewards. We showcase our method on the problem of Boolean logic synthesis of hardware chip designs, where the quality of results (QoR) depends on the sequence of optimizations applied. Our work improves on expert-crafted and heuristics-based approaches, enhancing QoRs by an average of 13%. Second, we delve into the difficulties surrounding hyperparameter tuning within combinatorial solvers. While current practices adopt a few well-performing configuration parameters and use them for new unseen problems, machine learning can help in selecting instance-aware configuration parameters. Rather than training a model to directly predict configuration parameters, we propose a new method based on deep metric learning, which enables the selection of hyperparameters by identifying similarities in problems being repeatedly solved. Empirical results on Mixed Integer Linear Programming (MILP) problems show that our method is capable of predicting a solver's hyperparameters that improve solutions' costs by up to 38%, and is practical to deploy on production environments. Third, we investigate the exploration-exploitation dilemma in existing optimization algorithms. For large problem instances that necessitate prolonged solving times, we present a model capable of predicting a solver's outcome on scalable compute resources. The goal is to let solvers explore more of the solution space, while maintaining the costs of the provisioned compute resources. In scaling electronic design automation (EDA) workloads to cloud environments, our method is able to predict the most promising exploration runs of an optimization algorithm, while reducing the total compute costs by 35%. Finally, we advocate for the development of next-generation combinatorial optimization algorithms that are native to hardware

accelerators such as GPUs. Instead of merely parallelizing existing algorithms' implementations, we propose a new approach grounded in contemporary deep learning frameworks that solves the Maximum Satisfiability (MaxSAT) problem through a single differentiable function. We demonstrate that it is feasible to develop GPU-native methods for combinatorial optimization, despite their discrete domains. This could have a significant impact on developing new approaches for tackling large-scale optimizations.

Machine Learning Methods for Combinatorial Optimization

by
Abdelrahman Hosny Ibrahim
B. S., Assiut University, 2013
Sc. M., University of Connecticut, 2016

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
October 2023

This dissertation by Abdelrahman Hosny Ibrahim is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ________________ ____

Sherief Reda, Director

Recommended to the Graduate Council

Date ________________ ____

George Konidaris, Reader

Date ________________ ____

Yu Cheng, Reader

Approved by the Graduate Council

Date ________________ ____

Thomas A. Lewis
Dean of the Graduate School

Vitae

Abdelrahman Hosny was born and raised in Assiut, Egypt. He received his B.Sc. in Computer Science from Assiut University, Egypt in 2013. He received his M.Sc. in Computer Science and Engineering from the University of Connecticut in 2016. His main research interests are in utilizing machine learning methods in combinatorial optimization problems with applications in chip design, operations research, and mixed integer linear programming solvers.

`abdelrahman_hosny@brown.edu`
`https://abdelrahmanhosny.me`
Brown University, RI, USA

Selected Publications:

1. A. Hosny, S. Hashemi, M. Shalan and S. Reda, "DRiLLS: Deep Reinforcement Learning for Logic Synthesis," 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), 2020, pp. 581-586.
2. A. Hosny and S. Reda. "Characterizing and optimizing EDA flows for the cloud." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 41.9 (2021): 3040-3051.
3. A. Hosny and S. Reda. "Automatic MILP solver configuration by learning problem similarities." Annals of Operations Research (2023): 1-28.
4. A. Hosny, M. Neseem, and S. Reda. "Sparse Bitmap Compression for Memory-Efficient Training on the Edge." In 2021 IEEE/ACM Symposium on Edge Computing (SEC), pp. 14-25. IEEE, 2021.
5. A., Tutu, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim et al. "Toward an open-source digital flow: First learnings from the openroad project." In Proceedings of the 56th Annual Design Automation Conference 2019.

6. A. Hosny, and A. B. Kahng. "Tutorial: Open-Source EDA and Machine Learning for IC Design: A Live Update." 2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID). IEEE, 2020.

7. A. Hosny and S. Reda. “torchmSAT: A Progressive Neural Network Approach To The Maximum Satisfiability Problem.” Preprint.

Acknowledgements

This thesis would not have been possible without the constant support, guidance and inspirations of many kind individuals. First and foremost, I would like to express my sincere gratitude to my advisor and mentor, Prof. Sherief Reda, whose guidance, support, and valuable insights during the course of my research has made this thesis possible. I would also like to thank Prof. George Konidaris and Prof. Yu Cheng for being on my defense committee and taking the time to review my thesis.

I am extremely thankful for the productive collaborations with all my collaborators, Soheil Hashemi, Mohamed Shalan, my advisor Prof. Sherief Reda, and many others during my work on the OpenROAD project. My work would not have been possible without them. Specifically, I would like to sincerely thank Prof. Andrew B. Kahng and Tom Spyrou for their valuable insights and guidance. I would also like to thank my fellow colleagues at Prof. Reda's group at Brown: Soheil Hashimi, Hokchhay Tann, Sofiane Chetoui, Marina Hesham, Ahmed Agiza, Jingxiao Ma, Manar Abdelatty and many others for making the last five years memorable.

I cannot express enough my immense gratitude to my wife, Menna Hasan, for her great care and for being a one-of-a-kind supporter to me during all the high and low times of my PhD journey. I would like to thank my parents and my siblings for their unwavering support and love. I have learned a lot from them. Without them, none of what I achieved today would be possible.

Last but not least, the inspirations in this thesis could not have been accomplished without the continuous encouragement of my friends. To Ali Nasser, whose constant inspiration and challenges to push my limits have been invaluable, and to George Wilkes, whose limitless support during extended study sessions has been a mainstay, I extend my heartfelt gratitude.

Contents

List of Tables **x**

List of Figures **xi**

1 Introduction **1**

1.1 Taxonomy of Optimization Problems 2

1.2 Pathways for Machine Learning in Combinatorial Optimization 4

1.3 Thesis Contributions 6

2 Background **9**

2.1 Basic Concepts 9

2.2 Greedy Algorithms 10

2.3 Dynamic Programming 11

2.4 Linear Programming 13

2.5 Branch-and-Bound 15

3 Sequential Optimization Using Reinforcement Learning **18**

3.1 Introduction 18

3.2 Related Work 20

3.3 Method 21

3.3.1 Background on Reinforcement Learning 21

3.3.2 DRiLLS 22

3.4 Empirical Results 26

3.4.1 Design Space Exploration 27

3.4.2 Comparison to Other Techniques 27

3.5 Conclusion 29

4 Hyper-parameter Tuning Using Deep Metric Learning **31**

4.1 Introduction 31

4.2 Motivation 33

4.3 Related Work 34

4.4 Preliminaries 38
4.4.1 MILP Formulation 38
4.4.2 Graph Neural Networks 38
4.4.3 Metric Learning 39
4.5 Data Validation 40
4.6 Method 42
4.6.1 Learning MILP Similarity 42
4.6.2 Predicting Configuration Parameters 45
4.7 Empirical Results 47
4.7.1 Dataset 47
4.7.2 Experimental Setup 48
4.7.3 Instance Embedding 49
4.7.4 Prediction Accuracy 51
4.7.5 Comparing to Baselines 51
4.8 Conclusion 53

5 Efficient Exploration Using Predictive Modeling 55

5.1 Introduction 55
5.2 Preliminaries 57
5.3 Related Work 59
5.4 Method 61
5.4.1 EDA Flow Characterization 62
5.4.2 Runtime Prediction 65
5.4.3 Optimizing Virtual Machine Provisioning 68
5.5 Empirical Results 71
5.6 Conclusion 75

6 Fast GPU-native Combinatorial Optimization 76

6.1 Introduction 76
6.2 Motivation 77
6.3 Preliminaries 79
6.4 Related Work 80
6.5 Method 81
6.6 Empirical Results 85
6.7 Conclusion 88

7 Summary and Possible Extensions 89

7.1 Summary of Contributions 89
7.2 Potential Future Research 92

A Data Management in Metric Learning 94

B Dataset Details in torchmSAT **96**

C Raw Results in torchmSAT **99**

D GPU Acceleration in torchmSAT **104**

Bibliography **107**

List of Tables

3.1 Multi-objective reward function in reinforcement learning 24
3.2 Area-delay comparison of logic synthesis optimization using reinformcement learning 30

4.1 Instance-aware solver configuration methods 38
4.2 Mixed integer linear programming dataset 47
4.3 Results of using deep metric learning for configuring solvers' hyperparameters 52

5.1 System configuration used for EDA flow characterization 62
5.2 Results of minimizing the total cost of EDA cloud deployments using GNNs 73

6.1 Results of using backpropagation in solving MaxSAT 88

B.1 MaxSAT dataset 98

C.1 Raw results for solving MaxSAT using backpropagation 100

List of Figures

1.1 Taxonomy of optimization problems 2
1.2 Summary of machine learning methods for combinatorial optimization 4

3.1 DRiLLS framework architecture 23
3.2 Traces of DRiLLS agent navigating the design space 27
3.3 Design area vs. delay trade-offs using DRiLLS 29

4.1 Effect of configuration parameters on the solution cost using SCIP 32
4.2 Cost reduction by searching and evaluating the configuration space 33
4.3 A high-level pipeline of using metric learning for parameters configuration 34
4.4 Correlation between problem instance similarities and their solver's configuration . . 41
4.5 Correlation between problem instances and their costs under different configurations 42
4.6 Overview of training deep metric learning model to learn problem similarities 43
4.7 Vector representations of MILP problem instances visualized using t-SNE 50
4.8 Results of using configurations from the nearest neighbor in the learned metric space 51
4.9 Similarity in the learned embedding space 53

5.1 A reference EDA flow on the cloud 57
5.2 Workflow of optimizing EDA cloud deployments 61
5.3 Performance characterization of four representative EDA jobs 62
5.4 Routing speedup for different designs 63
5.5 VM placement scenarios in a cloud environment. 64
5.6 Impact of VM placement on the total runtime 64
5.7 Overall system architecture for analyzing and optimizing EDA flows on the cloud . . 65
5.8 Our proposed runtime prediction model for EDA jobs 66
5.9 Graph extraction from design netlist 67
5.10 Graph convolution operation for chip designs 67
5.11 Runtime as a function of the design size 71
5.12 Runtime prediction errors using our GNN model 72
5.13 Cost savings from running our multi-choice knapsack optimization algorithm 74
5.14 Further cost savings from choosing to run EDA jobs on spot VMs 74

6.1 Existing MaxSAT solvers vs. our proposed backpropagation-based method 78
6.2 Overview of torchmSAT neural network architecture 82
6.3 Performance of torchmSAT as compared to the state-of-the-art MaxSAT solvers . . . 86
6.4 Running torchmSAT on CPU vs. GPU 87

D.1 Running torchmSAT on CPU vs. GPU (10mins) 105
D.2 Running torchmSAT on CPU vs. GPU (5mins) 105
D.3 Running torchmSAT on CPU vs. GPU (1min.) 106

Chapter 1

Introduction

A mathematical optimization problem is a type of problem in which a specified objective function is either minimized or maximized with respect to the domain of its input variables. These types of problems are prevalent across a wide array of fields, including engineering design [166], operations research [151], economics [73], and many others [189, 183]. They play a critical role in decision-making processes and are essential in finding the most effective solutions for various challenges.

Over the decades, various techniques have been developed to identify optimal solutions for these optimization problems [187]. Some of these methods are based on calculus, such as gradient descent [33], Lagrange multiplier methods [25], or Newton's method [138, 150], while others rely on numerical or iterative techniques, such as the simplex algorithm [41] or simulated annealing [93]. These methods have been widely used and have proven to be effective for a broad range of applications.

However, many real-world optimization problems are considered intractable, as they are part of Karp's NP-Complete list [83] – a collection of problems whose solutions are difficult to find in polynomial time. Due to their inherent complexity, these problems pose significant challenges for traditional optimization techniques, often requiring the development of novel approaches.

State-of-the-art algorithms for solving intractable optimization problems have primarily relied on expert-crafted heuristics to find suboptimal solutions within a reasonable time [187]. These heuristics are carefully designed to guide the search process in a way that leads to finding better solutions faster than evaluating the entire solution space. Some examples of general-purpose heuristic-based approaches include genetic algorithms [66], simulated annealing [93], and Tabu search [59]. Moreover, problem-specific heuristics-based algorithms have also been well-established, such as Adaptive Large Neighborhood Search (ALNS) [153] for the Vehicle Routing Problem [42], Open-WBO [125] for the MaxSAT problem [56], and Meet-in-the-middle [68] for the Knapsack problem [86].

Despite their widespread application and success, these state-of-the-art algorithms often have a limitation: they are generic in nature and do not take into account the specific characteristics of the input data. This means that they may overlook potential gains that could be achieved by making decisions based on patterns or structure in the data. As a result, there is an ongoing effort to develop new optimization techniques that can better exploit the unique features of individual problem instances, leading to more efficient and effective solutions [21, 98].

In this dissertation, we argue that leveraging problem-specific data can enhance the performance of optimization algorithms by incorporating machine learning (ML) models, leading to improved efficiency and solution quality. The main idea behind this direction is to utilize patterns and results from previously encountered problem instances to inform and guide the search for optimal solutions in new problem instances. Problem-specific data can be comprised of features and characteristics of previously seen or solved optimization problems. By incorporating this data into the solving process, ML models learn to identify patterns and relationships that can be exploited to enhance the performance of optimization algorithms.

In order to introduce the various pathways for integrating machine learning in combinatorial optimization, we first present a taxonomy of optimization problems in Section 1.1. The goal is to set the context for the contributions of this dissertation. Next, we present a high-level overview of the pathways we have studied in this dissertation in Section 1.2, and summarize our contributions in Section 1.3.

1.1 Taxonomy of Optimization Problems

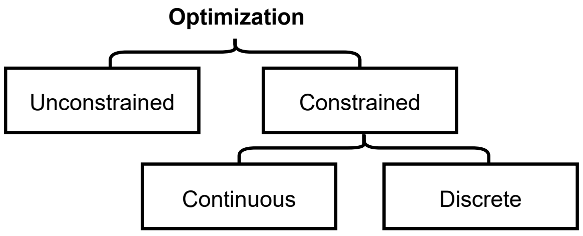


Figure 1.1: Taxonomy of optimization problems

Optimization problems are often encountered in various contexts and can be formulated as either minimization or maximization problems. In the case of minimization, the objective function is represented as $\arg\min_{\mathbf{x}} f(\mathbf{x})$, while for maximization, it is denoted as $\arg\max_{\mathbf{x}} f(\mathbf{x})$. The complexity of these optimization problems is often determined by the domain of their input variables, $\mathbf{x}$, which defines the search space that an algorithm needs to explore.

A high-level taxonomy of optimization problems can be visualized in Figure 1.1, which categorizes them into various classes based on their characteristics. One such class is unconstrained optimization problems, where the input domain has no explicit boundaries, or the boundaries are considered soft constraints. In this scenario, the input variables $\mathbf{x}$ typically belong to the real number domain, i.e., $\mathbf{x} \in \mathbb{R}$. A well-known example of this type of problem is minimizing the loss function in a machine learning model, where the input variables $\mathbf{x}$ represent the weights of the model.

On the other hand, constrained optimization problems introduce additional "subject to" terms in the problem formulation, which restrict the feasible region of the input variables. For instance, if the input values $\mathbf{x}$ are limited to a specific continuous range, such as $\mathbf{x} \in [1.5, 5.25]$, the optimization algorithms must ensure the feasibility of the solution before determining its optimality. In this case, the input domain is continuous, and algorithms need to account for these constraints during the search process. In more challenging cases, the input variables $\mathbf{x}$ are restricted to a discrete set of values, e.g., $\mathbf{x} \in 0, 1$. Solving optimization problems with discrete input domains can be extremely difficult, as finding the optimal solution often requires an exhaustive brute-force search. This approach can quickly become computationally infeasible for problems with a large number of input variables or complex constraints. In such cases, researchers often resort to employing heuristics or approximation algorithms to find near-optimal solutions within a reasonable time frame.

State-of-the-art methods for constrained optimization often rely on handcrafted heuristics to find a (sub)optimal solution within a reasonable time frame [39]. These methods typically employ well-established optimization techniques, ranging from linear and integer programming to gradient-based and evolutionary algorithms, to tackle various types of problems. Although these approaches have proven to be effective in many cases, their reliance on generic problem-solving strategies may limit their ability to exploit domain-specific knowledge or to adapt to the unique characteristics of individual problem instances.

A vast number of solvers have been developed to address different variations of optimization problems, depending on factors such as the objective function (linear or non-linear) and the constraints (continuous or discrete) [47]. By design, these solvers are meant to be generic and do not make any assumptions about the numerical structure of the input variables. However, this generality comes at the cost of potentially overlooking valuable insights that could be gained from the problem's structure.

Over the last two decades, it has become increasingly apparent that many optimization problems are being solved repeatedly across various domains, generating a wealth of data that can offer valuable insights into the optimization process [79, 143]. For instance, in the field of Electronic Design Automation (EDA), optimizing integrated circuits is a critical aspect of developing new chips, particularly in advanced technology nodes [79]. Likewise, combinatorial optimization problems have numerous applications in operations research [143], such as last-mile delivery companies regularly solving the vehicle routing problem as daily delivery tasks (stops and routes) change [114]. Despite the repeated and frequent occurrence of these problems, the data generated during the optimization process often remains untapped. Existing algorithms and generic solvers tend to overlook the opportunity to leverage the patterns present in the input data for making more informed and effective optimization decisions. This gap in the current state-of-the-art methods presents a unique opportunity for further research and development.

The primary goal of this dissertation is to explore possible avenues for systematically integrating domain-specific knowledge and historical data into the optimization process, with the aim of enhancing the efficiency and solution quality of existing algorithms. By leveraging the wealth of information available from previously solved problems, it is possible to develop new techniques or augment existing methods that can adapt more effectively to the unique characteristics of individual problem instances, ultimately leading to better optimization outcomes.

1.2 Pathways for Machine Learning in Combinatorial Optimization

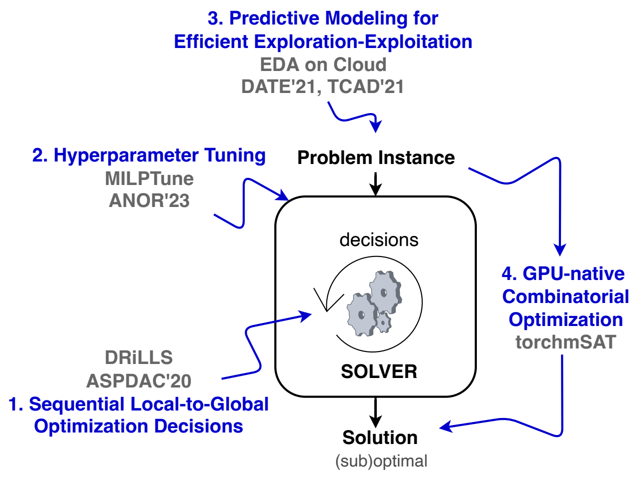


Figure 1.2: Summary of our proposed machine learning methods for combinatorial optimization.

In this dissertation, we focus on constrained optimization problems, where various techniques and methods have been developed to solve problem instances characterized by an objective function, $f(\mathbf{x})$, and a set of constraints for the input variables, $\mathbf{x}$. The primary goal of these methods is to find an optimal solution, either by minimizing or maximizing the objective function, while adhering to the specified constraints and time limits. In recent years, machine learning (ML) models have been integrated into the optimization process to improve solvers' performance and adaptability. In this context, we present a comprehensive recipe illustrating different approaches to incorporate ML models into constrained optimization solvers, depicted in Figure 1.2. Our framework can be divided into four main components: (1) sequential local-to-global optimization decisions, (2) hyperparameter tuning, (3) predictive modeling for efficient exploration-exploitation, and (4) fast GPU-native combinatorial optimization. Each of these pathways plays a vital role in enhancing the solving process and achieving better results within the given constraints and time limits.

Firstly, machine learning (ML) methods can be incorporated into the solving process of a numerical solution, playing a critical role in determining local optimization decisions that affect the global structure of the problem. In such scenarios, the iterative optimization process can be effectively modeled as Markov Decision Process (MDP) [18]. A model-based reinforcement learning (RL) approach, which utilizes learned models of the environment to plan and make decisions, can be employed in this context [169]. Our research contributions, detailed in Chapter 3, demonstrate the development of a reinforcement learning agent that successfully minimized the area of a chip layout while adhering to specific timing constraints. This approach highlights the potential of model-based RL in addressing complex optimization problems with multiple constraints. In a similar vein, researchers at Google have also explored the benefits of model-based RL in chip design optimization. They developed an

RL agent capable of planning chip layouts for their Tensor Processing Units (TPUs) [127]. This RL agent leverages a graph-based representation of the chip design problem, allowing for efficient exploration and exploitation of the solution space. The success of this approach further underscores the potential of model-based reinforcement learning techniques in addressing challenging optimization problems with real-world implications. Expanding the scope of ML integration in optimization problems, such as those addressed in chip design, can lead to more efficient and effective solutions that account for multiple constraints and dynamically adapt to problem instances. By leveraging the power of reinforcement learning and Markov Decision Processes, researchers and practitioners can tackle complex numerical problems and develop better optimization algorithms.

Secondly, machine learning (ML) methods can also be integrated externally to the solving loop of a generic solver (e.g. Mixed Integer Linear Programming solvers), playing a complementary role in enhancing the optimization process. In this approach, the output of the ML model can be utilized to tune the hyperparameters of the optimization algorithms. We present our research contributions in Chapter 4, showcasing the potential of ML methods integrated externally to the solving loop. We employ deep metric learning techniques to configure Mixed Integer Linear Programming solvers effectively. Deep metric learning focuses on learning meaningful distance metrics between data points, which can be useful for configuring solvers based on problem instance similarities or differences. Unlike supervised learning methods which are impractical to deploy in this context, predicting hyperparameter configurations from similar problem instances that are previously solved introduces negligible overhead and is practical to deploy and continuously tune in production environments.

Thirdly, we address the exploration-exploitation dilemma in looking for an optimal solution in a massive solution space. The hypothesis is that letting a solver explore more of the solution space improves the solution quality, but also comes at a significant runtime and computation costs. Therefore, in Chapter 5, we demonstrate how predictive modeling can be leveraged to help an agent select the most promising solver runs to continue exploring while reducing the overall cost and meeting time constraints. Expanding the integration of ML methods, both inside and outside the solving loop, can significantly enhance the performance of optimization solvers. Collectively, we show that supervised learning, metric learning and reinforcement learning are powerful methods for enhancing solutions of existing solvers.

Finally, we call for a fresh view on approaching combinatorial optimization problems and present a novel framework to model such problems as a single differentiable function capable of producing fast approximate solutions. By leveraging ML acceleration hardware, such as Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs), and implementing GPU-native algorithms using contemporary deep learning libraries, the backpropagation process can effectively become the solving process, leading to more efficient and faster solutions. Unlike parallelizing classic algorithms on multi-core environments, the idea behind this approach is to harness the power of specialized hardware and ML techniques to optimize the solving process itself. By formulating the problem instance in a way that is conducive to modeling as a neural network, researchers can tap into the inherent parallelism and computational capabilities of GPUs and TPUs. To that end, our research contributions in

Chapter 6 approximates solutions to the Maximum Satisfiability Problem (MaxSAT), showing the potential of leveraging GPU parallelism in such a context. We propose torchmSAT – a progressive approach that continually refines and improves its solutions over time. One of the core advantages of torchmSAT is its independence from a SAT oracle, a feature that differentiates it from traditional MaxSAT solvers. This makes our method more self-sufficient and less reliant on external components. Experimental results show that our method outperforms two existing MaxSAT solvers, and is on par with another state-of-the-art solver for small to medium problem sizes. Additionally, torchmSAT is able to benefit from GPU acceleration, allowing for more rapid exploration of feasible solution regions. Despite some limitations, torchmSAT represents a promising step forward in GPU-native combinatorial optimization algorithms.

In conclusion, the integration of machine learning models into the optimization process, whether it be within the solving loop, externally, or through the prediction of feasible solutions, holds great potential for improving the performance and adaptability of optimization algorithms. By combining the strengths of ML techniques and specialized hardware, researchers and practitioners can tackle complex optimization problems more effectively, leading to innovative solutions that address the challenges of real-world applications.

1.3 Thesis Contributions

In this section, we outline the major contributions made in this thesis regarding the integration of machine learning models into the optimization process of combinatorial problems.

DRiLLS: Deep Reinforcement Learning for Logic Synthesis. Logic synthesis requires extensive tuning of the synthesis optimization flow where the quality of results (QoR) depends on the sequence of optimizations used. Efficient design space exploration is challenging due to the exponential number of possible optimization permutations. Therefore, automating the optimization process is necessary. In this work, we propose a novel reinforcement learning-based methodology that navigates the optimization space without human intervention. We demonstrate the training of an Advantage Actor Critic (A2C) agent that seeks to minimize area subject to a timing constraint. Using the proposed methodology, designs can be optimized autonomously with no-humans in-loop. Evaluation on the comprehensive EPFL benchmark suite shows that the agent outperforms existing exploration methodologies and improves QoRs by an average of 13%.

Characterizing and Optimizing EDA Flows for the Cloud. Design space exploration in logic synthesis and parameter tuning in physical design require a massive amount of compute resources in order to meet tapeout schedules. To address this need, cloud computing provides semiconductor and electronics companies with instant access to scalable compute resources. However, deploying EDA jobs on the cloud requires EDA teams to deeply understand the characteristics of their jobs in cloud environments. Unfortunately, there has been little to no public information on these characteristics. Thus, in this thesis, we first formulate the problem of deploying EDA jobs to the cloud. To address the problem, we characterize the performance of four EDA main applications, namely: synthesis, placement, routing and static timing analysis. We show that different EDA jobs require different compute configurations in order to achieve the best performance. Using observations from our characterization, we propose a novel model based on Graph Convolutional Networks to predict the total runtime of a given stage on different configurations. Our model achieves a prediction accuracy of 87%. Furthermore, we present a new formulation for optimizing cloud deployments in order to reduce costs while meeting deadline constraints. We present a pseudo-polynomial optimal solution using a multi-choice knapsack mapping that reduces deployment costs by 35.29%, with minimal overhead to the total runtime. In addition, we describe a cloud-ready solution, called EDA Analytics Central, for the continuous optimization of a design across an EDA flow. We used this system in building our runtime prediction model.

Automatic MILP Solver Configuration By Learning Problem Similarities. A large number of real-world optimization problems can be formulated as Mixed Integer Linear Programs (MILP). MILP solvers expose numerous configuration parameters to control their internal algorithms. Solutions, and their associated costs or runtimes, are significantly affected by the choice of the configuration parameters, even when problem instances have the same number of decision variables and constraints. On one hand, using the default solver configuration leads to poor suboptimal solutions. On the other hand, searching and evaluating a large number of configurations for every problem instance is time-consuming and, in some cases, infeasible. In this study, we aim to predict configuration parameters for unseen problem instances that yield lower-cost solutions without the time overhead of searching-and-evaluating configurations at the solving time. Toward that goal, we first investigate the cost correlation of MILP problem instances that come from the same distribution when solved using different configurations. We show that instances that have similar costs using one solver configuration also have similar costs using another solver configuration in the same runtime environment. After that, we present a methodology based on Deep Metric Learning to learn MILP similarities that correlate with their final solutions' costs. At inference time, given a new problem instance, it is first projected into the learned metric space using the trained model, and configuration parameters are instantly predicted using previously-explored configurations from the nearest neighbor instance in the learned embedding space. Empirical results on real-world problem benchmarks show that our method predicts configuration parameters that improve solutions' costs by up to 38% compared to existing approaches.

torchmSAT: A Progressive Neural Network Approach To The Maximum Satisfiability Problem The remarkable achievements of machine learning techniques in analyzing discrete structures have drawn significant attention towards their integration into combinatorial optimization algorithms. Typically, these methodologies improve existing solvers by injecting learned models within the solving loop to enhance the efficiency of the search process. In this work, we derive a single differentiable function capable of approximating solutions for the Maximum Satisfiability Problem (MaxSAT). Then, we present a novel neural network architecture to model our differentiable function, and progressively solve MaxSAT using backpropagation. This approach eliminates the need for labeled data or a neural network training phase, as the training process functions as the solving algorithm. Additionally, we analyze the feasibility of leveraging the computational power of GPUs to accelerate these computations. Experimental results on challenging MaxSAT instances show that our proposed methodology outperforms two existing MaxSAT solvers, and is on par with another in terms of solution cost, without necessitating any training or access to an underlying SAT solver. Given that numerous NP-hard problems can be reduced to MaxSAT, our novel technique paves the way for a new generation of solvers poised to benefit from neural network hardware acceleration.

Chapter 2

Background

In this chapter, we introduce some fundamental algorithmic ideas which will be used as baselines in the subsequent chapters. The aim of reviewing these ideas in a concise format is twofold. One one hand, we provide the reader, who is new to combinatorial optimization, with a basic introduction to understand the context of our contributions in this dissertation. On the other hand, we establish a common conceptual cornerstone from which the other chapters can improve these ideas as they are inherently integrated in each of our contributions. We will also formalize the notation used in this dissertation in its more general formulation.

2.1 Basic Concepts

As presented in Section 1.2, this dissertation focuses on constrained optimization problems. The most general form of such problems can be represented as:

$$\begin{aligned} \arg\min_{\mathbf{x}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}^\top \mathbf{x} \geq \mathbf{b}, \\ & \mathbf{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}, \end{aligned} \tag{2.1}$$

where $\mathbf{c} \in \mathbb{R}^n$ denotes the coefficients of the linear objective, $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$ denote the coefficients and upper bounds of the linear constraints, respectively, n is the total number of decision variables, $p \leq n$ is the number of integer-constrained variables, and m is the number of linear constraints. The goal is to find feasible assignments for $\mathbf{x}$ that minimize the objective $\mathbf{c}^\top \mathbf{x}$.

While simple and concise, this formulation encodes a large number of practical problems. If p is zero, i.e. the domain of $\mathbf{x}$ is continuous, the problem becomes relatively easy to solve even with a massive number of variables. This will be referred to as a Linear Program in Subsection 2.4. When p is a positive number, the problem becomes intractable and only approximate algorithms can find a sub-optimal solution in a polynomial time.

2.2 Greedy Algorithms

In a greedy algorithm, a solver starts with an empty solution, i.e. assignment of $\mathbf{x}$, and repeatedly extends the current solution by making locally optimal choices at each step until a complete solution is obtained. Greedy methods can also start with a complete feasible solution and then try to improve the current solution further via local moves. While greedy algorithms can be applied to certain MILP problems, they are not guaranteed to find the optimal solution in all cases because they make locally optimal choices at each step. Greedy algorithms are more suited for specific problem types or as heuristics to quickly find a suboptimal solution.

For example, the Knapsack problem can be formulated as a MILP problem and solved using a greedy algorithm:

$$
\begin{aligned}
\arg\max_{\mathbf{x}} \quad & \sum_{i=1}^{n} v_i x_i \\
\text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leq W \\
& x_i \in \{0,1\} \quad \forall i \in \{1,\ldots,n\}
\end{aligned}
\tag{2.2}
$$

where v_i represents the value of item i, w_i represents the weight of item i, W is the maximum weight capacity of the knapsack, and n is the number of items. The MILP formulation is meant to decide which items to take to maximize the total value, while not exceeding the weight capacity of the knapsack. A greedy algorithm to solve this problem is to sort the items by *value-to-weight* ratio in a descending order, then iteratively add items to the knapsack as long as they don't exceed the weight capacity.

Greedy algorithms exhibit advantages in solving optimization problems.

1. **Simplicity:** The greedy algorithm is easy to understand and implement. The algorithm typically involves sorting items by a specific criterion (e.g., value-to-weight ratio) and iteratively selecting items based on this criterion.

2. **Computational efficiency:** Greedy algorithms are usually computationally efficient as they make a single pass through the sorted list of items. The time complexity is dominated by the sorting step, which is typically $\mathcal{O}(n \log n)$, where n is the number of items.

3. **Speed:** Greedy algorithms can quickly provide a solution, which can be useful in practice when an exact solution may not be required, or when the problem size is too large for more sophisticated algorithms to handle within a reasonable time frame.

4. **Good performance on certain problem types:** For some specific problem types, such as the fractional knapsack problem, the greedy algorithm is guaranteed to find the optimal solution.

However, greedy algorithms also have significant limitations.

1. **Suboptimal solutions:** The main drawback of greedy algorithms is that they do not always yield optimal solutions for all problem types, especially for problems that do not have a greedy-choice property. Greedy algorithms make locally optimal choices at each step, which may not always lead to the globally optimal solution.

2. **Sensitivity to input data:** The performance of a greedy algorithm can be sensitive to the input data and may depend on the order in which items are considered. In some cases, a small change in input data can significantly impact the quality of the solution.

3. **Lack of exploration:** Greedy algorithms do not explore alternative solutions and commit to a specific choice at each step. This lack of exploration can result in suboptimal solutions, as the algorithm does not consider the consequences of its choices beyond the current step.

4. **Limited applicability:** Greedy algorithms may not be suitable for all MILP problems, as they rely on problem-specific heuristics and work best on problems with specific structures. For more general MILP problems, other methods like dynamic programming, or branch-and-bound may be more appropriate for finding the optimal solution.

Accordingly, better algorithms can address some of these limitations by improving optimality, tolerating data variance, or offering enhanced exploration capabilities. We use a greedy algorithm baseline in our work in Chapter 3.

2.3 Dynamic Programming

Dynamic programming breaks down problems into simpler, overlapping subproblems. It is particularly well-suited for problems that exhibit optimal substructure and overlapping subproblems, which are common characteristics of many combinatorial optimization problems. The fundamental steps of dynamic programming are as follows:

1. **Subproblems:** Break the original problem down into smaller, simpler subproblems. These subproblems are usually defined by the problem's parameters or dimensions, and their solutions can be combined to form the optimal solution for the larger problem. Determine the base cases, which are the simplest subproblems with known solutions.

2. **The recurrence relation:** Develop a mathematical expression that captures the relationship between the solution of a subproblem and the solutions of its smaller subproblems. This expression, known as the recurrence relation, is the basis of the dynamic programming approach and is used to build the solution iteratively.

3. **Iterative search:** Design a table or a data structure to store the results of the subproblems. Using the recurrence relation and base cases, solve the subproblems in a systematic manner,

either iteratively (bottom-up approach) or recursively (top-down approach). In the bottom-up approach, the algorithm starts with the base cases and builds the solution for the larger problem iteratively by solving the subproblems in increasing order of complexity. In the top-down approach, also known as memoization, the algorithm starts with the original problem and recursively breaks it down into smaller subproblems, caching the results to avoid redundant computations.

Dynamic programming addresses the exploration limitation of greedy algorithms. While still providing suboptimal solutions, dynamic programming is empirically better than a simple greedy algorithm. Take for example the knapsack problem presented in Equation 2.2. We can develop a dynamic programming solution that breaks down the problem into smaller ones by create a table K with dimensions $(n+1) \times (W+1)$, where n is the number of items and W is the maximum weight capacity of the knapsack. The rows represent items from 0 to n, and the columns represent possible weight capacities from 0 to W. The search starts by initializing the first row and the first column of the table K to 0, as there is no value when no items are selected or when the weight capacity is 0. After that, the algorithm iterates through the items ($i = 1$ to n) and the weight capacities ($w = 1$ to W). For each item i and weight capacity w, the algorithm calculates the maximum value (i.e. Objective in Equation 2.2) by either including the item i or not including it if the weight of item i (w_i) is less than or equal to the current weight capacity (w):

- **Including item i:** value $=$ v_i (value of item i) $+$ $K[i-1][w-w_i]$ (the maximum value that can be obtained from the remaining capacity after including item i).
- **Excluding item i:** value $=$ $K[i-1][w]$ (the maximum value that can be obtained without item i).

Then, $K[i][w] = \max(\text{value\_including\_i}, \text{value\_excluding\_i})$. If the weight of item i (w_i) is greater than the current weight capacity (W), then $K[i][w] = K[i-1][w]$ (the maximum value that can be obtained without item i). The value in the bottom-right corner of the table, $K[n][W]$, represents the maximum value that can be obtained for the given knapsack problem.

The presented algorithm satisfies the MILP formulation. The objective function in the MILP formulation is to maximize the total value of the items in the knapsack, which corresponds to the bottom-right value in the dynamic programming table ($K[n][W]$). The constraint in the MILP formulation ensures that the total weight of the selected items does not exceed the knapsack's weight capacity (W). In the dynamic programming algorithm, this constraint is incorporated by considering weight capacities from 0 to W in the table and making decisions based on the weight of each item. The decision variables in the MILP formulation are binary ($x_i \in 0, 1$). In the dynamic programming algorithm, the choices are made by either including or not including an item in the knapsack, which implicitly represents the binary nature of the decision variables.

Dynamic programming has advantages over greedy algorithms in solving optimization problems.

1. **Optimal solution:** Dynamic programming guarantees an optimal solution for the problem, unlike greedy algorithms, which may only provide suboptimal solutions in some cases.

2. **Applicability:** Dynamic programming can be applied to a wide range of problems, including various combinatorial optimization problems, where the problem can be broken down into smaller overlapping subproblems.

3. **Time complexity:** Dynamic programming can significantly reduce the time complexity of a problem, as it avoids redundant computations by storing and reusing the solutions of subproblems.

4. **Flexibility:** Dynamic programming can be adapted to various problem variations and constraints, making it a versatile technique for problem-solving.

Nevertheless, dynamic programming suffers from the following limitations:

1. **Space complexity:** Dynamic programming often requires a table or data structure to store the results of subproblems, which can lead to high space complexity, especially for large problem sizes or high-dimensional problems.

2. **Problem formulation:** Dynamic programming requires the problem to be broken down into smaller overlapping subproblems and a recursive relationship to be identified. Formulating the problem in this manner can be challenging for some problems.

3. **Traceback complexity:** In some cases, reconstructing the optimal solution from the table or data structure can be complex and time-consuming.

4. **Scalability:** For very large problem instances or problems with a high number of dimensions, dynamic programming may not scale well due to the increased time and space complexity.

In summary dynamic programming improves on greedy algorithms, but presents a pseudo-polynomial algorithm which does not scale well on large problems. We incorporate dynamic programming in our work in Chapter 5.

2.4 Linear Programming

Linear programming (LP) is an optimization technique used to find the best possible solution to Equation 2.1 *if and only if* $p = 0$. In other words, the domain of the variables must be continuous. To solve the original MILP, we can create a relaxation of the problem by allowing the decision variables to be continuous rather than integers. This relaxation transforms the problem into a Linear Programming (LP) problem, which can be solved more efficiently than MILP problems. Since LP methods could produce fractional solutions, finding a feasible solution for the original problem is achieved by rounding the fractional solution, which in itself may not necessarily be optimal. However, rounding the LP relaxation solution does not always guarantee optimality, especially for more complex instances of MILP.

The Simplex method is a widely used algorithm for solving linear programming problems. The algorithm works by iteratively moving along the edges of the feasible region (defined by the constraints) to find the optimal solution. Variables in a problem can be classified as either basic or non-basic variables. These classifications depend on the current solution and the tableau representation of the problem. A non-basic variable is a variable that is not part of the current basis (set of linearly independent variables) in the tableau. Non-basic variables are set to zero in the current basic feasible solution. In contrast, basic variables correspond to the variables that form the current basis and have non-zero values in the basic feasible solution. The algorithm works as follows:

1. **Initialization:** Convert the linear programming problem into standard form, which involves adding slack variables to transform inequality constraints into equality constraints. Then, determine an initial basic feasible solution. If the reduced costs (the coefficients of the non-basic variables in the objective function) are all nonnegative, the current solution is optimal. Otherwise, proceed to the next step.

2. **Pivot selection:** Choose a nonbasic variable with a negative reduced cost to enter the basis (this is called the entering variable). Determine the leaving variable by applying the minimum ratio test, which selects the variable that reaches its upper bound first when the entering variable is increased.

3. **Update the solution:** Perform a pivot operation, which involves updating the basis and the solution by swapping the entering and leaving variables. This results in a new basic feasible solution. Repeat these steps until an optimal solution is found or the problem is determined to be unbounded.

In practice, there are numerous optimizations and variations of the algorithm, such as the use of different pivot selection rules and the Two-Phase Simplex method for finding an initial basic feasible solution. Additionally, it's important to note that this algorithm assumes the problem is in maximization form; if the problem is a minimization problem, it needs to be converted to a maximization problem (e.g., by negating the objective function) before applying the Simplex method.

Linear programming, including the Simplex method and its descendants, are well-studied and understood. They are known for:

1. **Generalizability:** Unlike greedy algorithms and dynamic programming, LP is not limited to a specific problem type, and does not need to exploit certain problem characteristics to find a solution.

2. **Versatility:** Linear programming can be applied to a wide range of problems in real-world applications such as economics, engineering, and logistics, among others.

3. **Efficient algorithms:** There are several efficient algorithms, such as the Simplex method and the interior point method, for solving linear programming problems. In many cases, these algorithms converge to the optimal solution in a reasonable number of iterations.

4. **Sensitivity analysis:** Linear programming allows for sensitivity analysis, which helps understand how changes in the coefficients of the objective function or constraints affect the optimal solution. This feature is particularly useful for decision-making and scenario analysis.

However, linear programming has its own limitation:

1. **Inability to handle integer constraints:** Linear programming is designed for problems with continuous variables. It is not directly applicable to problems with integer or mixed-integer constraints, which require specialized algorithms like branch-and-bound or cutting-plane methods.

2. **Sensitive to problem formulation:** The performance of linear programming algorithms can be sensitive to the problem's formulation. Some formulations may lead to faster convergence or more stable solutions, while others may cause slow convergence or cycling.

3. **Scalability:** Although linear programming can handle large-scale problems, the time and memory requirements can still grow rapidly with the problem size, making it difficult to solve very large or high-dimensional problems.

MILP problems can be solved using a combination of LP techniques and other methods, such as branch-and-bound, branch-and-cut, or cutting-plane algorithms. The Simplex method can be used to solve the LP relaxations within these algorithms, providing bounds and helping to guide the search for the optimal integer solution. In the branch-and-bound algorithm, for instance, the search tree is explored by branching on fractional variables and solving LP relaxations at each node to obtain bounds that can be used to prune the tree and reduce the search space as we will discuss in the next section. Chapter 6 reflects on linear programming methods while it investigates gradient-based methods to solve a MILP.

2.5 Branch-and-Bound

To tackle MILP problems (Equation 2.1), one common approach is to first relax the integer constraints, creating an LP relaxation. Solving this LP relaxation provides a bound on the optimal solution of the original MILP problem. The branch-and-bound algorithm is a well-established technique to systematically explore the search space, combining the power of LP solvers with additional methods to handle the integer constraints and find the optimal solution for the MILP problem. To solve a MILP using branch-and-bound, the algorithm follows these high-level steps:

1. **Relax the integer constraints:** Convert the MILP problem into an LP problem by temporarily ignoring the integer constraints. This is called the LP relaxation, and it provides an upper bound (for maximization problems) or lower bound (for minimization problems) for the optimal solution of the MILP problem.

2. **Solve the LP relaxation:** Use an LP solver, such as the Simplex method, to find the optimal solution to the LP relaxation. If the solution already satisfies the integer constraints, it is the optimal solution for the original MILP problem. Otherwise, proceed to the next step.

3. **Branch:** Create new subproblems by branching on one of the fractional variables in the current LP relaxation solution. This involves adding new constraints to the problem that force the selected variable to take either the floor or ceiling of its fractional value. These new subproblems are called child nodes, and the process of creating them is known as branching.

4. **Bound:** Solve the LP relaxation for each child node to obtain bounds on the optimal solution. If the LP relaxation solution of a child node is infeasible or has a worse objective value than the best known integer solution, the node can be discarded (pruned). If the LP relaxation solution of a child node satisfies the integer constraints, update the best known integer solution.

5. **Select and explore:** Choose one of the remaining unexplored child nodes and repeat steps 3 and 4. Continue this process of branching, bounding, and exploring until all nodes have been pruned or the search space has been exhausted.

In its essence, the branch-and-bound algorithm is a search for the optimal solution. Instead of a brute-force search, the idea of the method is to prune the search space at each bounding step. Branch-and-bound is widely deployed in real world settings and remains the state-of-the-art in solving MILPs as they have plausible advantages:

1. **Optimality:** Branch-and-bound guarantees finding the global optimal solution (if it exists) within a finite amount of time, unlike previous methods that may only find suboptimal solutions.

2. **Pruning:** The bounding mechanism of branch-and-bound efficiently prunes the search space by eliminating subproblems that cannot lead to better solutions than the current best known solution. This significantly reduces the number of nodes to be explored in the search tree.

3. **Applicability:** The branch-and-bound framework can be applied to various types of optimization problems, including MILP, combinatorial optimization problems, and non-linear optimization problems with certain modifications.

Nonetheless, limitations of the branch-and-bound method are well-studied and can be summarized below:

1. **Exponential time complexity:** The worst-case time complexity of branch-and-bound is exponential, which means that solving large-scale or complex MILP problems can be computationally expensive and time-consuming.

2. **Memory requirements:** Branch-and-bound algorithms may require significant memory to store the search tree, especially for large-scale problems or problems with a large number of integer variables.

3. **Difficulty handling non-convex problems:** While branch-and-bound can be adapted to solve some non-linear and non-convex optimization problems, it may struggle with problems that have a large number of local optima or a highly non-convex search space.

In summary, branch-and-bound offers several advantages, such as guaranteeing global optimality, pruning the search space efficiently, and its flexibility and applicability to various types of optimization problems. However, it also has some limitations, including exponential time complexity, memory requirements, sensitivity to problem formulation and branching strategy, and difficulty handling non-convex problems. In Chapter 4, we will explore methods to improve the performance of solvers that are based on the branch-and-bound method.

Chapter 3

Sequential Optimization Using Reinforcement Learning

3.1 Introduction

Logic synthesis is a fundamental process in the field of electronic design automation, which entails the systematic conversion of high-level design descriptions, typically expressed in hardware description languages, into optimized gate-level representations. This transformation enables the realization of digital circuits that meet stringent performance, power, and area constraints. As an essential step in the design of application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs), logic synthesis bridges the gap between abstract design specifications and physically implementable hardware, ensuring the efficient and accurate translation of a designer's intent into tangible electronic systems.

Contemporary logic synthesis tools utilize And-Inverter Graphs (AIGs) to encode the essential attributes for Boolean function optimization. The logic synthesis procedure is predominantly comprised of three interdependent steps: pre-mapping optimizations, technology mapping, and post-mapping optimizations. In the initial pre-mapping optimization phase, the AIG undergoes technology-agnostic transformations to minimize graph size and, consequently, reduce the total area while conforming to delay constraints. Subsequently, during the technology mapping stage, the generic intermediate nodes are assigned to specific standard cells from a given technology (e.g., ASIC standard cells). Lastly, the post-mapping optimization step encompasses technology-dependent refinements such as up-sizing and down-sizing of standard cells to further enhance the design's performance and resource utilization.

Designing an efficient logic synthesis optimization flow is a complex endeavor, necessitating expertise from seasoned designers. The primary challenge in creating such flows stems from the exponential expansion of the search space due to the numerous available transformations. Specifically, the various transformation options and their differing recurrences and permutations can profoundly impact the Quality of Results (QoR) [188, 191]. Moreover, the escalating disparity and intricacy in circuit designs have further exacerbated the optimization flow design process.

It is imperative to acknowledge that a universally optimal pre-defined sequence of transformations does not exist, as it is incapable of generating the best QoR for all potential circuits. Consequently, the optimization flows mandate meticulous fine-tuning for each individual input to achieve the desired level of performance and efficiency. Concurrently, advancements in machine learning (ML), particularly reinforcement learning (RL), have empowered autonomous agents to enhance their proficiency in navigating intricate environments. Recent accomplishments in implementing these agents have demonstrated performance levels comparable to, or even surpassing, human expertise [162, 76]. For example, AlphaGo has been recognized as the inaugural computer program to defeat a professional human Go player [162].

In logic synthesis frameworks, there exist a rich set of primitive transformations, each optimizing the circuit using a different algorithm (e.g. balancing, restructuring). Permutations of these optimizations generate different QoR. Furthermore, different repetitions of the same transformations affect the QoR and therefore result in an **exponentially growing search space**. Synthesis flows for large circuits often have tens or hundreds of optimization commands.

We define $\mathbb{A} = \{a_1, a_2, ...a_n\}$ as the set of available optimizations in a logic synthesis tool. Let k be the length of an optimization flow. Assuming that optimizations can be processed independently (e.g. no constraint for running a_1 before a_2), there exists n^k possible flows. Yu *et. al.* show that different flows indeed result in divergent area and delay results [188]. While human experts have traditionally guided the search, the increasing complexity of the designs and synthesis optimizations have highlighted the need for an autonomous exploration methodology.

In light of this, we propose a novel methodology based on RL that aims at producing logic synthesis optimization flows. Our contributions in this work are as follows:

- We tackle the intricate task of devising an efficient design space exploration strategy for logic synthesis optimization. By transforming the problem into a game-like environment that can be comprehended by a reinforcement learning agent, we effectively leverage the potential of advanced machine learning techniques to facilitate this exploration. To this end, we formulate a comprehensive feature set, derived from the characteristics of the And-Inverter Graph (AIG), which aids in capturing essential aspects of the optimization problem.

- We introduce a novel multi-objective reward function that guides the reinforcement learning agent in its pursuit of minimizing the circuit area while adhering to specific delay constraints. This approach enables a more targeted and effective exploration of the vast design space, ultimately leading to optimized logic synthesis solutions that satisfy the stringent performance, power, and area requirements of modern electronic systems.

- We present DRiLLS (**D**eep **Re**inforcement **L**earning-based **L**ogic **S**ynthesis), a novel framework that harnesses the power of reinforcement learning to create logic synthesis optimization flows. By employing this approach, we effectively eliminate the reliance on human experts for tuning synthesis parameters, thereby streamlining the optimization process. DRiLLS is a versatile framework, applicable to a diverse range of circuit designs without necessitating specific configurations or setups. This adaptability ensures that DRiLLS is a valuable tool for the broader electronic design community, enabling the efficient generation of optimized logic synthesis solutions across various design contexts.

- We showcase the efficacy of our proposed methodology using the EPFL arithmetic benchmark suite [9] as a testbed. In our evaluation, we compare DRiLLS against the best results obtained from the benchmark suite when mapped to a standard cell library, as well as against classical optimization algorithms, including greedy heuristics. Furthermore, we assess expert-developed flows as a baseline for comparison purposes. Our findings reveal that DRiLLS consistently surpasses the performance of previous techniques [9, 186], highlighting its potential as a robust and effective solution for logic synthesis optimization in various design scenarios.

The rest of the chapter summarizes relevant previous work in Section 3.2. Next, in Section 3.3, we present a background on RL that is utilized in our approach and provide a detailed discussion on the proposed methodology. After that, we summarize our experimental results in Section 3.4. Finally, Section 3.5 summarizes the main contributions of this work.

3.2 Related Work

The study of methodologies for design space exploration (DSE) in computing systems and electronic design automation (EDA) technology has garnered considerable attention from researchers. At the architectural level, Ipek *et al.* introduce predictive models founded on neural networks to examine the design space encompassing memory, processor, and multi-chip processor domains, while forecasting performance [74]. In a similar vein, Ozisikyilmaz *et al.* investigate design space pruning through performance prediction for various computing configurations [142]. Their approach employs three statistical models, each fine-tuned on a small subset of potential designs.

Additionally, a learning-based methodology that leverages random forests for design space exploration in high-level synthesis flows has been proposed [112]. This work further underscores the growing interest in utilizing advanced machine learning techniques for optimizing design processes in the field of electronic systems.

In more recent research, Ziegler *et al.* introduced SynTunSys [191], a synthesis parameter tuning system that iteratively amalgamates optimizations while concentrating on the "survivor set" for further examination. In each iteration, candidate scenarios are assigned estimated costs, and those with the lowest cost values undergo evaluation. The cost estimator is subsequently updated based on the learned costs [190].

Alternatively, Yu *et al.* formulated the problem of logic synthesis design flow composition as a classification problem [188], employing convolutional neural networks to categorize sample flows encoded as images into "angel" or "devil" flows. Their approach necessitates a fixed length for optimization and a large sample size of pre-defined optimization flows for training and testing. Our work diverges from previous studies in that we propose the use of a reinforcement learning agent to explore the search space with the objective of optimizing specific synthesis metrics (e.g., area and delay). This enables variable length optimization flows without requiring sample flows for training.

In recent years, reinforcement learning (RL) agents have exhibited remarkable prowess in navigating complex environments [130, 162]. While early RL research primarily focused on domains with fully observable state spaces or where features could be handcrafted, Mnih *et al.* broadened these capabilities by introducing deep Q-networks (DQN) [130]. By leveraging the latest advancements in deep neural networks, their agent attains state-of-the-art performance, rivaling human abilities.

Building upon the capabilities of RL agents, Lillicrap *et al.* expanded the action domain to encompass continuous domains, specifically targeting physical domains in their work [109]. This development further underscores the growing potential of RL in addressing a diverse range of complex problems across various domains.

3.3 Method

3.3.1 Background on Reinforcement Learning

In this section, we briefly discuss the background necessary for developing our methodology. In reinforcement learning, an agent is trained to choose actions, in an iterative manner, that maximize its expected future reward. Formally,

- At each iteration k, and based on the current state of the system s_k, the agent chooses an action a_k from a finite set of possible actions $\mathbb{A}$.
- With the application of the action at step k, the system moves to the next state s_{k+1} and a reward of $g(s_k, a_k)$ is then provided to the agent.
- The agent iteratively applies actions, changing the state of the system and getting rewards. It is then trained based on the collected experience to move toward maximizing its reward in future iterations.

A policy is defined as a mapping $\mathcal{M}$ that, for each given state, assigns a probability mass function $\mathcal{M}(\cdot|a)$ for an action [95]. There are two major categories for implementing the mapping $\mathcal{M}$: value-based and policy-based methodologies. In value-based methods (e.g. Q-learning) a value function is learned by the system that effectively maps $(state, action)$ pairs to a singular value [180], and picks the maximum over all possible actions. On the contrary, in policy-based methods (e.g. policy gradient), the optimization is performed directly on the policy ($\mathcal{M}$) [170]. Actor Critic algorithms [95], as a hybrid class, combine the benefits of both aforementioned classes.

In actor critic methods, a tunable critic network provides a measure of how good the taken action is (similar to a reward function), while the tunable actor network chooses the actions based on the current state. More formally defined, the actor policy function is of the form $\pi_\theta(s, a)$, and the critic function is of the form $\hat{q}_w(s,a)$; where s, and a represent the state and the action, while θ, and w represent the tunable parameters within each network. Therefore, there exist two sets of parameters, one for each network, that need to be optimized. The gradient optimization for the critic network is performed as,

$$\Delta w = \beta \delta \nabla_w \hat{q}_w(s_k, a_k) \tag{3.1}$$

where β sets different learning rate for policy and value. δ is the temporal difference error, which is defined as

$$\delta = R(s,a) + \gamma \hat{q}_w(s_{k+1}, a_{k+1}) - \hat{q}_w(s_k, a_k) \tag{3.2}$$

where γ is the discount factor. Similarly, the gradient optimization for the policy update (actor network) is then defined as

$$\Delta \theta = \alpha \nabla_\theta (\log \pi_\theta(s,a)) \hat{q}_w(s,a) \tag{3.3}$$

where α sets the learning rate. Note that actor network policy update is a function of the critic network as well, which allows it to take into consideration not only the current state of the environment, but also the history of learning from the critic network.

While very effective, actor critic models can suffer from high variability in action probabilities. Advantage functions are proposed as a solution to reduce this variability. The advantage function is defined as

$$A(s,a) = Q(s,a) - V(s) \tag{3.4}$$

where $Q(s,a)$ represents the Q value for action a in state s, and $V(s)$ represents the average value for the given state. In this work, we do not want to compute $Q(s,a)$. Instead, we formulate an estimate of the advantage function as

$$A(s) = r + \gamma V(s') - V(s) \tag{3.5}$$

where r is the current reward and γ is the discount factor. This achieves the same result without learning the Q function [96]. Next, we describe the proposed DSE methodology based on reinforcement learning.

3.3.2 DRiLLS

DRiLLS, an acronym for **D**eep **Re**inforcement **L**earning-based **L**ogic **S**ynthesis, adeptly transforms the design space exploration problem into a game-like environment. Contrasting with many reinforcement learning environments where gamification governs the environmental behavior, the task at hand encompasses combinatorial optimization for a given circuit design. This presents a unique

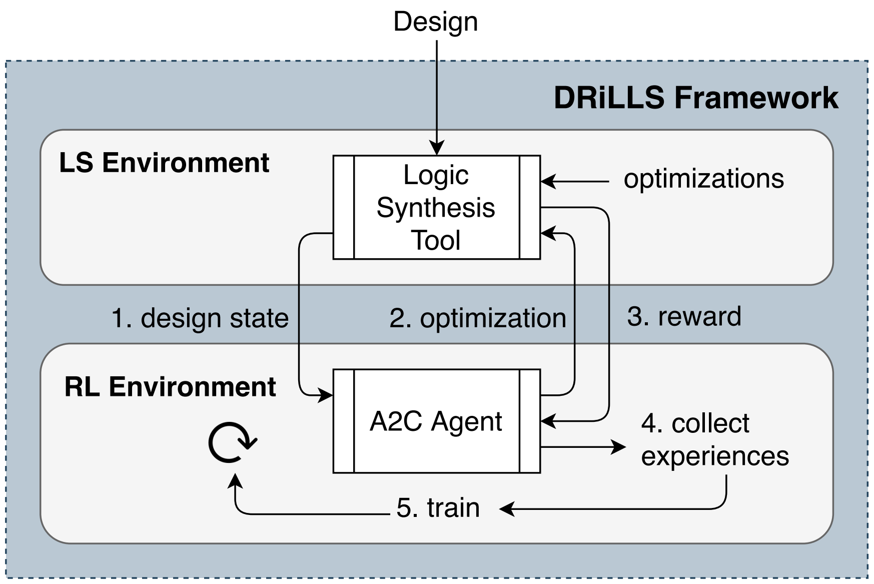


Figure 3.1: The architecture of DRiLLS Framework. Numbers on the arrows represent the workflow of our methodology, and are illustrated separately in Section 3.3.2.

challenge in defining the game's state (i.e., environment) and establishing long-term incentives for the agent to explore the design space without succumbing to local minima.

Figure 3.1 illustrates the architecture of our proposed methodology, which comprises two primary components: the *Logic Synthesis* environment, responsible for setting up the design space exploration problem as a reinforcement learning task, and the *Reinforcement Learning* environment, which utilizes an *Advantage Actor Critic agent (A2C)* to traverse the environment in search of the optimal optimization for a given state. Next, we delve into each component and the interplay between them in greater detail.

1. Design State Representation. To model the combinatorial optimization for logic synthesis as a game, we define the *state* of the logic synthesis environment as a set of metrics obtained from the synthesis tool for a given circuit design, which serves as the feature set for the A2C agent. As mentioned earlier, the *state* also embodies the environment's response to an *optimization* suggested by the second component of our framework, namely the *Agent*. We specifically extract the following state vector:

$$\textbf{AIG state} = \begin{bmatrix} \#\ primary\ I/O \\ \#\ nodes \\ \#\ edges \\ \#\ levels \\ \#\ latches \\ \%\ ANDs \\ \%\ NOTs \end{bmatrix}.$$

To confine the states within a particular range, as necessitated by the agent's neural networks,

Table 3.1: Formulation of the multi-objective reward function. ***Decr.*** stands for Decrease and ***Incr.*** stands for Increase.

			Optimizing (Area)		
			Decr.	*None*	*Incr.*
Constraint (Delay)	Met		+++	*0*	-
	Not Met	*Decr.*	+++	++	+
		None	++	0	- -
		Incr.	-	- -	- - -

we normalize all state values by their corresponding initial input design values. Normalization is also essential for model generalization, allowing for application to unseen designs. While optimizations alter all elements in the *state* vector except for the number of primary inputs and outputs, values within the *state* vector depict representative characteristics of the circuit. For instance, a large # *nodes* value guides the agent towards reducing the number of nodes, which can be achieved by restructuring the current AIG and maximizing the sharing of other nodes available in the current network (e.g., *resub* and $refactor$ commands in ABC). Furthermore, a large # *levels* value directs the agent towards selecting a *balance* transformation. Consequently, the *state* vector represents the circuit design at a specific optimization step and aligns with the optimization space, which we will discuss next.

2. Optimization Space. The agent explores the search space consisting of seven primitive transformations within the ABC synthesis framework [128]. Specifically, $\mathbb{A}$ = {resub, resub -z, rewrite, rewrite -z, refactor, refactor -z, balance}. The first six transformations aim to reduce the size of the AIG, while the final transformation, balance, decreases the number of levels. These transformations interact with the *state* vector representation discussed previously, making them suitable for the reward function described in the following section.

3. Reward Function. We define a multi-objective reward function that considers the changes in both design area and delay. Specifically, the agent is rewarded for reducing the design area while maintaining the delay under a pre-defined constraint value. Table 3.1 presents the reward formulation of this function. For each metric (design area or delay), a transformation may decrease, increase, or make no change to the metric. Consequently, we assign the highest reward (represented as +++) for a transformation performed on a given AIG state that reduces the area and satisfies the delay constraint. We give the lowest negative reward when the transformation executed increases the design area and delay without meeting the constraint. Between these two extremes, the values and magnitudes of the reward have been carefully selected to aid the agent's exploration. Fundamentally, we prioritize meeting the delay constraint. When not met, a positive reward is also given if the delay has improved (i.e., decreased). This reward strategy prevents the agent from receiving negative rewards in all attempts in situations where the delay constraint is too stringent for the design to meet. Furthermore, when the area increases and the delay decreases (but does not meet the constraint), a small positive reward is given as the agent is learning from not meeting the constraint.

Algorithm 1: DRiLLS Framework

```
Input  : Design, Primitive Transformations
Output: Optimization_Flow
1  $env$ = Initialize(LS_Env);
2  $agent$ = Initialize(A2C);
3  for $episode = 1$ to $N$ do
4      $episode\_design\_states$ = [];
5      $optimization\_sequence$ = [];
6      $synth\_rewards$ = [];
7      $design\_state$ = $env$.reset();
8      for $iteration = 1$ to $k$ do
9          $opt\_probs$ = agent.ActorForward($design\_state$);
10         $primitive\_opt$ = RandomChoice($opt\_prob$);
11         [$next\_design\_state$, $synth\_reward$] = $env$.perform($primitive\_opt$);
12         $episode\_design\_states$.append($design\_state$);
13         $optimization\_sequence$.append($primitive\_opt$);
14         $synth\_rewards$.append($synth\_reward$);
15         $design\_state$ = $next\_design\_state$;
16     end
17     $episode\_rewards$ = DiscountRewards($synth\_rewards$, $gamma$)
18     $loss$ = agent.OptimizerForward($episode\_design\_states$, $optimization\_sequence$,
         $episode\_rewards$);
19     $agent$.update($loss$);
20     log($episode$);
21 end
```

This reward formulation has proven to be efficient, as we will discuss in the following section.

4. Collecting Experiences. Algorithm 1 outlines the operation of our proposed methodology. In this algorithm, lines 1 and 2 initialize the logic synthesis environment and the agent, respectively. The agent is then trained over N episodes, where in each episode, the logic synthesis environment is restarted; i.e., the original input design is reloaded (line 7). Following this, in lines 8-16, the agent iteratively suggests a sequence of k primitive optimizations to create the optimization flow. Specifically, in line 9, the agent computes the probability distribution for selecting a primitive optimization from the optimization space, $\mathbb{A}$. Then, in line 10, one of the primitive optimizations is chosen according to the probability distribution calculated in line 9. Next, in line 11, the chosen optimization is executed to determine its impact on the $design_state$. Furthermore, the reward is computed using the reward function presented in Table 3.1. Subsequently, we store the synthesis state, the optimization performed, and the reward in pre-initialized variables. Ultimately, we transition the state of the agent to the state after performing the optimization. The number of iterations is limited by k to provide the game with a termination condition, as the optimization improvements on a given circuit design diminish in later iterations. After completing all iterations, we train the A2C agent using the collected experiences, as we will discuss next.

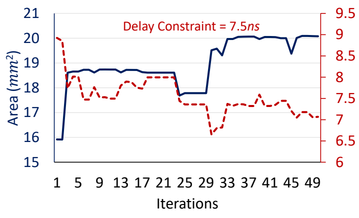
5. A2C Agent Training. The training process begins by discounting the delay rewards over iterations in order to prioritize earlier iterations in selecting an effective optimization (line 17). Following that, in lines 18-19, the loss is calculated, and the actor and critic networks are trained to minimize the loss value as described next. As discussed in Section 3.3.1, the agent has hybrid policy-based and value-based networks, referred to as the actor and critic, respectively. Both networks have an input layer of size equal to the *AIG state* vector length. Moreover, a reward, r, is passed to the critic network for training, while a discounted reward is passed to the actor network (Equation 3.5). The actor network outputs a probability distribution over the available transformations. Consequently, the output layer in the actor network has a size equal to that of $\mathbb{A}$. Since the agent is initialized with random parameters, the transformations chosen at the beginning of the training process may not necessarily be optimal. The parameters of both networks are updated to reduce the loss using a gradient-based optimizer. This procedure is repeated for a predefined number of times (called episodes), during which the agent is trained to predict improved optimization flows. In fact, the choice of a hybrid reinforcement learning architecture is well-suited for combinatorial optimization tasks, as it allows the agent to explore diverse optimization sequences while maintaining a path towards optimal designs.

3.4 Empirical Results

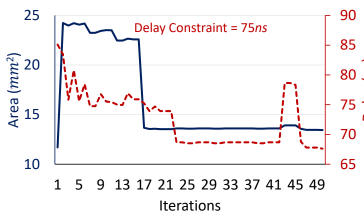
We demonstrate the proposed methodology using the open-source synthesis framework ABC v1.01 [128]. DRiLLS is implemented in Python v3.5.2, and TensorFlow r1.12 [2] is employed to train the A2C agent neural networks. All experiments are synthesized using ASAP7, a 7 nm standard cell library in a typical processing corner. We evaluate our framework on EPFL arithmetic benchmarks [9], which display a wide range of circuit characteristics. The characteristics of the evaluated benchmarks (e.g., I/Os, number of nodes, edges, and levels) can be found in [9]. The experimental parameters are set as follows:

- **Episodes (N):** 50, **Iterations (k):** 50
- **Networks Size:** *Actor:* 2 fully connected layers, 20 hidden units each. *Critic:* one hidden layer with 10 units.
- **Weight initialization:** Xavier initialization [58]
- **Optimizer:** Adam [90], **Learning Rate (α):** 0.01
- **Discount rate (γ):** 0.99

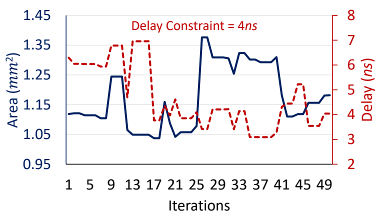
We use a small number of layers, as we observe that deeper neural networks exhibit random behavior and do not train well in this framework. This can be attributed to the small number of features and transformations used. The experimental results are obtained using a machine with an Intel Xeon 2x14 cores@2.4 GHz, 128GB RAM, and 1x500GB SSD, running Ubuntu 16.04 LTS. We will now present our results.



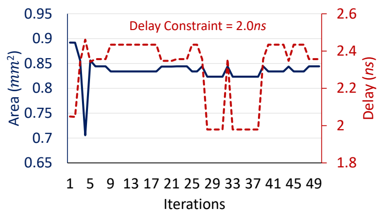
(a) Log2



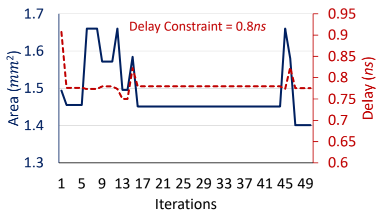
(b) Divisor



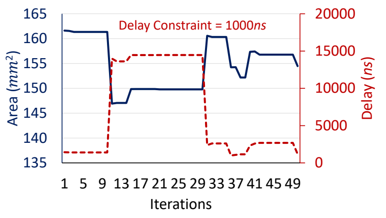
(c) Max



(d) Adder



(e) Barrel Shifter



(f) Hypotenuse

Figure 3.2: Traces of DRiLLS agent navigating the design space to find a design with a minimum area while meeting the delay constraint.

3.4.1 Design Space Exploration

Figure 3.2 illustrates the agent's search for an optimized design that minimizes the area while meeting the delay constraint. We plot one episode that finds the global minimum for several representative benchmarks. In general, Figure 3.2 depicts the agent's attempts to balance between reducing the design area and satisfying the delay constraint. For example, we can observe the agent's various trials to execute a transformation that decreases the delay to meet the constraint but increases the design area, such as iteration 30 in Log2 and iteration 26 in Max. Occasionally, exploration saturates, as we can see in the nearly straight lines during some iterations. This demonstrates the actor-critic networks' ability to guide the exploration while occasionally exploring other transformations that might reveal new search paths.

3.4.2 Comparison to Other Techniques

We compare the agent's performance against *EPFL best results*, *expert-crafted scripts*, and a *greedy heuristic algorithm*:

1. *EPFL best results*: best results are provided for size and depth. We compare against best results for size, since it is more relevant to the agent's nature of optimizing for area when mapping to a standard cell library.

2. *Expert-crafted scripts*: we maintain a record of expert-crafted synthesis optimizations derived from [186].

3. *Greedy heuristics algorithm*: we developed a baseline comparison that takes an initial input design and spawns parallel threads to perform each of the given AIG transformations on the

design. Afterwards, each thread performs the mapping step using the delay constraint. The algorithm then evaluates the mapped designs from all threads, and keeps the one with the minimum area for the next iteration. After that, the process is repeated until two iterations yield the same area.

Table 3.2 presents the results of the mentioned comparisons. The area and delay for the initial design are obtained by loading the non-optimized designs in ABC and mapping them to ASAP7 without performing any transformation on the AIG. The delay is reported using the built-in timer in ABC (using the *stime* command). We use the initial run to select a delay constraint value that challenges all the methods studied in this work. We make the following observations:

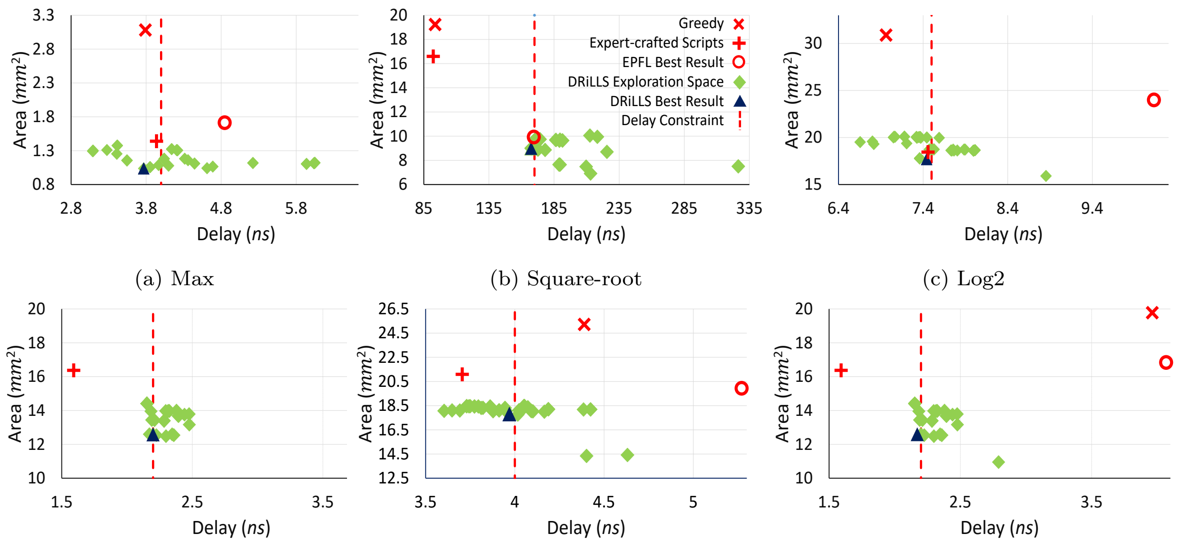
- The greedy algorithm has a single optimization target (area). Although the delay constraint was met in 4 designs, it is attributed to the best-effort mapping step that considers the delay constraint. The increase in the area occurs in the first iteration that tries to meet the delay constraint while mapping. Since the algorithm meets the stop criteria in the first few iterations, it fails to reduce the area subject to a delay constraint. Results show the smallest average area improvement.

- Although expert-crafted synthesis scripts have not improved the designs' areas, they produced optimized designs that meet the delay constraint in 9 out of 10 designs. This is not surprising, as the techniques used aim to meet the delay constraint, therefore accepting near-optimal area results [186].

- EPFL best results have shown decent improvements in 3 designs, meeting the delay constraint in 4 of them. Although we benchmarked on the best results in terms of size, not depth, it is reasonable that their optimization techniques have not been designed for standard cell library mapping.

- DRiLLS agent meets the delay constraint in all designs while simultaneously improving the design area by an average of 13.19%. In the two designs that DRiLLS increased their area, it, in fact, met the delay constraint which the un-optimized design did not meet. This proves that the reward function defined earlier is an effective one for training the agent. Moreover, DRiLLS outperforms EPFL best result in all designs except *Barrel shifter*.

Figure 3.3 expands upon Table 3.2 by displaying the area-delay trade-offs provided by DRiLLS in comparison to the greedy algorithm, expert-developed synthesis scripts, and the EPFL best results for six benchmark cases. We define **exploration run time** as the total time taken by the agent, which includes interaction with the *Logic Synthesis Environment*, extraction of AIG characteristics, and optimization of the agent network parameters. The exploration time for the smallest design (Adder) is *3.25 minutes*, while for the largest (Hypotenuse), it takes *25.46 minutes*. The average exploration time per episode is *12.76 minutes*. It is worth noting that once a model is trained on a specific circuit design, it can be used (reloaded) for new explorations on different circuits without the need for retraining.



(a) Max (b) Square-root (c) Log2

(d) Sin (e) Multipler (f) Square

Figure 3.3: Design area vs. delay trade-offs. The vertical dotted red line shows the delay constraint. For each benchmark, DRiLLS exploration space is indicated in green diamonds. A highlighted triangle represents the best optimized design that meets the delay constraint. Other methods are shown in red color with a cross mark, a plus mark and a circle for greedy, expert-crafted and EPFL result respectively.

3.5 Conclusion

The primary objective of developing DRiLLS is to create an autonomous framework capable of exploring the optimization space for a given circuit design and achieving high Quality of Result (QoR) without human intervention. Modeling this problem in a reinforcement learning context provides the machine with a trial-and-error approach, similar to how human experts gain experience in optimizing designs. In this study, we present a reinforcement learning-based methodology that facilitates autonomous and efficient exploration of the logic synthesis design space. Our proposed methodology transforms the complex search space into a "game" in which an advantage actor-critic (A2C) agent learns to maximize its reward (minimize area subject to a delay constraint) by iteratively selecting primitive transformations with the highest expected reward. We have developed an AIG state representation that effectively represents the feature set of a design state. Moreover, we introduce a novel multi-objective reward function that guides the agent's exploration process, allowing it to find a minimal design area subject to delay constraint. Evaluating ten representative benchmarks, our proposed methodology demonstrates superior performance compared to existing methods. DRiLLS proves the viability of utilizing Reinforcement Learning for combinatorial optimization of hardware circuit designs. It has considerable potential for application in related physical synthesis tasks, eliminating the need for human expertise. The framework is open-source under a permissive license (BSD-3) and publicly available on GitHub[1].

[1]https://github.com/scale-lab/DRiLLS

Table 3.2: Area-delay comparison of logic synthesis optimization results. A greedy algorithm optimizes for area. Expert-crafted scripts are derived from [186]. EPFL best results for size are available at [9].

Benchmark	***Delay***	**Initial Design**		**Greedy**			**Expert-crafted [186]**			**EPFL Best Size [9]**			**DRiLLS**		
	Constr. *(ns)*	***Area*** *(um^2)*	***Delay*** *(ns)*	***Area*** *(um^2)*	***Delay*** *(ns)*	***Impr.*** *(%)*	***Area*** *(um^2)*	***Delay*** *(ns)*	***Impr.*** *(%)*	***Area*** *(um^2)*	***Delay*** *(ns)*	***Impr.*** *(%)*	***Area*** *(um^2)*	***Delay*** *(ns)*	***Impr.*** *(%)*
Adder	2.00	867	2.02	1011	4.10	-16%	1772	1.82	-104%	1690	1.87	-94%	823	1.97	5%
B. Shifter	0.80	2499	1.03	2935	0.66	-17%	1534	0.77	38%	1040	0.77	58%	1400	0.77	43%
Divisor	75.00	12388	75.83	22439	79.14	-81%	21167	65.05	-70%	16031	74.91	-29%	13441	67.61	-8%
Hypotenuse	1000.00	176938	1774.32	236271	563.12	-33%	210828	525.34	-19%	169468	1503.88	4%	154227	995.95	12%
Log2	7.50	19633	7.63	30893	6.96	-57%	18451	7.45	6%	23999	10.12	-22%	17687	7.44	9%
Max	4.00	1427	4.48	3082	3.79	-115%	1440	3.93	-0.88%	1713	4.84	-20%	1037	3.76	27%
Multiplier	4.00	19617	3.83	25219	4.38	-28%	21094	3.70	-7%	19940	5.27	-1%	17797	3.96	9%
Sin	3.80	3893	3.65	5501	2.88	-41%	4421	2.19	-13%	4892	4.14	-25%	3050	3.76	21%
Square-root	170.00	11719	329.46	19233	93.71	-64%	16594	92.30	-41%	9934	169.46	15%	9002	167.47	23%
Square	2.20	11157	2.27	19776	3.96	-77%	16373	1.59	-46%	16838	4.06	-50%	12584	2.199	-12%
Avg. Area Imprv.		0.00%		-53.31%			-26.00%			-16.69%			13.19%		
Constraint Met		2/10		4/10			9/10			4/10			10/10		

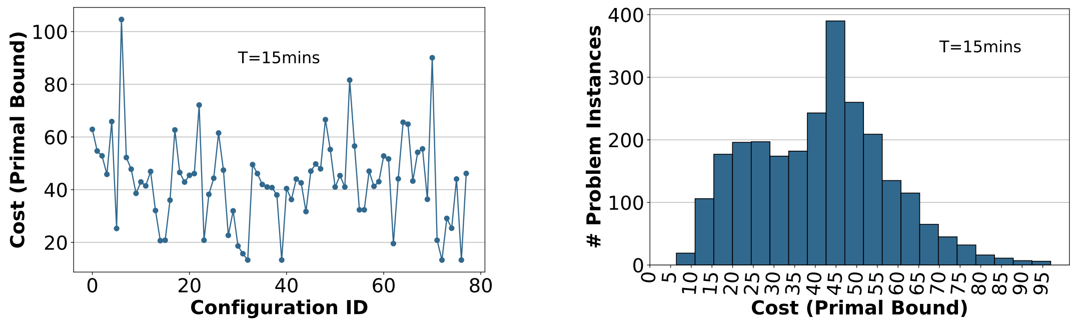
Chapter 4

Hyper-parameter Tuning Using Deep Metric Learning

4.1 Introduction

Mixed Integer Linear Programs (MILP) is a class of NP-hard problems where the goal is to minimize a linear objective function subject to linear constraints, with some or all decision variables restricted to integer or binary values [54]. This formulation has applications in numerous fields, such as transportation, retail, manufacturing and management [143, 17]. For example, last-mile delivery companies repeatedly solve the vehicle routing problem as daily delivery tasks (stops and routes) change, with the goal of minimizing total delivery costs [114]. Similarly, crew scheduling problems have to be solved daily or weekly in the aviation industry, where the MILP formulation is the most practical notation for expressing such problems [46]. Over the years, solvers have been well researched and practically engineered to address these problems, such as SCIP [55], CPLEX [120], and Gurobi [30]. These solvers mostly use branch-and-bound methods combined with heuristics to direct the search process for solving a MILP [4]. In order to tune their behavior, they expose a large number of configuration parameters that control the search trajectory. In implementing a branching strategy, SCIP exposes configuration parameters to help in selecting the most promising decision variable to branch on at each node in the branch-and-bound tree, which significantly impact the efficiency and effectiveness of the solver. For example, the branching score function, $branching/scorefunc \in \{p, s, q\}$, and the branching score factor, $branching/scorefac \in \{0, 1\}$, help evaluate the potential of expanding a specific branch in the search tree. Those are just two of more than 2500 parameters with integer, continuous or categorical configuration spaces.

Automatic algorithm configuration is the task of identifying optimal parameter configurations for solving unseen problem instances by training on a collection of representative problem instances [50]. This process can be divided into two distinct phases. The primary tuning phase involves selecting a parameters configuration based on a set of training instances representative of a specific



(a) Same Problem Instance

(b) Same Configuration

Figure 4.1: Effect of configuration parameters on the solution cost using SCIP [115] ($T = 15mins$ as suggested by [129]). (a) changing the parameters of the branch-and-bound algorithm on the same instance. (b) using a single configuration on different instances. All problem instances have 195 decision variables and 1083 constraints.

problem. Subsequently, during the testing phase, the chosen parameter configuration is employed to tackle unseen instances of the same problem. The objective is to identify, during the tuning phase, a parameters configuration that minimizes a particular cost metric over the set of instances that will be encountered during the deployment phase. There exist a rich literature on efficient search methods for automatic algorithm configuration for optimization problems [113, 27, 28, 123, 67, 87]. They mostly differ in how they navigate the huge search space to find potential configurations fast during the tuning phase.

In Figure 4.1, we investigate the effect of configuration parameters on problem instances from the Item Placement benchmark in the ML4CO dataset [129]. In Figure 4.1(a), different configuration parameters directly impact the solution's cost of the same problem instance. A solution is an assignment to the decision variables, and its cost is the value of the objective function in the formulated MILP, which is to be minimized. In addition, using a single configuration for all problem instances does not yield the same solution's cost as shown in Figure 4.1(b). As a result, branch-and-bound configuration parameters significantly affect the solution quality. Note that increasing the time limit of the solver might not necessarily lead to better solutions since modern solvers are already heavily optimized to find solutions fast. Moreover, time limits are usually determined by the real-world context where a solver is deployed. Therefore, searching and evaluating configuration parameters is desirable as it can potentially improve the cost of the solutions. In Figure 4.2, we search the configuration space of every problem instance independently using SMAC [111]. We observe that up to 89% cost reduction can be obtained by searching for a parameters configuration that makes the branch-and-bound algorithm more efficient for the given problem instance. Unfortunately, this search is time-consuming and cannot be performed online for every new problem instance. Therefore, there is a need for methods to configure solvers on-the-fly while maintaining the expected cost of using a configuration tuned per instance.

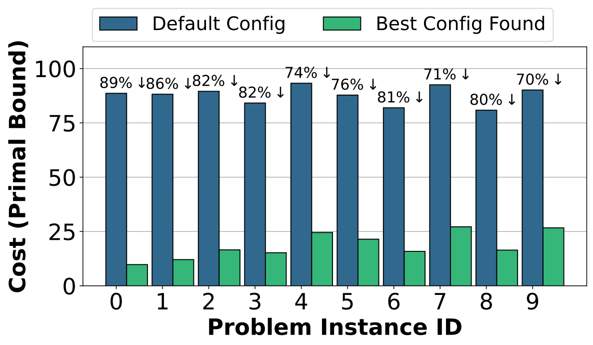
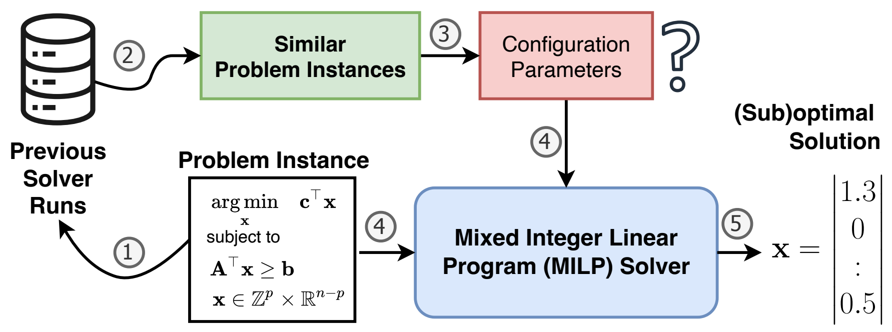


Figure 4.2: A significant cost reduction (up to 89%) can be achieved by searching and evaluating the configuration space of every problem instance independently. However, searching-and-evaluating takes 15 minutes (time-limit) for each evaluated configuration to complete (e.g., 10 evaluations is 150 minutes). Data shown is obtained using SMAC [111] on problem instances from the ML4CO dataset [129].

4.2 Motivation

Recently, machine learning (ML) has shown promising results for solving MILP problems [21, 32]. The motivation behind applying machine learning is to capture redundant patterns and characteristics in problems that are being solved repeatedly. Researchers have been able to achieve promising results by either integrating models within the solver's branch-and-bound loop [57, 108, 179, 88] or replacing the solver with an end-to-end algorithm that takes the raw problem instance as input and directly or iteratively output a feasible solution [89, 175, 20, 97]. Learning to configure solvers has also been explored early in [78, 185, 117]. The idea is to make a solver configuration instance-specific. In that direction, a problem instance is represented as a vector of hand-engineered features and similar instances are clustered together based on their vector representation. Then, various sets of configurations are evaluated and assigned to each cluster. The limitation of these works has been that features are designed rather than learned and instances within a single cluster might not, in fact, be correlated to their final solutions' costs. Nonetheless, this direction has opened the door for instance-specific solver configuration. More recently, meta learning on MILP has seen growing interest [100, 31]. Toward learning MILP representations for solver configuration, Valentin *et al.* [172] have proposed a supervised learning approach to predict a configuration for a specific problem instance amongst a finite set of configurations. However, this approach requires manually labeled data and is limited to the set of candidate configurations chosen a priori for training (< 60). In other words, supervised learning restricts the ability to explore the broader configuration space once a model is trained and deployed.

In this work, we address the gap in existing approaches by: (1) learning representative MILP similarities that correlate with the final solutions' costs, and (2) using a learning method that does not restrict the number of configurations to select from. We pursue these endeavors in a novel way through two contributions. First, we learn an embedding space for MILP instances using *Deep Metric Learning* [101]. Deep metric learning is a subfield of machine learning that focuses



Similar Problem Instances
Configuration Parameters
?
Previous Solver Runs
Problem Instance
$\arg\min_{\mathbf{x}} \ \mathbf{c}^\top \mathbf{x}$
subject to
$\mathbf{A}^\top \mathbf{x} \geq \mathbf{b}$
$\mathbf{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}$
Mixed Integer Linear Program (MILP) Solver
(Sub)optimal Solution
$\mathbf{x} = \begin{vmatrix} 1.3 \\ 0 \\ \vdots \\ 0.5 \end{vmatrix}$
1
2
3
4
4
5

Figure 4.3: A high-level pipeline of our proposed method. The goal is to select config. parameters based on similarity with previously solved problem instances.

on learning a distance function between input data points using deep neural networks. The goal is to create a meaningful representation in which similar data points are mapped close together, and dissimilar data points are mapped further apart, enabling more effective clustering, classification, or retrieval tasks. Using deep metric learning, we learn a representative embedding function for MILP, where problem instances with similar costs are closer to each other. Unlike existing instance-aware approaches, instances' features are not hand-engineered, but learned based on *Graph Convolutional Networks* [92], that allows our model to capture the relationships between decision variables and constraints. Second, we predict a parameters configuration for new problem instances using nearest neighbor search on the learned metric space, which does not limit the number of configurations to predict from. Our method is summarized in Figure 4.3. The goal is to select configuration parameters based on a new problem instance's similarity with previously solved problem instances from the same distribution. Same distribution instances are problem instances that share similar number of decision variables and constraints, and define a given optimization problem that is being solved repeatedly.

We show that our predictions correlate with the final solution's cost. In other words, finding a closer instance in the learned metric space and using its well-performing configuration parameters would ultimately improve the solver's performance on the new unseen instance. We evaluate our approach on real-world benchmarks from the ML4CO competition dataset [129] using SCIP solver [55], and compare against both using an incumbent configuration from SMAC [111], and predicted configurations from existing instance-aware methods. Our method solves more instances with lower costs than the baselines and achieves up to 38% improvement in the cost.

4.3 Related Work

Machine Learning for Combinatorial Optimization. Learning-based optimization methods have seen growing interest lately [21, 32]. Broadly speaking, they can be divided into methods inside the solvers [88, 57, 108, 179], methods outside the solvers [100, 31], and methods that replace the solvers [89, 175, 20, 97]. Our work is amongst methods outside the solver, which aims at improving the solver's performance by instantly predicting instance-aware parameters configuration. This is orthogonal to existing work and can benefit from existing hyper-parameter search methods when performed offline.

Instance-aware Solver Configuration. Instance-aware configuration methods have been explored early in ISAC [78], which stands for Instance-Specific Algorithm Configuration. The method extracts features from problem instances and assigns problem instances with similar feature vectors to a cluster using g-means clustering. Features include problem size, proportion of different variable types (e.g., discrete vs continuous), constraint types, coefficients of the objective function, the linear constraint matrix and the right hand side of the constraints. After that, assuming that problem instances with similar features behave similarly under the same configuration, local search is used to find good parameters for each cluster of instances. Although this approach allows us to bypass the expensive search-and-evaluate at deployment time, features are hand-engineered and need to be adapted for each problem, e.g., as in [13]. In other words, the algorithm requires further refining of the distance metric in the feature space so that it can find better clusters. Hydra-MIP [185] enhanced this approach by including features from short solver runs before selecting a configuration for a complete solver run. It also uses pair-wise decision forests to select amongst candidate configuration parameters. Our approach is different since problem features are *learned* during training, and correlates similarity to the costs of final solutions. Moreover, these approaches assign a single parameters configuration for each cluster, which limits the portfolio of configurations available at inference time. More recently, supervised deep learning was investigated in [172]. The method selects a limited number of configuration parameter sets, and collects training data by running the solver using the selected configurations on all problem instances separately. Using the labeled data, it can predict the cost of running the solver on a new unseen instance using one of the configurations used during training. Aside from the massive labeled data required for training, this approach limits the potential of exploring other sets of configurations after the model is trained and deployed. Exploring further solver configurations would require solving and labeling more problem instances, then re-training the model.

Shallow Embedding vs. Deep Embedding. Instance-aware configuration methods represent a MILP instance as a vector of values that encapsulate the primary characteristics of the problem instance. The objective of a specific embedding (i.e., encoding) method is to ensure that similarity in the embedding space (e.g., dot product) closely mirrors the similarity found in the original problem representation. The effectiveness of an embedding method is determined by its ability to uniquely identify and distinguish between problem instances that may share similar properties (such as the number of decision variables and constraints) but exhibit differences in their solutions' costs within the same solving environment. A powerful embedding method can accurately differentiate between such instances, enabling more effective and tailored configuration selection for each problem instance. *Shallow* embedding is the simplest encoding approach, where the encoder is just an embedding lookup. For example, in ISAC [78], problem instances are encoded as feature vectors for the Set Covering problem that include a normalized cost vector $\mathbf{c}$, bag densities, item costs and coverings, in addition to other density functions. These values are aggregated (using minimum, maximum, average and standard deviation) to construct the final feature embedding of the problem instance. Similarly, authors in [13] use ISAC's method and focuses on the maximum satisfiability problem (MaxSAT), with hand-engineered features that include problem size, balance features and local search features. Hydra-MIP [185] extracts more features by executing short runs of the solver (CPLEX) using a default configuration on each new instance. These features include pre-solving statistics, cutting planes usage, and the branch-and-bound tree information. While shallow embedding is straightforward to compute, these encoders are non-injective [184]. That is, different MILP instances could have the same embedding using a shallow feature vector. Moreover, and by design, shallow embeddings are not necessarily correlated with the final costs of the solver's solutions.

In contrast, *Deep* encoders learn the embedding function during training according to a defined loss function. In other words, a deep encoder is characterized by learnable parameters of a deep neural network that defines embedding similarity based on a loss function. In our method, we train a Graph Convolutional Network (GCN) to embed MILP instances to an embedding space where the similarity of instances is defined based on their final solutions' costs in the same solving environment. Deep embedding is injective and uniquely encodes problem instances even if they have the same shallow embedding (e.g., number of decision variables). In Section 4.7, we show that when the size of the problem remains relatively similar, but the coefficients or structure vary, deep embedding has a larger discriminative power over shallow embedding. In problems where the problem size varies significantly, shallow embedding could be enough. In designing a system that is invariant to the problem size, deep embedding addresses the need without hand-engineering features for each problem separately.

Configuration Space Search. During the process of identifying similar problem instances, addressing the selection of parameter configurations remains a challenge. In situations where ample time is available for exploration, such as testing different parameter configurations on a single problem instance, several methods have been proposed to navigate this vast search space. These methods aim to identify a single robust configuration[1] across a collection of problem instances, denoted as $\mathcal{I}$. Random search [23], evolutionary algorithms [140], Bandit methods [106], and Bayesian-based optimization [159] are among the top-performing methods that have been applied successfully in various optimization contexts [113, 27, 28, 123, 67, 87].

The SMAC package [111] is an instance of model-based optimization that employs Bayesian optimization for searching parameters configuration [69]. The central concept of SMAC revolves around building a probabilistic model, specifically a random forest model, which predicts the performance of an algorithm on a set of instances, given a specific configuration. By sequentially updating this model based on the observed performance of algorithm configurations, SMAC is able to efficiently search for an optimal or near-optimal configuration within a pre-defined search space. The key components of SMAC include the acquisition function [77], which guides the search process in terms

[1]Also called incumbent configuration in the context of parameters configuration search; not to be confused with the incumbent solution of the solver itself, which is the $\mathbf{x}$'s assignment with minimum cost of the MILP objective function.

of exploration and exploitation trade-off, and the intensification procedure [106], responsible for selecting a new incumbent configuration. The most common acquisition function used in SMAC is the Expected Improvement (EI) function [165, 70], which aims to minimize the expected runtime of the target algorithm.

Another popular package for algorithm configuration, the irace package [113] is based on the Iterated Racing framework, which is a derivative of the F-race procedure [29, 15]. The main idea behind irace is to iteratively sample and compare algorithm configurations on an increasing set of problem instances, using statistical tests to eliminate poorly performing candidates. This iterative process continues until a termination criterion is met, usually when a maximum number of iterations or a maximum time is reached. The irace package is particularly well-suited for discrete and categorical parameter spaces, as it does not require any explicit modeling of the performance landscape. The search process is guided by a combination of adaptive sampling and statistical tests, which provide a balance between exploration and exploitation. The elimination of underperforming configurations is carried out using a statistical test, most commonly the Friedman test or the two-sample t-test, which considers the performance of the remaining configurations. The key differences between both packages is that SMAC adopts a model-based optimization strategy with Bayesian optimization, building a surrogate model to predict algorithm performance, while irace is a model-free approach relying on iterative sampling and statistical tests to identify the best-performing configurations.

In this work, our method of selecting parameters configurations based on problem instance similarity is agnostic to the package used for the offline configuration search phase. We use SMAC for its interoperability with the SCIP solver [55] through its Python binding [115] along with the PyTorch Ecosystem [136] used for training the deep metric learning model. Nonetheless, after the model training phase is completed, which entails learning the similarity, the system illustrated in Figure 4.6 can be adapted to incorporate the irace package for an additional offline search of the configuration space of previously solved problem instances. Our approach eliminates the necessity to retrain the previously acquired similarity models.

Predicting Solver Configuration. Aside from using shallow or deep embedding, predicting a solver configuration for an unseen instance requires selecting an already-evaluated configuration from similar instances in the embedding space. ISAC [78] and MaxSAT [13] use G-means clustering to cluster similar instances. Then, they assign a single configuration to each cluster to be used for new instances that are embedded into that cluster. This approach evolves by refining the distance metric in the feature space so that it can find better clusters in future iterations. Hydra-MIP [185] uses pair-wise weighted random forests (RFs) to select amongst m algorithms for solving the instance, by building $m(m-1)/2$ RFs and taking a weighted vote. When the number of parameters configuration to select from is large (i.e., large m), calculating pair-wise RFs becomes computationally infeasible. In our method, we use k-nearest neighbor (KNN) to predict a parameter configuration from the closest problem instance in the learned embedding space. This allows our approach to scale the exploration of configuration parameters without the restriction of refining clusters, or re-building a limited number of pair-wise RFs. In both Hydra-MIP [185] and our method, k configurations

Table 4.1: Summary of instance-aware solver configuration methods. Details of the methods are described in Section 4.3.

	ISAC [78, 13]	**Hydra-MIP** [185]	**Our Method**
Features	Hand-crafted	Hand-crafted	Learned
Embedding	Shallow	Shallow	Deep
Injectivity	Non-injective	Non-injective	Injective
Offline Search	Genetic Algorithm	Regression/Iterative	Bayesian Search
Inference	G-means Clustering	Decision Forests	KNN
#Configs Predicted	1	k (hyperparameter)	k (hyperparameter)

can be predicted at once (with ranks) to potentially run the solver using multiple configurations in parallel. Table 4.1 summarizes the differences between our method and existing instance-aware solver configuration methods.

4.4 Preliminaries

4.4.1 MILP Formulation

In this work, we consider MILP instances formulated as:

$$\arg\min_{\mathbf{x}} \quad \mathbf{c}^\top \mathbf{x}, \qquad \text{subject to} \quad \mathbf{A}^\top \mathbf{x} \geq \mathbf{b}, \qquad \text{and} \quad \mathbf{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \tag{4.1}$$

where $\mathbf{c} \in \mathbb{R}^n$ denotes the coefficients of the linear objective, $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$ denote the coefficients and upper bounds of the linear constraints, respectively. n is the total number of decision variables, $p \leq n$ is the number of integer-constrained variables, and m is the number of linear constraints. The goal is to find feasible assignments for $\mathbf{x}$ that minimize the objective $\mathbf{c}^\top \mathbf{x}$. A MILP solver constructs a search tree to find feasible solutions with minimum costs. The cost of the solution found by the solver by the end of its search, or if the time limit is reached, is called the primal bound. It serves as an upper bound to the set of feasible solutions. While there are other methods to measure the solver's performance, (e.g., dual bound, primal-dual gap, primal-dual integral [4]), we adopt the primal bound at the end of the time limit for the purpose of training the metric learning model.

4.4.2 Graph Neural Networks

Graph Neural Networks (GNNs) offer a powerful paradigm for analyzing complex relational data, which is often encountered in Operations Research problems [176, 104]. GNNs are designed to learn meaningful representations of nodes in a graph by incorporating both node features, edge features and graph structure. The core principle behind GNNs is message-passing, where information is aggregated from neighboring nodes to update the representations iteratively.

The message-passing framework for GNNs can be formalized as follows. Let $G = (V, E)$ denote a graph with nodes V and edges E. Each node $v_i \in V$ is associated with a feature vector x_i. The goal is to learn a representation h_i for each node v_i. The message-passing process in GNNs typically consists of L layers, where each layer l updates the node representations based on the previous layer's representations. The update at each layer can be expressed as:

$$m_i^{(l)} = \sum_{j \in \mathcal{N}(v_i)} M^{(l)}(h_j^{(l-1)}, h_i^{(l-1)} e_{ji}), \tag{4.2}$$

$$h_i^{(l)} = U^{(l)}(h_i^{(l-1)}, m_i^{(l)}) \tag{4.3}$$

where $\mathcal{N}(v_i)$ denotes the set of neighboring nodes of v_i, $M^{(l)}$ is a message function that computes the messages $m_i^{(l)}$ to be sent from node j to node i at layer l, $U^{(l)}$ is an update function that computes the new node representation $h_i^{(l)}$ using the aggregated messages, and e_{ji} represents the edge features, if present. Both $M^{(l)}$ and $U^{(l)}$ are typically implemented as neural networks, allowing GNNs to learn complex, nonlinear relationships between nodes. In this work, we use a GNN from [57] to model the relationships between decision variables and constraints in Equation 4.1.

4.4.3 Metric Learning

Deep learning models require a vast amount of data in order to make reliable predictions. In a supervised learning setting, the goal is to map inputs to labels as in a standard classification or regression problem. When the number of classes is huge, supervised learning fails to address real-world applications. For example, face verification systems have a large number of classes, but the number of examples per class is small or non-existent [154]. In this case, the goal is to develop a model that learns object categories from a few training examples. But deep learning models do not work well with a small number of data points. In order to address this issue, we learn a similarity function between data points, which helps us to predict object categories given small data for training. This paradigm is known as metric learning [101]. In this paradigm, a model is trained to learn a distance function (or similarity function) over the inputs themselves. Here, similarity is subjective, so the distance may have a different meaning depending on the data. In other words, the model learns relationships in the training data regardless of what it actually means in its application domain. Metric learning has seen growing adoption in real-world applications, such as face verification [154, 178, 45], video understanding [103] and text analysis [43].

Measuring distances is a critical aspect of metric learning. Given two instances of some object representation, $\mathcal{I}_i$ and $\mathcal{I}_j$, a distance function, d, measures how far the two instances are from each other. The Euclidean distance is challenging to reason about in higher dimensions even if the data is perfectly isotropic and features are independent from each other. Therefore, the goal is to define new distance metrics in higher dimensional spaces that are based on the properties of the data itself. These are non-isotropic distances reflecting some intrinsic structures of the data. A parametric model is trained to project instances to the new metric space through either a linear transformation of the data such as the Mahalanobis distance [44], or a non-linear transformation of the data using deep learning [84]. This projection step allows the Euclidean distance to capture relationships between the features that are non-linear or more complex; in our case, the correlation between a problem instance structure and its solution's cost on a given solving environment.

In metric learning, instead of requiring labels for training, the model requires weak supervision at the instance level, where triplets of (anchor a, positive p, negative n) are fed into the model. The model is trained to learn a distance metric that puts positive instances close to the anchor and negative instances far from the anchor. This is achieved by a Triplet loss function [154]:

$$L = \sum_{i}^{N} [||f(\mathcal{I}_i^a) - f(\mathcal{I}_i^p)||^2 - ||f(\mathcal{I}_i^a) - f(\mathcal{I}_i^n)||^2 + \alpha]_+ \tag{4.4}$$

where N is the number of triplets sampled during training. $\mathcal{I}^a$, $\mathcal{I}^p$ and $\mathcal{I}^n$ represent the anchor instance, similar instance and dissimilar instance, respectively. f is a parametric model that projects instances to a learned metric space. The loss increases when the first squared distance (anchor-positive) is larger than the second squared distance (anchor-negative). So, f is trained to decrease this loss. In other words, it tries to make the first squared distance smaller, and the second square distance larger. Here, the loss, L, will be equal to zero if the first squared distance is α-less than the second squared distance. While there are other variants of the loss functions for metric learning, e.g. Contrastive Loss [94], Triplet loss can provide more stable training compared to contrastive loss, as it considers both positive and negative examples simultaneously for each anchor point [177]. In addition, Triplet loss focuses on ensuring that the distance between the anchor and positive example is smaller than the distance between the anchor and negative example, by a margin. This allows the model to learn a more balanced similarity metric, especially for complex structures such as MILP. In Section 4.6, we present a number of modifications during training in order to avoid having a zero-loss early during training.

4.5 Data Validation

The fundamental motivation of our work is to define similarity among MILP instances based on their final solutions' costs after running the solver in the same environment (i.e., host machine, software environment, configuration parameters, time limit, and random seed), and under the assumption that all MILP instances are coming from the same problem distribution. Same distribution instances are problem instances that share similar number of decision variables and constraints, and define a problem that is being solved repeatedly. To our knowledge, no prior work has explored correlating MILP similarity to the costs (objective function) of their final solutions.

First, we validate the assumption that MILP instances which have similar costs when solved in a specific environment would have similar costs when changing the solver configuration. For this validation, we use the Item Placement benchmark from the ML4CO dataset [129]. We run the solver on 25% of the training dataset (i.e., 2500 MILP instances) using the default solver configuration

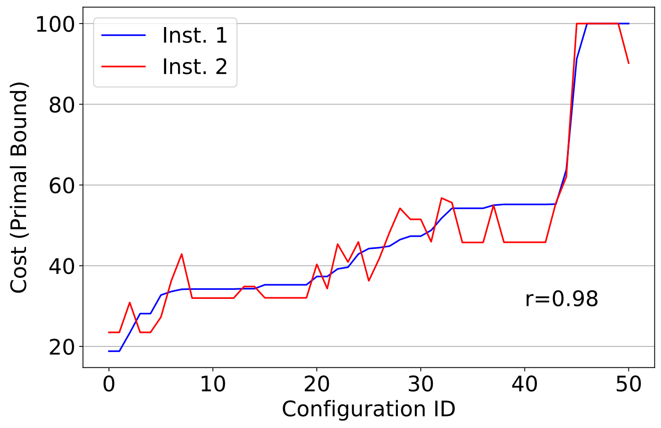
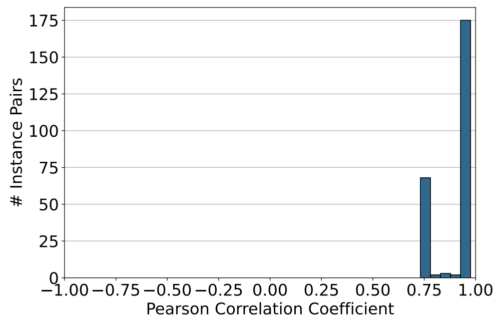
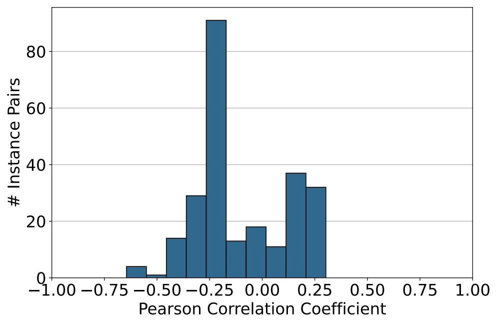


Figure 4.4: Two MILP instances that have similar costs when using the same default configuration have similar costs when using other configurations. Other environment variables are fixed: solver version, machine (cpu and memory) and random seed. r is the Pearson correlation coefficient.

and a time limit of 15 minutes. Each solver run is executed independently and is given the same compute and memory resources. In Figure 4.4, we select two MILP instances that have similar costs ($|C(\mathcal{I}_1) - C(\mathcal{I}_2)| \leq C_{thr}$) when using the default solver configuration, and solve both of them independently using different sets of configurations while still fixing all other hyper-parameters (i.e., cpu, memory, solver version, time limit and random seed). Here, $C_{thr} = 1$, and the $C \in [0, 100]$. We observe that the costs of the two instances are indeed positively correlated with a Pearson correlation coefficient of $r = 0.98$. We extend this investigation to validate if this is the case for other pairs of similar and dissimilar MILP instances in the collected dataset. So, we select 250 MILP instances (10%) that are similar in their costs, and another 250 MILP instances (10%) that are largely dissimilar in their costs. We run each pair of instances independently using eight other solver configurations and report their final solutions' costs. Figure 4.5 shows a histogram of the Pearson correlation coefficient for similar and dissimilar pairs of instances. We observe that similar pairs of instances have a Pearson correlation coefficient > 0.75, which indicates a high positive correlation, while dissimilar pairs of instances either have a small correlation coefficient < 0.3, or a negative coefficient indicating an inverse correlation. This finding confirms that if we are able to define MILP similarity based on their final solutions' costs (unlike [78, 185] that define similarity based on hand-crafted features without correlation to the final solutions' costs), we will be able to predict an effective parameters configuration for the solver for a new unseen MILP instance by fetching a previously-evaluated configuration from a similar instance.



(a) Similar instance pairs $|C(\mathcal{I}_i) - C(\mathcal{I}_j)| \leq C_{thr}$



(b) Dissimilar instances pairs $|C(\mathcal{I}_i) - C(\mathcal{I}_j)| \gg C_{thr}$

Figure 4.5: Correlation between problem instances and their costs at different parameters configurations. (a) Pairs of instances with similar costs when using the default configuration are solved simultaneously using other configurations and their cost correlations are measured using Pearson correlation coefficient. (b) Likewise, pairs of instances with large cost difference when using the default configuration are solved simultaneously using other configurations and their cost correlations are measured. Problem instance pairs were retrieved from the Item Placement dataset [129] where each instance has 185 decision variables and 1083 constraints. The solver is run on 250 random similar pairs and 250 random dissimilar pairs using eight different configurations for each run. Here, $C_{thr} = 1$, and the $C \in [0, 100]$.

4.6 Method

In order to define MILP similarity based on the final solutions' costs, our approach is to use *Deep Metric Learning* to learn the instance embeddings, and based on that predict instance-aware parameters configuration. Figure 4.6 shows an overview of our methodology. In contrast to supervised learning where a large amount of data needs to be collected in order to train the model, we collect training data on a small subset of the problem instances available. The method is divided into two major parts: (1) a training phase to learn MILP similarities based on costs, and (2) an inference phase to predict a parameters configuration for a new MILP instance. We present the details for each phase in Subsections 4.6.1 and 4.6.2, respectively.

4.6.1 Learning MILP Similarity

In the training phase (5.1) of Figure 4.6, and given two MILP instances, $\mathcal{I}_i$ and $\mathcal{I}_j$, the goal is to train a parametric model that recognizes whether $\mathcal{I}_i$ and $\mathcal{I}_j$ are similar or not. As discussed in Section 4.4, similarity is subjective and depends on the domain. In our case, there is no natural way to find out whether two instances are similar or not just from their given problem formulation (Equation 4.1). Even though one could map it to a graph isomorphism problem, small perturbations of $\mathbf{A}$ can lead to different solutions from the solver. For example, a slight change in a constraint's coefficients could make the constraint trivial, or make the MILP instance infeasible [172].

We divide the training stage into four main steps. On one hand (5.1.1), we sample MILP instances from the training set based on their final solutions costs. On the other hand (5.1.4), we define our

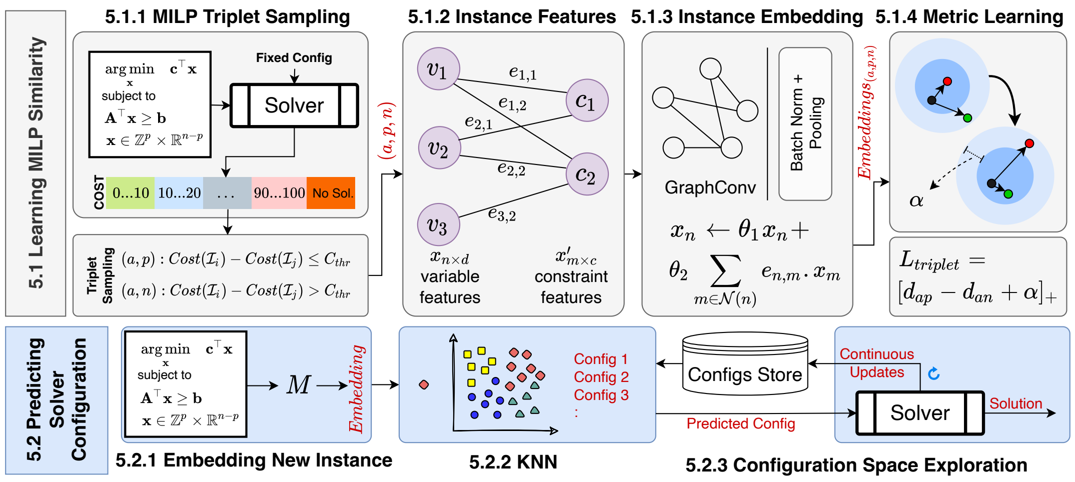

5.1 Learning MILP Similarity
5.1.1 MILP Triplet Sampling
5.1.2 Instance Features
5.1.3 Instance Embedding
5.1.4 Metric Learning
$\arg\min_x \mathbf{c}^\top \mathbf{x}$ subject to $\mathbf{A}^\top \mathbf{x} \geq \mathbf{b}$, $\mathbf{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}$
Fixed Config
Solver
COST
0...10
10...20
...
90...100
No Sol.
Triplet Sampling
$(a,p): Cost(\mathcal{I}_i) - Cost(\mathcal{I}_j) \leq C_{thr}$
$(a,n): Cost(\mathcal{I}_i) - Cost(\mathcal{I}_j) > C_{thr}$
(a,p,n)
v_1 v_2 v_3 c_1 c_2
$e_{1,1}$ $e_{1,2}$ $e_{2,1}$ $e_{2,2}$ $e_{3,2}$
$x_{n\times d}$ variable features
$x'_{m\times c}$ constraint features
GraphConv
Batch Norm + Pooling
$x_n \leftarrow \theta_1 x_n + \theta_2 \sum_{m\in\mathcal{N}(n)} e_{n,m} \cdot x_m$
$Embeddings_{(a,p,n)}$
α
$L_{triplet} = [d_{ap} - d_{an} + \alpha]_+$
5.2 Predicting Solver Configuration
M
$Embedding$
5.2.1 Embedding New Instance
Config 1
Config 2
Config 3
5.2.2 KNN
Configs Store
Continuous Updates
Predicted Config
Solver
Solution
5.2.3 Configuration Space Exploration

Figure 4.6: Overview of our method. Triplet samples are first collected on a few instances using the default solver configuration. Instance features are extracted as a bipartite graph [57], then embedded using a graph convolutional network. A triplet loss [154] function is used to train the model end-to-end. C_{thr}: cost threshold for similarity, a: anchor instance, p: positive/similar instance, n: negative/dissimilar instance, θ: learnable parameters of the GNN, d: distance between embeddings, as defined in Equation 4.4. M refers to the trained GNN model. KNN refers to using k-nearest neighbors to find a similar instance in the learned embedding space. The Config Store is a continuously-updated data store to expand the portfolio of explored configurations after model deployment.

loss function with the goal of bringing the learned embeddings of similar instances closer to each other, and dissimilar instances further from each other. In between (5.1.2 and 5.1.3) lies a Graph Convolutional Network (GCN) model that extracts features from problem instances and passes them through convolutional layers of learnable parameters that reduce the loss during training.

MILP Triplet Sampling

In our method, if the difference between the solution cost of instance $\mathcal{I}_i$ and $\mathcal{I}_j$ is below a certain threshold C_{thr}, then $\mathcal{I}_i$ and $\mathcal{I}_j$ are considered similar for the purpose of training the model. If the cost difference is above C_{thr}, the instances are considered dissimilar. In the triplet sampling step, the goal is to look up for similar and dissimilar instances in the training dataset. Algorithm 2 shows the steps for the mining and training procedures. In line 1, we sample an arbitrary anchor instance (a). In line 2, we sample a similar instance (p) where the difference in their costs under the default solver configuration is less than a threshold. In our work, we introduce a new sampling schedule for the training procedure. The goal is to avoid crunching the loss (Equation 4.4) to zero prematurely. Therefore, in line 3, we start with hard negative sampling by looking for instances that have a cost difference much larger than the threshold. The idea is that when starting with these negative pairs $(a,\ n)$, the model gets a chance to be able to push their embeddings further away from each other when training for a certain number of epochs (line 4). Then, in lines 5-6, this restriction is relaxed

Algorithm 2: MILP Triplet Sampling

```
1 Input: Training dataset of MILP instances ($\mathcal{I}$)
2 Input: Costs using default solver configuration ($C_i$)
3 Input: Cost threshold ($C_{thr}$)
4 Output: Triplets ($a$, $p$, $n$)
    1: Select an arbitrary anchor instance ($a$) from $\mathcal{I}$.
    2: Find a similar problem instance ($p$) such that $|C(\mathcal{I}_a) - C(\mathcal{I}_p)| < C_{thr}$
    3: Find a dissimilar problem instance ($n$) such that $|C(\mathcal{I}_a) - C(\mathcal{I}_n)| \gg C_{thr}$
    4: Train GNN model parameters for $e_1$ epochs.
    5: Find a dissimilar problem instance ($n$) such that $|C(\mathcal{I}_a) - C(\mathcal{I}_n)| > C_{thr}$
    6: Continue training GNN model parameters for $e_2$ epochs.
       Using Loss: $L_{triplet} = [d_{ap} - d_{an} + \alpha]_+$
```

and the training loop starts seeing negative instances that have slightly larger cost difference than positive instances. Theoretically, triplet sampling can be done using any other defined measure of similarity. We chose to use the cost after running the solver in order to correlate similarity with the final solutions costs.

Feature Extraction

The MILP formulation represented in Section 4.4 does not restrict the order of the decision variables in the objective, nor the number and order of the constraints. Therefore, a feature extractor needs to be invariant to their order to handle instances of varying sizes. In Step 5.1.2 of Figure 4.6, we represent a MILP instance using the bi-partite graph representation from [57]. Each decision variable is represented as a node, and each constraint is also represented as a node. An undirected edge between a decision variable, v_i, and a constraint, a_j, exists if v_i appears in a_j, that is if $\mathbf{A}_{ij} \neq 0$. Variable nodes have features represented as the variable type (binary, integer or continuous) in addition to its lower and upper bounds. They are represented as $X \in \mathbb{R}^{n \times d}$, where n is the number of nodes and d is the features dimension. Constraint nodes have features represented in their (in)equality symbol ($<$, $>$, $=$). They are represented as $X' \in \mathbb{R}^{m \times a}$, where m is the number of constraints and a is the features dimension. Edge features represent the coefficients of a decision variable appearing in a constraint, $E \in \mathbb{R}^{n \times m \times e}$, where e is the number of edges. These features are extracted once before the solver starts the branch-and-bound procedure, namely at the root node. Therefore, each problem instance has a single graph structure representation before any cuts happen at the root node (part of the heuristics-based algorithms). While the original representation in [57] has additional features, we only extract the features of the problem instance, and not the solver's state.

Instance Embedding

In Step 5.1.3 of Figure 4.6, we parameterize our distance metric model using a GCN model [92]. The network structure has four convolutional layers, and the convolutional operator is implemented as defined in [135]. The network parameters, θ_1 and θ_2, are updated within the end-to-end training procedure where features of the decision variables are updated as: $x_n \leftarrow \theta_1 x_n + \theta_2 \sum_{m \in \mathcal{N}(n)} e_{n,m} . x'_m$. Similarly, the features of the constraints are updated as $x'_m \leftarrow \theta_1 x'_m + \theta_2 \sum_{n \in \mathcal{N}(m)} e_{m,n} . x_n$. Graph embeddings are then passed through batch normalization, max-pooling and attention pooling layers to produce a latent vector which is used for the downstream metric learning loss.

Model Training

In Step 5.1.4 of Figure 4.6, the model is trained end-to-end using the loss function defined in Equation 4.4. The distance function used is the Euclidean distance on the learned metric space. Remember that the Euclidean distance tends to underperform when calculated on high-dimensional data. However, the non-linear step introduced by the graph neural network enables it to capture relationships between the features of the problem instances that are consistent with their correlation to the final solution costs. In essence, the projection of the problem instance into a learned space allows the Euclidean distance metric to overcome biased outcomes.

The training proceeds for a number of predefined epochs, while ensuring that the loss does not fall to zero by adopting the proposed triplet sampling schedule in Algorithm 2. The larger the value of α, the further positive instances are pushed away from negative ones. However, choosing a large value of α will make the model set the value of the distance function d as zero. Thus, α should be tuned for the training procedure.

4.6.2 Predicting Configuration Parameters

In the inference phase (5.2) of Figure 4.6, the solver is invoked to solve a MILP instance using a given configuration (or default if none is provided). The goal of this phase is to allow a real-world solver deployment to continue to autonomously improve over time as more configuration parameters are explored. Thus, we propose a closed-loop system where solutions from real-world problems are continuously saved for future evaluation and use.

Embedding New Instances

As motivated earlier in Section 4.2, we find effective configurations by using a configuration from similar instances in the learned metric space. Therefore, the first step is to embed (i.e., encode) the new problem instance using the learned model (M) from the training phase. The embedding time is negligible compared to the solving time as it takes a few milliseconds to extract MILP features and run them through the small GCN. One advantage of adopting a deep embedding method in our approach is that it is inductive [64], and can generate embeddings for MILP instances of different sizes (i.e., number of decision variables or constraints). In other words, it does not require re-training the model to accommodate new instances seen in a real deployment.

Algorithm 3: Predicting Solver Configuration

```
1 Input: Unseen MILP instance ($\mathcal{I}$)
2 Input: Trained embedding model ($M$)
3 Parameters: # of nearest neighbors ($k$), # of predicted configurations ($n$)
4 Output: Predicted solver configuration
    1: Embed instance $\mathcal{I}$ using $M$.
    2: Retrieve previously-solved $k$ nearest neighbors.
    3: Select $n$ previously-explored configurations from each retrieved neighbor in a non-descending
       order according to their associated costs.
    4: Return the configuration with the lowest cost.
```

Nearest Neighbor Instances

A trained model is a model capable of measuring a distance metric between MILP instances. The final embeddings of the instances are saved in a central store to be used in the prediction step. Algorithm 3 gives the steps performed for predicting a parameters configuration for a new unseen MILP instance. In Step 1, the problem instance is first embedded using the trained model. In Step 2, we perform a nearest neighbor search on the learned metric space. We introduce two tuning parameters for the prediction: (1) k, representing the number of nearest neighbors we want to fetch, and (2) n, representing the number of configurations for each neighbor, sorted in a non-descending order by their solutions' costs. In Step 3, we retrieve n previously-explored configurations for each of the k neighbors. Then, in Step 4, we predict a parameters configuration as the one with the minimum cost. If $k = 1$ and $n = 1$, then the algorithm predicts the lowest cost configuration parameters of the nearest neighbor. In multi-core environments (e.g., cloud), a practitioner may choose to run the solver in parallel using different configuration parameters and gather an ensemble of solutions for the new problem instance. In this case, k and n can be exposed as hyperparameters for the prediction model.

Configuration Space Exploration

As mentioned in Section 4.3, during the process of identifying similar problem instances, selecting an appropriate parameter configuration remains a challenge. Essentially, when adequate time is available for exploration, navigate a vast search space requires a an exploration strategy for configuration parameters that are most likely to yield good results. Given the huge number of potential solver configurations, we term this issue as the exploration problem. In our approach, we provide initial configurations to the problem instances used for similarity lookup by independently searching the configuration space of each instance with SMAC [111]. The primary goal of SMAC is to find an optimal set of configuration parameters for a given algorithm to minimize a specific performance metric (e.g., MILP objective cost in our context) within a user-defined search space of possible configurations. SMAC is based on a Bayesian optimization framework that utilizes surrogate models, such as Gaussian Process Regression or Random Forests, to model the objective function. It employs an acquisition function, such as Expected Improvement (EI), to balance exploration and exploitation

Table 4.2: Dataset Statistics

	# Decision Variables			# Constraints		
Benchmark	Count	Avg.	Median	Count	Avg.	Median
Item Placement	195	195	195	1,083	1,083	1,083
Load Balancing	61,000	61,000	61,000	64,081–64,504	64,307	64,308
Anonymous	1,613–92,261	33,998	4,399	1,080–12,6621	43,373	2,599

during the search process. SMAC iteratively refines its surrogate model by querying new points in the configuration space space and updating the model with their corresponding objective function values. This step is performed offline, separate from the training and inference loops.

However, once the model is deployed in a real-world setting, we enable it to evolve by incorporating a feedback loop in which a solver saves its results to the data store. Each data point consists of a problem instance's embedding, the configuration employed for solving, and the cost obtained from the solver. Future lookups using KNN can immediately benefit from the newly added data point without retraining the model since similarity is based on the already-learned embeddings. This design choice allows our method to be deployed in real-world environments without requiring frequent model retraining. For the implementation details of the data store, refer to Appendix A.

4.7 Empirical Results

4.7.1 Dataset

We used the publicly available dataset from the ML4CO competition [129]. The dataset consists of three problem benchmarks. The first two problem benchmarks (item placement and load balancing) are extracted from applications of large-scale systems at Google, while the third benchmark is extracted from MIRPLIB – a library of maritime inventory routing problems[2]. The item placement and load balancing benchmarks contain 10,000 MILP instances for training (9,900) and testing (100), while the anonymous problem contains only 118 instances (98 and 20 for training and testing, respectively). The dataset is available to download from the ML4CO competition website[3] with a full description on the problems formulation and their sources. The smallest of these problems are extremely hard to solve to optimality. For example, after 48 hours of solving time using SCIP, an instance of the Item Placement dataset was not solved to optimality on a high-end machine (Section 4.7.2). In fact, after 2 hours, the solver reports a gap of 22.00% and a search progress completion of 23.05%. After 12 hours, the solver reports a gap of 14.00% and a search progress completion of 32.05%. After 48 hours, the solver reports a gap of 10.28% and a search progress completion of 35.60%. In this section, we show some statistics on the dataset and reflect on how they affect our approach of metric learning.

[2]Link: `https://mirplib.scl.gatech.edu/instances`

[3]Link: `https://github.com/ds4dm/ml4co-competition`

Table 4.2 shows the number of decision variables and constraints in each benchmark. All instances in the Item Placement benchmark have the same number of decision variables and constraints. The Load Balancing benchmark has the same number of decision variables, but the number of constraints varies within a small range. The Anonymous benchmark exhibits a large variance in both the number of decision variables and constraints. For a MILP solver, a high variance in the number of decision variables or constraints has a direct impact on its solution. It also affects the learned embeddings of these instances. While the high variance gives more discriminative power to the model (M), it does not directly serve the purpose of finding a configuration for new instances using the nearest neighbor. The reason is that the nearest neighbor might indeed not be close in distance in the learned metric space, and the predicted parameters configuration would not be directly correlated to the solver's solution. Therefore, it is critical that the definition of "same distribution" instances include the number of decision variables and constraints for the purpose of finding a parameters configuration using metric learning.

4.7.2 Experimental Setup

In this section, we provide details on our runtime environment, the data utilized for training, and the training methodology. Subsequently, we design a series of experiments to evaluate the effectiveness of our approach, both in terms of learning meaningful MILP embeddings and its influence on the final solution's cost when employing the complete system illustrated in Figure 4.6. First, in Section 4.7.3, we delve into the learned MILP embeddings and examine their correlation with the final solution costs when solved in the same environment. Next, in Section 4.7.4, we explore the precision of the predicted configurations in identifying suitable configuration parameters. Finally, in Section 4.7.5, we compare our method with existing approaches for selecting parameter configurations and discuss the implications of learning improved similarity models as they relate to the predicted costs after solving.

Runtime Environment

The experimental results are obtained using a machine with Intel Xeon E5-2680 2x14cores@2.4 GHz, 128GB RAM, and a Tesla P40 GPU. The model was developed using PyTorch (v1.11.0+cu113) [144], Pytorch Geometric (v2.0.4) [52], and PyTorch Metric Learning (v1.3.0) [136]. We used Ecole (v0.7.3) [147] for graph feature extraction, convolution operators modified and adopted from [172], PySCIPOpt (v3.5.0) [115] as the MILP solver, and SMAC3 (v1.2) [111] for the offline configuration space search.

MILP Triplet Sampling

Given the training dataset, we run the MILP solver on all instances using the default parameters configuration of the solver with a time limit of 15 minutes as suggested by [129]. The total number of solved instances by the end of the time limit were 2599, 1727 and 38 for the item placement, load balancing and anonymous benchmarks, respectively. This represents 26%, 17% and 38% of the training benchmarks, respectively. We implemented the triplet sampling schedule as discussed in Section 4.6, where hard negative sampling was used for the first 50 epochs, and the training continues for 100 epochs in total. We used a batch size of 256 for the item placement, 64 for load balancing, and the full 98 instances for the anonymous benchmark.

Model Training

The model consists of a graph neural network of four layers with 64 as the dimension of the hidden layers. It is trained for each benchmark separately in order for the triplet sampling and training to run on data coming from the same distribution. The output from the convolutional layers is passed into a batch normalization layer, followed by a max pooling layer and an attention pooling layer. The output embedding size is set to 256. We set $\alpha = 0.1$ in the loss function.

4.7.3 Instance Embedding

We visualize the instance embeddings of the GNN before and after model training and compare it to using shallow embeddings in Figure 4.7. The color bar represents the cost of the solution using the default configuration parameters. The shallow embedding vector encodes presolving statistics as in Hydra-MIP [185], which include the problem size, the minimum, maximum, average and standard deviation of the objective coefficients ($\mathbf{c}$) and the constraints coefficients ($\mathbf{A}$, $\mathbf{b}$). While Hydra-MIP's shallow embedding includes more features such as the cutting planes usage and the branch-and-bound tree information, such information is not available before running the solver[4]. From Figure 4.7, we observe that in the item placement benchmark, shallow embeddings do not offer any discriminative power to the problem instances. In the load balancing benchmark, shallow embeddings could indeed cluster problem instances, but clusters are not correlated with the final solver's costs. In the anonymous benchmark, instances with similar costs were clustered close to each other, which gives shallow embedding a discriminative power in this case. Analyzing this result in light of the dataset statistics (Table 4.2), we see that the anonymous benchmark has a high variance in the number of decision variables and constraints. Therefore, a feature vector that includes aggregated values could distinguish the problem instances. On the other hand, where item placement has the same number of decision variables and constraints, a shallow feature vector could not capture the graph connectivity properties, nor the coefficients values. Between these two cases, the load balancing benchmark has the same number of decision variables, while the number of constraints do not have a high variance (64,081 to 64,504 constraints). Shallow embedding was able to cluster problem instances, but its clusters were not correlated to the final solver's costs. The learned embeddings in our method is discriminative in the three benchmarks.

[4]The implementation of shallow embedding is provided in the supplementary material. There is no publicly available implementation of Hydra-MIP.

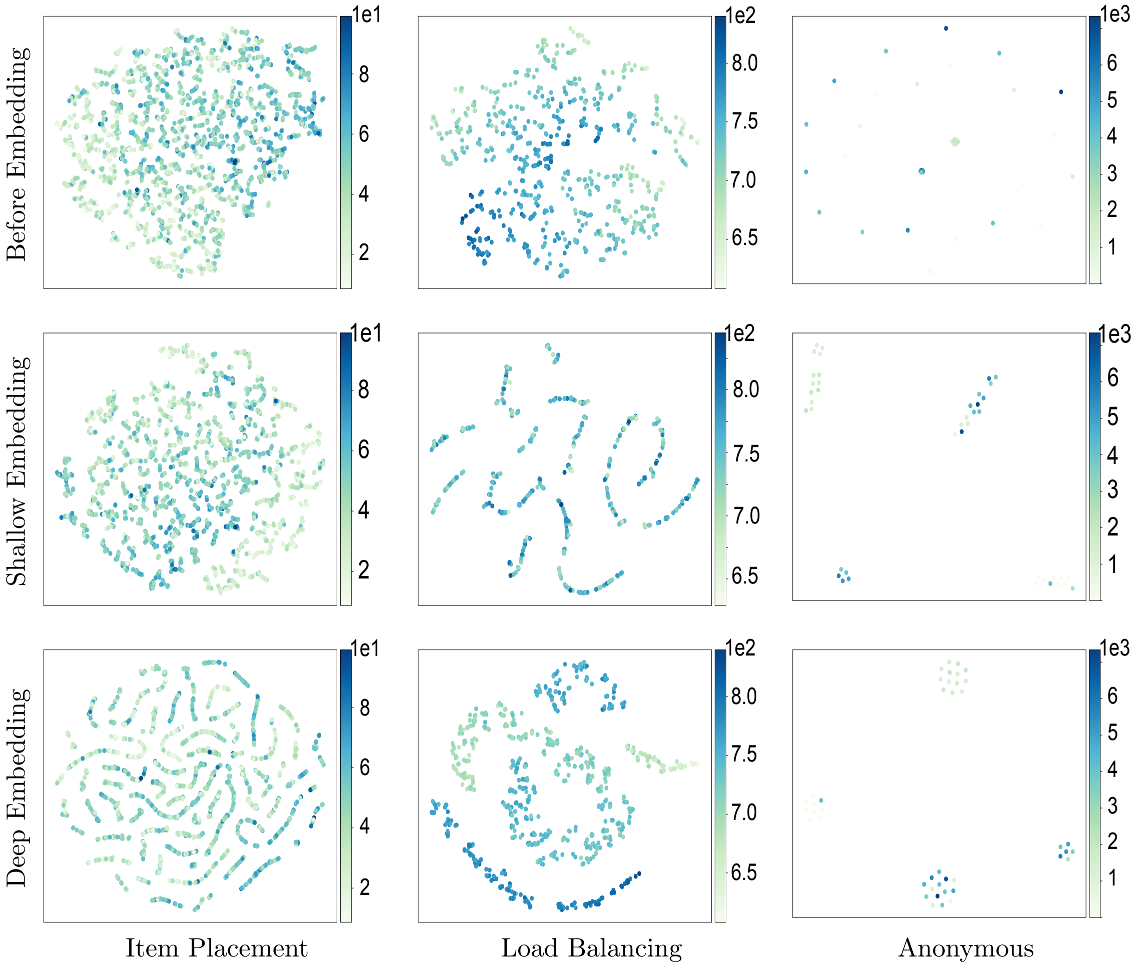
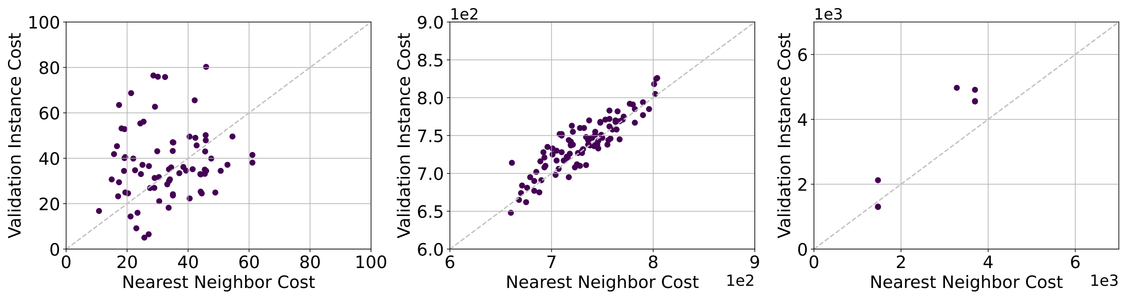


Figure 4.7: Vector representations of MILP problem instances visualized using t-SNE [173]. Each point is a problem instance where the color denotes its solution's cost using SCIP's [115] default configuration. Item placement, load balancing and anonymous represent benchmarks from the ML4CO dataset [129]. The first row (Before Embedding) represents the feature vector of instances before learning any similarities (i.e., random). The second row (Shallow Embedding) encodes the vector representation using [185]. The third row (Deep Embedding - Our Method) encodes the vector representation in the learned embedding space. In the item placement benchmark, shallow embedding does not offer any discriminative capability. In the load balancing benchmark, it could cluster problem instances, but clusters are not correlated with the final solver's costs. Shallow embedding uniquely embeds the anonymous benchmark and the embedding is correlated to the final costs. The discriminative power of deep embedding is evident in the three benchmarks, where similarity is directly correlated with the cost after running the solver using the default configuration.



(a) Item Placement MAE=18.07 (b) Load Balancing MAE=14.46 (c) Anonymous MAE=801.36

Figure 4.8: Cost (primal bound) of using the predicted configuration from the nearest neighbor in the learned metric space (x-axis) as compared to the actual cost after using it for the validation instance (y-axis).

4.7.4 Prediction Accuracy

A key question in our approach is whether the nearest neighbor in the embedding space would exhibit a similar solver behavior when using its parameter configuration. Here, we embed the validation instances using our trained model, and then obtain a parameters configuration from the nearest neighbor. Then, we run the solver using the predicted parameters configuration on the validation instances (T=15mins). Figure 4.8 plots the solution's cost of the predicted parameters configuration from the nearest neighbor (x-axis) vs. its solution's cost on the validation instance (y-axis). It shows that there is indeed a correlation between the final cost of the solution using the predicted parameters configuration, and the stored nearest neighbor cost using that configuration. The mean absolute errors (MAE) were 18.07, 14.46, and 801.36 for item placement, load balancing and anonymous, respectively. This correlation proves that, in reality, similar MILP instances based on the learned metric space expose similar solver behaviors yielding similar solution costs. In other words, finding a good parameters configuration for one problem instance can be used for similar instances without repeating an exhaustive search at deployment time.

4.7.5 Comparing to Baselines

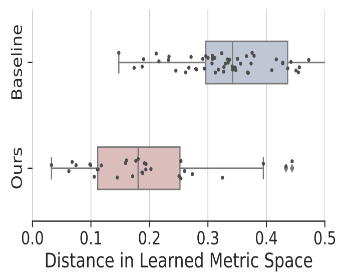
We compare our method against existing approaches in Table 4.3. The first baseline is using SCIP's default configuration, which is usually used by most practitioners. In addition, we obtain an incumbent configuration by performing a configuration space search on the training instances using SMAC [111]. We perform this search for each benchmark separately. Although the number of unique configurations explored was 51012 over a period of over 12000 core-hours, this represents a small subset of the configuration space. Moreover, we implement Hydra-MIP [185], which uses a statistics-based vector for instance embedding and pair-wise weighted random forests for configuration selection. In Hydra-MIP, the pairwise weighted random forests (RFs) method is used to select amongst m algorithms for solving the instance, by building $m \cdot (m-1)/2$ RFs and taking a weighted vote. In our processed dataset, the number of *unique* configurations explored offline using

Table 4.3: Our Method vs. Existing Approaches. In the dataset [129], there are 100, 100 and 20 test instances for the item placement, load balancing and anonymous benchmarks, respectively. Imprv. represents the average solution's cost improvement over the cost obtained using the default configuration of the SCIP solver. Cost is the value of the MILP objective function using the solution found by the solver. Since the smallest problem instance takes several days to solve to optimality, we limit the runtime to 15mins as suggested in [129] (Section 4.7.1). Wins represents the number of instances that a method solved with the lowest cost within the time limit. Shallow embedding + KNN (our baseline) uses the same embedding vector as [185]. Deep embedding (our method) is evaluated at $k = 1$ and $n = 1$ (See. Algo. 3).

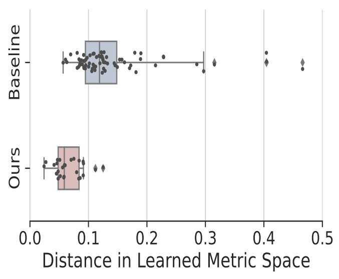
	Item Placement		Load Balancing		Anonymous	
Configuration	Wins	Imprv. $\downarrow$	Wins	Imprv. $\downarrow$	Wins	Imprv. $\downarrow$
No Solution Found	0	-	0	-	11	-
Default SCIP Config	1	-	34	-	1	-
Incumbent from SMAC [111]	8	0.24±0.16	4	0.01±0.03	1	0.01±0.00
Hydra-MIP [185]	10	0.25±0.09	17	0.02±0.01	0	-
Shallow Embedding + KNN	16	0.17±0.08	5	0.04±0.06	3	0.11±0.02
Deep Embedding + KNN	**65**	**0.38±0.06**	**40**	**0.04±0.03**	**4**	**0.26±0.07**

SMAC are 22580, 27971 and 461 for the item placement, load balancing and anonymous training benchmarks, respectively. Among those, the number of *unique* configurations that worked *best* on their respective instances (excluding unsolved instances) are 4325, 3987 and 53. As a result for the Hydra-MIP approach, building the portfolio by performing algorithm selection using pairwise RFs is computationally infeasible (memory and compute). For example, in the item placement benchmark, a total of $4325 \times 4324/2 = 9350650$ RFs are needed. To obtain results for Hydra-MIP, we selected a subset of the top 100 performing configurations in the item placement and the load balancing benchmark, and used all 53 best configurations of the anonymous benchmark. Lastly, we compare against using the shallow embedding from Hydra-MIP with KNN, which avoids the scalability limitation of RFs. Table 4.3 reports the number of instances solved with the lowest cost in each method, along with the average cost improvement over using the default configuration. We see that our method predicts configurations that solve more instances, with up to 38% improvement in the cost of the objective function (confidence level of 95%).

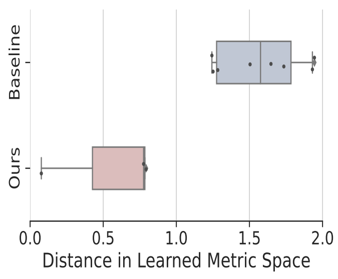
Moreover, we investigate how our method brings instances with similar final costs close to each other by plotting the winning predictions against their distance from their neighbors in the learned embedding space. In Figure 4.9, the x-axis represents the distance between the validation instance and its nearest neighbor, while the y-axis represents the method that offers a better parameters configuration. We observe that the smaller the distance between the validation instance and its nearest neighbor in the learned embedding space, the more probable the neighbor's parameters configuration to yield a better solution than other baselines. In other words, our method correlates the similarity of the learned embedding to the final solution costs.



(a) Item Placement



(b) Load Balancing



(c) Anonymous

Figure 4.9: Similarity in the learned embedding space. The x-axis represents the distance between the validation instance and its nearest neighbor in the learned metric space. The y-axis represents the method that gives a better solution. The closer the nearest neighbor to the validation instance, the better the predicted configuration by our method. Baseline represents the method with the lowest cost amongst the default configuration, SMAC's incumbent, Hydra-MIP and shallow embedding.

4.8 Conclusion

Generalizing to Other Solvers. MILP solvers expose different configuration parameters for their internal algorithms. For example, while SCIP exposes over 2500 parameters[5], CPLEX exposes 182 parameters[6] and Gurobi exposes 100 parameters[7]. Due to the different algorithm implementations, only a small subset of parameters have an exact match across all solvers. SCIP has been used in this work for a number of reasons: (1) it is a stable open-source solver and its algorithms are comprehensively documented, while commercial tools hide their implementation details (2) it exposes a large number of configuration parameters to tune, and (3) it has been used in previous related works [57, 100, 147, 172].

In order to generalize our method to other solvers, it is important to note that a solution's cost depends primarily on: (1) the problem instance, (2) the solver used (including the specific solver version), (3) the time limit, (4) the hardware resources given to the solver (cores and memory), in addition to (5) the configuration parameters. For the purpose of learning similarity between MILP instances, the solver's costs are used as a subjective measure of the similarity between two instances that use the same solver version, time limit, hardware resource, and configuration parameters. Replacing the solver with another solver is possible for the sake of getting costs that could be used to measure the similarity between different MILP instances. However, it is critical to fix all parameters of the solving environment (hardware, solver tool and its version, time limit, configuration parameters) in order for the cost to be representative of the similarity. Once a similarity measurement is established, two similar instances in one solver's environment could potentially be used to determine that these two instances will have similar costs in another solver's environment. However, we have not investigated this path in the scope of this study and will leave it for future work.

[5]`https://www.scipopt.org/doc/html/PARAMETERS.php`

[6]`https://www.ibm.com/docs/en/icos/12.8.0.0?topic=cplex-list-parameters`

[7]`https://www.gurobi.com/documentation/9.0/refman/parameters.html`

Limitations. Our adoption of metric learning in configuring MILP solvers relies on data that represent the same problem being solved repeatedly. In Section 4.6.1, we sampled triplets of anchor-positive and anchor-negative based on C_{thr}. It is infeasible to identify similar triplets if problem instances are coming from different distributions where the range of their costs varies significantly. For example, the cost range of the Item Placement benchmark is [0, 100] while the cost range of the Load Balancing is [500, 1000]. While finding a dissimilar pair is straightforward (e.g., one instance from each benchmark), it is hard to find a similar pair where the cost difference is $< C_{thr}$. This means that in order to train a deep embedding model for learning MILP similarity, the MILP formulation needs to represent a problem being solved repeatedly, which is materialized in the number of decision variables or constraints in the problem, as well as the range of their solutions' costs.

Reproducibility. In Section 4.7, we refer the reader to the original dataset to download. A link to the processed dataset (learned embeddings) is available in the supplementary material. In addition, we describe our setup for training and the pipeline architecture. The source code along with the training implementation is available in the supplementary material.

Chapter 5

Efficient Exploration Using Predictive Modeling

5.1 Introduction

The proliferating scale of technology nodes enables state-of-the-art systems-on-chip (SoCs) to host billions of transistors on a single die. The realization of these massive computational capabilities has been enabled by EDA (Electronic Design Automation) tools for front-end and back-end design. With hundreds of parameters to tune in each tool, design space exploration and efficient physical implementation have been increasingly challenging and require a massive amount of computations to achieve the required Quality of Results (QoR). EDA tools are essentially in a continuous search for an optimal functional design that meets the Performance, Power and Area (PPA) requirements. When the search space is huge, unbounded exploration is costly, or even infeasible. For large EDA companies, these challenges have demanded frequent upgrades to their compute infrastructure raising the cost of operation and maintenance. For small and rising EDA teams, the significant upfront capital investment of setting up a suitable compute environment for their EDA jobs has stiffled their innovation.

In the recent years, there has been a growing trend among EDA teams to utilize elastic compute environments (i.e. Cloud) to gain near-instant access to compute resources [156, 49]. Cloud vendors offer three main categories of elastic resources: (i) *Infrastructure as a Service (IaaS)* where users get virtual access to physical hardware resources (i.e. virtual machines), (ii) *Platform as a Service (PaaS)* where users run and manage their applications abstracting away the underlying server administration, and (iii) *Software as a Service (SaaS)* where users get on-demand access to software without having to manage their own installation. While PaaS and SaaS are the most appealing ways for users to run their computation on the cloud, EDA teams have benefited most from having fine control over their provisioned virtual machines in IaaS offerings.

Migrating EDA jobs to the cloud has helped teams meet the demands of their tapeout schedule, hence reducing the time to market. For example, horizontal scaling by launching more virtual machines allows EDA teams to complete a highly-parallelizable job, such as simulation, in less time. Moreover, the scale of the cloud has accelerated the design space exploration and the physical design optimization by launching many jobs in parallel with different parameters. Furthermore, EDA teams have the flexibility to choose the configuration of hardware that meets their needs for the exact pending job and only for the time needed to complete it. The advantage is that teams pay only for the amount of compute time spent down to per-second billing.

However, moving EDA jobs to the cloud is not a straightforward path, especially for companies with little or no experience managing cloud deployments. For example, design teams need to choose the right machine configurations that achieve the best performance for their job. While simulation and verification are known to be embarrassingly parallel (i.e. directly benefiting from the scale of the cloud), the compute requirements for the synthesis and physical design stages are not well-studied, especially in multi-tenancy environments. In addition, upfront runtime estimation is needed in order to best utilize the allocated budget (measured in \$) while meeting the tapeout schedule.

In addition, optimizing the cost of running cloud workloads have been intensively studied over the past decade, both from the vendor's operational perspective [38, 34, 35, 163, 149, 61, 164] and the customer deployment perspective [152, 37, 141]. However, most of the proposed techniques are general purpose and do not consider the potential optimization gain (performance-cost efficiency) that can be achieved from a better understanding of the workload being deployed (e.g. compute, memory and disk access patterns). Therefore, more focused studies have been dedicated to running HPC workloads on the cloud [63, 148, 167, 121]. In particular, it has become apparent that the cost-effectiveness of HPC on the cloud depends mainly on "raw performance" and "application scalability". While EDA jobs can be broadly classified as high-performance computing jobs [156], our study shows that they are inherently diverse in their compute requirements. For example, we show that while placement and routing need machines with higher memory-to-core ratio, logic synthesis can perform just as well on general-purpose machines. Therefore, deploying all EDA jobs on high-end machines incurs a huge and unneeded cost overruns. Unfortunately, there has been little to no public studies on the unique characteristics of specific EDA jobs as a high-performance workload.

In this work, we aim to efficiently explore the design space with robust predictive modeling. To that end, we perform an empirical study of characterizing and optimizing EDA jobs when running in cloud environments. We summarize our contributions as follows:

1. We identify and formulate the problem of migrating EDA jobs to the cloud. The goal is to utilize cloud resources in order to meet tapeout deadlines with minimum cost.

2. We characterize the performance of four EDA key applications (synthesis, placement, routing, and static timing analysis) under different machine configurations. Using our observations, we present practical recommendations for the types of cloud instances to provision for each application.

3. We propose an integrated framework for analyzing and optimizing EDA flows on the cloud.

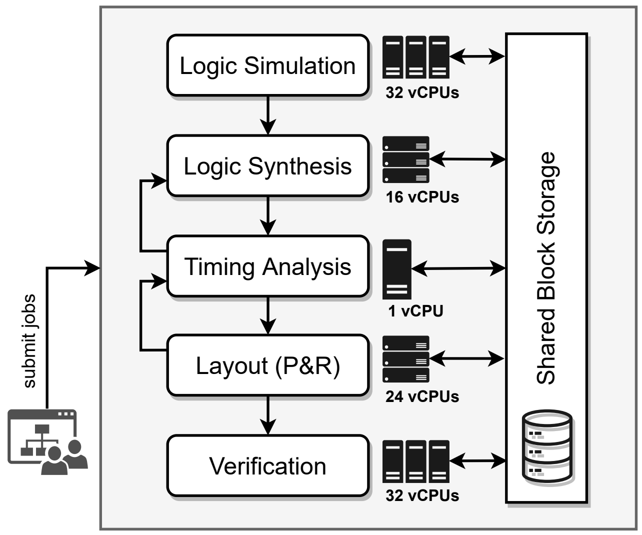


Figure 5.1: A reference EDA flow on the cloud

Using this framework, we developed a novel model based on Graph Convolutional Networks (GCNs) that predicts the total runtime for a given EDA job using certain machine configurations. Our model achieves a high runtime prediction accuracy of 87%.

4. We provide a new problem formulation for running EDA flows on the cloud as an optimization problem and map it to the classical multi-choice knapsack problem (NP-hard). We provide an open-source implementation that recommends optimal machine configurations that minimizes the total cloud deployment cost (measured in $) by an average of 35.29%, while meeting deadline schedule constraints.

5. We extend our optimization method to further reduce the deployment cost by up to 73% when using cloud spot instances (machines with lowered service level guarantees).

In the following, we give a brief background in Section5.2, and discuss the related work in Section 5.3. After that, in Section 5.4, we formulate the problem and discuss our proposed framework to address it. Then, in Section 5.5, we present our experimental results. Lastly, we summarize the work in Section 5.6.

5.2 Preliminaries

Cloud computing refers to the elastic compute resources that can be provisioned, scaled up or shutdown on demand. Cloud providers, such as Amazon Web Services, Microsoft Azure or Google Cloud, virtualize their physical infrastructure to share processing time, memory, storage and network bandwidth among many users (known as tenants). In other words, virtualization creates a *multi-tenancy* environment where more than one tenant use the same underlying hardware. In order to achieve this, cloud vendors use a specialized software called the *Hypervisor*. The Hypervisor isolates each tenant's resources in a Virtual Machine (VM) that is accessible only by its owner, or other authorized accounts.

Cloud Offerings. From a user perspective, cloud vendors offer three main categories of elastic resources: (i) *Infrastructure as a Service (IaaS)* where users get virtual access to physical hardware resources (i.e. virtual machines), (ii) *Platform as a Service (PaaS)* where users run and manage their applications, abstracting away the underlying server administration tasks, and (iii) *Software as a Service (SaaS)* where users get on-demand access to software without having to manage their own installations. The shortest path to migrating legacy EDA applications to the cloud is to make use of IaaS offerings.

In a standard IaaS offering, VMs are sold in units of: (i) vCPU: a virtual CPU is seen as a single physical CPU core, or a single thread if Simultaneous Multi-Threading (SMT) is enabled in the underlying hardware (called Hyper-Threading in Intel's processors), (ii) Memory: a fixed number of memory pages are solely reserved for the use of a VM and is expressed as the total memory size reserved, and (iii) Storage: the size and type of the underlying storage device partition mounted on the VM. In addition, cloud vendors offer different VM types that comprise varying combinations of vCPUs and memory. For example, compute-optimized VMs have more vCPUs per unit of memory that are suitable for compute-intensive applications. A memory-optimized VM has more memory allocated per vCPU, which makes it suitable for memory-bound applications. While IaaS offerings have advanced to virtualize other hardware such as network cards, GPUs and FPGAs, the scope of this study is limited only to the compute resources in a standard IaaS offering.

Cloud EDA Flow. Figure 5.1 shows a reference EDA flow, where different stages of the flow run on different VM configurations. The advantage is that a highly-parallel job, such as simulation or verification, can take advantage of launching more VMs to accelerate job completion. In addition, handing-off a job from one VM (e.g. running logic synthesis) to another (e.g. running timing analysis) can take advantage of shared block storages to avoid data transfer time. A shared block storage can be mounted to multiple VMs at the same time in what is known as a multi-attach feature. Our study is concerned with selecting VM configurations for the reference flow that minimize the total deployment cost while still meeting deadline constraints.

Virtual Machine Provisioning. Cloud vendors offer a programmable API (Application Programming Interface) to manage the life-cycle of a virtual machine. What that means is that EDA teams can, and should, automate the process of VM creation as needed for a given job. This helps in reducing the cost of on-demand VMs, as well as increasing the utilization of the provisioned resources (i.e. no idle resources are being billed). Moreover, full-environment images of a VM can be created as a template for future provisioning. This means that a provisioned VM will have all the tools, dependencies and environment configurations needed to run the job as soon as it starts, avoiding installation or setup time. Furthermore, VM provisioning time usually takes minutes, and in some cases seconds, which can be neglected as a proportion of the total runtime of an EDA job (usually measured in hours). Taking full advantage of the programmable API unleashes the potential of the cloud for expedited design space exploration as well as parameter tuning tasks in EDA tools.

Dynamic Scaling of Provisioned Resources. Virtual machines can be resized (i.e. increase or decrease memory or #vCPUs). However, for technical reasons, scaling cannot be performed dynamically. In other words, a VM has to be shut down, resized and then turned back on with the new resource allocations. This process will terminate any running jobs on the VM. EDA flows requiring dynamic scaling of resources can utilize software containers [24]. Containerization is a technology that offers the control and isolation of resources at the operating system kernel level. Cloud vendors have recently started to offer containers as a service (CaaS). This allows for the dynamic scaling of resources with lightweight API controls over the life-cycle of a provisioned container. EDA teams can also benefit from the fast provisioning time of containers for their EDA jobs.

5.3 Related Work

Optimizing cloud deployment costs is a two-sided coin. On one side, cloud vendors seek to minimize operational costs to be able to offer competitive pricing to their customers. To that end, a plethora of research studies have been conducted with the goal of minimizing the provisional costs of running and maintaining the infrastructure [38, 34, 35, 163, 149, 61, 164]. Efforts range from high-level scheduling and provisional algorithms to low-level power and energy optimization. On the other side, customers have also been striving to reduce their cloud spending. Rodriguez *et. al.* proposed a resource provisioning and scheduling strategy for scientific workflows [152]. Their meta-heiristics algorithm aims to minimize the total cost of workflow execution while meeting deadline constraints. Chen *et. al.* extended Rodriguez's work by implementing a genetic algorithm that was able to produce scheduling results that meet tighter deadlines and take advantage of the cloud heterogeneity [37]. However, both works are general-purpose workflow scheduling techniques and do not consider the inherent compute requirements of each workflow step. More recently, Osypanka and Nawrocki developed a machine learning framework to optimize the cost of a running system by continuously looking at historical data of usage patterns and modifying infrastructure configurations accordingly to save costs [141]. While these approaches generally work well, they overlook the optimization gain that can be achieved from a better understanding of the workload being deployed (e.g. compute, memory and disk access patterns), which is the case for most high-performance computing (HPC) workloads.

To gain a better understanding of HPC on the cloud, which is offered by multiple vendors [11, 60, 126], Gupta *et. al.* performed a performance evaluation on a range of platforms, from supercomputers to cloud environments [63]. They concluded that the public cloud is only cost-effective for small-scale HPC deployments, but should be considered to complement on-premises resources. Prukkantragorn and Tientanopajai also recognized the high cost of running HPC workloads on the cloud [148]. They analyzed traces from running the high-performance linpack benchmark [145] on a cloud provider with varying problem sizes. Their results showed that there is a pivot point (problem size), where larger problem sizes would finish execution faster when running on a high-end HPC offering. Problem sizes smaller than the pivot would always be more cost efficient to run on the low-end HPC offering. Nonetheless, they only consider single-application runs, and not workflow-like jobs similar to [152, 37]. Furthermore, Somasundaram *et. al.* developed a cloud resource broker to efficiently manage cloud resources for scientific applications [167]. The main limitation of their simulation is that it does not consider the hetereogenous nature of the cloud resources (i.e. different machine configurations). In their comparative study of HPC on the cloud, Marathe *et. al.* compared running HPC workloads on a dedicated cluster versus a cloud cluster for performance and cost efficiency [121]. The main conclusion was that the cost-effectiveness depends on the “raw performance” and the “application scalability” of the workload itself. This leads us to the importance of having a deeper look at the EDA-specific workload when being run on cloud resources. In other words, EDA jobs could have unique performance and scalability characteristics that require further studies.

This need has been realised by the EDA community who have witnessed increasing interest in using the public cloud. While cloud vendors try to maximize hardware utilization, EDA teams aim to get access to high-performance VMs with the lowest cost. On the architectural side, Lin *et. al.* proposed a prototype system design, called Web-EDA, for managing EDA projects on the cloud [110]. Similarly, Man *et. al.* described a system architecture for small and medium-sized teams aiming to perform IC design and testing on the cloud [118]. Both proposals have been addressing the architecture side of managing EDA assets in a distributed cloud environment. However, they do not address the computational requirements of the specific EDA jobs.

On the infrastructure side of managing cloud jobs, Kamath *et. al.* have described in detail Intel’s compute infrastructure, addressing the high availability requirements for EDA jobs that are interactive in nature [81]. While this work provides details on the software and hardware that supports thousands of design jobs, it can be classified as a private cloud environment where the organization owns and manages the underlying infrastructure. Moreover, Seghal *et. al.* discussed the security and licensing model of EDA tools on public cloud environments [156]. Authors also identified the types for workloads on the cloud that fit the EDA use-cases. However, they provided a broad classification for EDA tasks as single-run compute-intensive jobs. Our characterization shows that different jobs in the EDA workloads have different compute requirements.

On the application side, Chen *et. al.* have proposed a cloud-native floorplanning algorithm that takes advantage of the massive cloud scale [36]. More recently, scheduling EDA jobs on distributed environments have also gained interest [160, 137]. However, both studies have been focused on maximizing the utilization of a single cluster, and not on the performance of EDA jobs on multi-tenant virtualized environments.

Our work supports and advances the recent need to move EDA jobs to public and private cloud environments and offers detailed insights as well as a recipe for EDA teams to maximize the benefit from using the cloud while minimizing the cost.

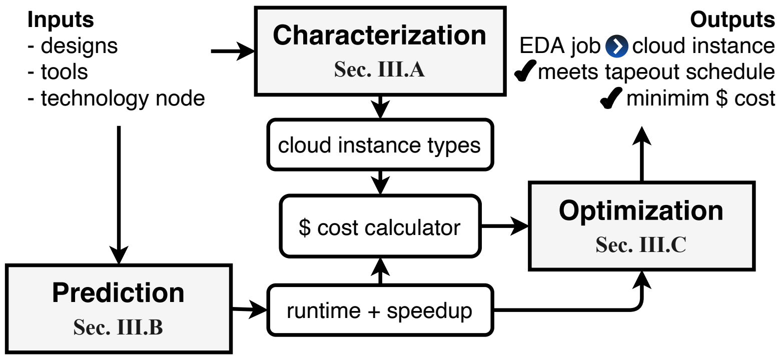


Figure 5.2: Workflow of optimizing EDA cloud deployments

5.4 Method

Problem Definition. A fundamental question that faces EDA teams when migrating their EDA jobs to the cloud is: what configurations of VMs should be provisioned for each job? And how can the job completion time be reduced while minimizing the cost?

In order to answer these questions, Figure 5.2 draws our workflow that we propose in this thesis. Specifically, we introduce the following problems:

Problem A. What is the right VM configuration for a given EDA job? In this context, a configuration refers to the size of the VM in terms of #vCPUs. To address this problem, we characterize four main EDA application jobs, namely: synthesis, placement, routing and static timing analysis. We focus on characteristics that are intrinsic to the EDA job which affect the completion time.

Problem B: Given a design (in RTL or Netlist), estimate the runtime for a given EDA job when using 1, 2, 4 and 8 vCPUs. Our proposed prediction model learns internal graph features of the design that affect the total runtime of a given job on different machine sizes.

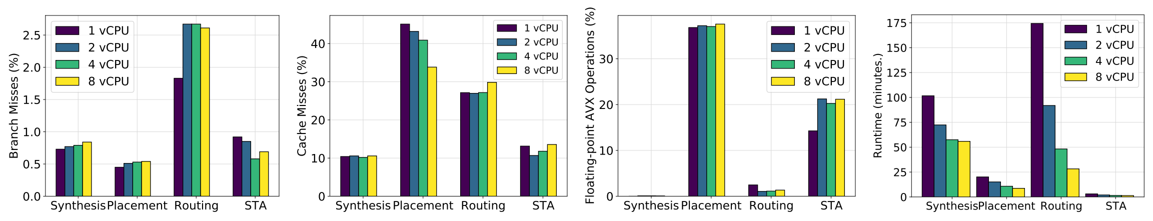
Problem C: Given the estimated runtime for each job under 1, 2, 4 and 8 vCPUs, as well as a deadline constraint, select a machine size for each job such that the deadline is met and the total cost is minimized. We address this problem using a mapping to the multi-choice knapsack problem and implement an optimal solution using dynamic programming.

To address Problem A, we characterized the four jobs using commercial tools and a SPARC core design from OpenPiton design benchmark [16] on a 14 nm technology node. All optimization efforts of the tools have been set to “high". In addition, timing-driven optimizations have been used when available. Other tuning parameters are kept to their default values. We then collected the execution data from the system's hardware performance counters for further analysis. Table 5.1 shows relevant system configuration details of our experimental setup. We used Linux Control Groups to simulate a multi-tenancy environment (reserving CPU cores and memory pages) as used in cloud hypervisors.

We used the Linux perf utility to instrument the hardware performance counters. We swept the values of the performance counters under 1, 2, 4 and 8 vCPUs, and experiments are repeated three times to get accurate measurements. This will show us how the different jobs can take advantage of

Table 5.1: Characterization system configuration

Processor	Intel Xeon E5-2680 v4 @ 2.40GHz (up to 3.30Ghz) Dual socket, 14 cores each (24 threads with hyper-threading) x86_64 architecture, Power governor: performance
Memory	128 GB DDR4
Cache	L1: 32 kB (per core), L2: 256 kB (per core) L3: 35 MB shared by all cores (per node)
Disk	Samsung SSD 850 EVO 500GB
OS	CentOS Linux 7 (Core), Control groups: enabled Linux kernel: 3.10.0-957.27.2.el7.x86_64



(a) Branch Misses (b) Cache Misses (c) Floating-point Operations (d) Total Runtime

Figure 5.3: Performance characterization of four representative EDA jobs

the cloud scalability, as well as their behavior. Due to license limitation of the available commercial tool, we were only able to analyze up to 8 threads. Additionally, due to the economic dynamics of the cloud offerings, virtual CPUs are sold in units of power 2 (e.g. 1, 2, 4, 8, 16, 32, ..etc.). Therefore, finer-grained analysis (e.g. at 5 vCPUs) would not apply to a real-world cloud offering. Nevertheless, the trend we observe in our analysis is indicative of the EDA tool behaviors and scaling-readiness.

It is important to note that performance counters measurements are influenced by both the underlying micro-architecture of the processor (e.g. pipeline depth, in-order vs. out-of-order execution paradigm, branch predictor accuracy, cache configuration, etc.) and the workload's characteristics (e.g. instruction mix, memory access patterns, types of branch instructions, etc.). Since we are interested in the workload's characteristics, we will be focusing on the relative comparison of the measurements, and not the absolute numbers (hence using the same platform).

5.4.1 EDA Flow Characterization

Branch Prediction. Figure 5.3 summarizes our findings from the characterization experiments. First, we observe that routing has a higher percentage of branch misses. We attribute this value to the nature of the routing algorithms where there can be many trials before a net is successfully routed with no design rule violations. In particular, graph search algorithms in the global routing step and bipartite matching algorithms in the detailed routing [146] encompass a large portion of conditional statements that cannot be avoided. Rip-up and reroute techniques also contribute to halting the continuous execution of the routing algorithms.

Memory Access Patterns. In Figure 5.3-b, we observe that placement and routing have

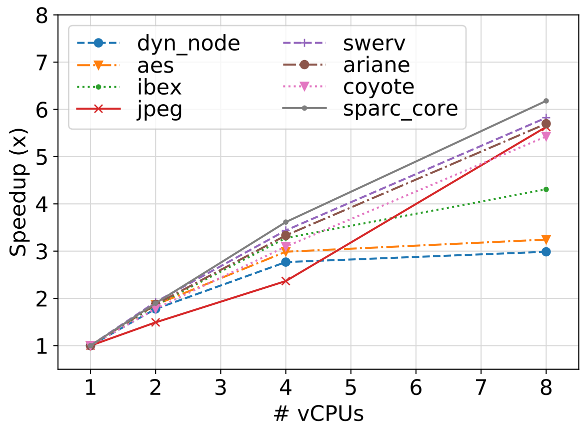


Figure 5.4: Routing speedup for different designs. *dyn_ node* is the smallest and *sparc_ core* is the largest (#instances).

significantly higher cache misses than synthesis and STA. Placement has a 45.11% cache misses rate when using 1 vCPU and 33.84% when using 8 vCPUs, while routing has 27.15% and 29.84% cache misses rate using 1 and 8 vCPUs respectively. We attribute this to the nature of the analytical component in the placement engine that tries to optimize the half-perimeter wire length (HPWL) across all the chip instances using convex optimization methods. This needs access to large vectors to calculate the gradients, hence benefiting from the more cache available with more vCPUs.

Floating-point Operations. In Figure 5.3-c, we observe that the placement job requires more floating-point operations that run on Advanced Vector Extensions (AVX) hardware. This can be attributed to the analytical engine that tries to optimize the wire length across all the chip area using convex optimization methods. This involves calculating gradients which relies on floating-point operations. The STA job comes next in its percentage usage of the AVX hardware. This is consistent with the nature of STA algorithms where calculating slacks involves graph traversal from inputs to outputs, with access to floating-point values in the technology library.

Scalability and Speedup. In Figure 5.3-d, we observe that the routing job scales well with more #vCPUs. This is consistent with the nature of the routing job, where nets in independent grid cells can be routed in parallel with no conflict, as opposed to synthesis, placement and STA where internal algorithms have inherent dependencies. Further analysis of the routing job, Figure 5.4 plots the speedups achieved on different designs of different characteristics and sizes. It shows that adding more vCPUs does not eminently scale the routing job in all designs. Smaller designs (such as dynamic_node and aes) have almost equal speedups for 4 and 8 vCPUs. This means that the provisioned vCPUs might not offer the expected benefit from the cloud scale, and that there is an opportunity to achieve the same outcome in nearly the same time with less resources.

Results from Figure 3 play a vital role in our optimization framework. They are used to determine the type of virtual machines that give the best performance for a given job. Subsequently, they determine the pricing (per second) for a job to be executed, which is also what our optimization algorithm minimizes.

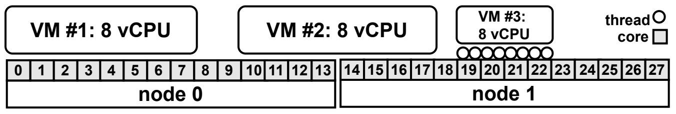


Figure 5.5: VM placement scenarios in a cloud environment.

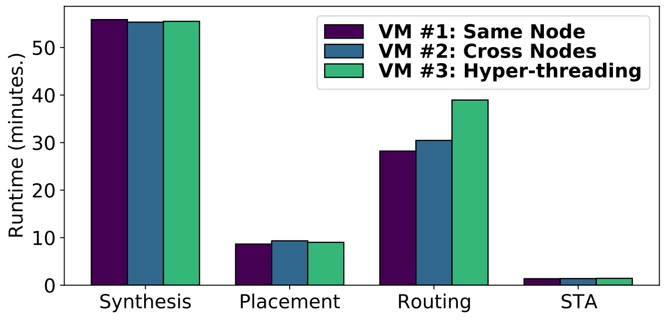


Figure 5.6: Impact of VM placement on the total runtime.

Hyper-threading and NUMA effects. The placement of a VM on a physical server can significantly impact the performance of EDA workloads running inside it. Since modern cloud data centers use dual-socket servers with hyper-threading enabled, non-uniform memory access (NUMA) effects and competition for L1 cache can negatively impact the tenant's workload. Figure 5.6 shows three different scenarios of VM placement on a cloud server. In particular, VM #1 is given 8 vCPUs on 8 different cores on the same node. This means that the virtual CPUs only share the last-level cache (LLC), but have their own level 1 cache (L1). VM #2 is allocated across the two nodes. While each vCPU has its own L1 cache, accessing LLC on the other node takes longer. VM #3 is allocated on a single node where hyper-threading is enabled. This is a scenario where each 2 vCPUs share the same L1 cache, and all vCPUs share the same LLC.

We observe that, for the routing job, and as compared to VM #1, VM #2 and VM #3 take 7.98% and 38% more time to complete the job respectively. The cache misses rates are 28%, 24% and 29% for VMs 1, 2 and 3 respectively. Although VM #2 has less cache misses rate, it takes longer as it accesses parts of the data on the other node. Additionally, VM #3 reveals the competition for cache that happens when hyper-threading is enabled.

Main Takeaways. From the point of view of EDA teams running their EDA applications on the cloud, we summarize our main recommendations:

1. Synthesis and STA jobs perform well on general-purpose VM instances with a balance between computations and memory access. Placement and routing require VM instances with higher memory-to-core ratio, with routing demanding more available L1 and LLC cache.

2. Placement jobs should be run on a compute instance with an underlying processor that supports Advanced Vector Extensions (AVX). STA jobs would also benefit from AVX hardware.

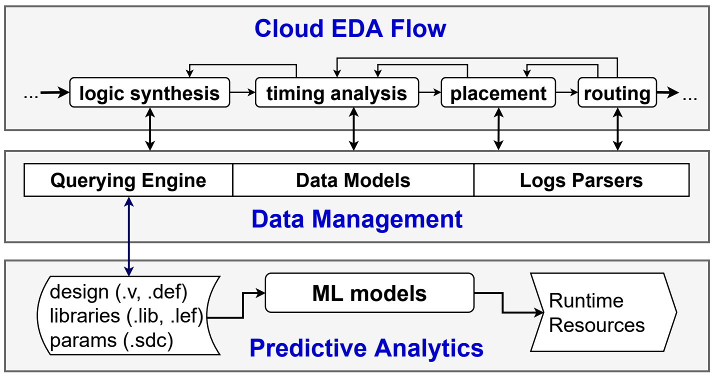


Figure 5.7: Overall system architecture for analyzing and optimizing EDA flows on the cloud.

3. On large designs, routing jobs scale well with the number of vCPUs allocated. However, on small designs, speedup is capped at a certain point.

5.4.2 Runtime Prediction

To address Problem B, we state that the runtime of chip design tasks depends on a number of factors such as the design itself, the tools used, the technology node, the parameters used to instruct the tools and the VM configuration. Without losing generality, when using the same tools, technology node, default parameters and VM configuration, the runtime of a certain job depends on the complexity of the design itself.

In order to build a predictive model for cloud EDA flows, we propose a fully-integrated workflow to collect and analyze data, as well as to build and iteratively fine-tune models shown in Figure 5.7. At the heart of the workflow lies a data management module that aids in collecting, storing and analyzing EDA data. In addition, the module offers a flexible querying engine to help in developing predictive models. We describe each module in the following subsections.

Data Management. Migrating to a cloud infrastructure not only solves existing problems in legacy EDA projects, but also opens doors for a new generation of data-driven EDA applications. In [51, 80, 65], the authors proposed a system for measuring a design process by collecting the characteristics of design artifacts, design process, and QoR during the system development and using these data points to optimize the design and improve productivity.

To manage EDA data collected from cloud environments, we propose a fully-integrated data management framework, called EDA Analytics Central (EDAAC for short). The goal of EDAAC is to make it easy and reproducible to perform the following tasks:

- **Data collection:** from log files using predefined shell tools and python scripts that extract relevant metrics from an EDA flow.
- **Data storage:** to store and index data in a persistent structural database that can support data analytics tasks. Data is then accessed through standardized data models.

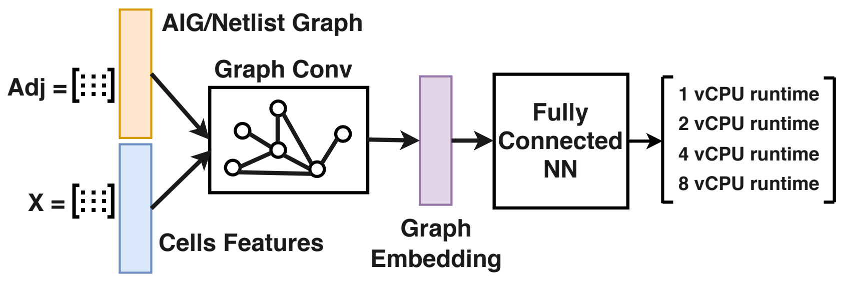


Figure 5.8: Our proposed runtime prediction model

- **Data querying:** to efficiently build predictive models utilizing the collected data, with the goal of improving the execution of EDA flows on the cloud.

We discuss the implementation details on EDAAC in Section 5.5.

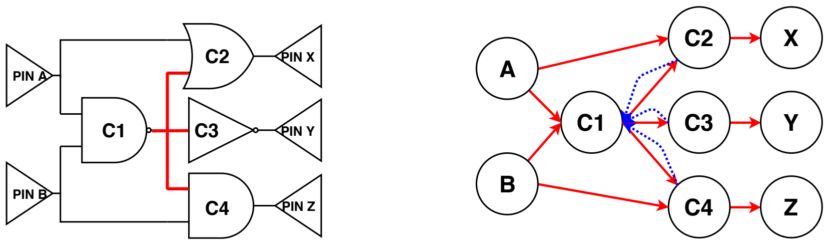
Predictive Model. A convenient starting point for building a runtime prediction model is to use the size of the design as the main feature, and fit a regression model on the collected data [181]. Hence, we built a baseline model that takes as input the number of nodes and edges of the representative graph of the design and outputs the runtime for different machine sizes (i.e. #vCPUs). As we will observe in the experiments section, these simple features are not distinguishing factors of the "complexity" of the design. In other words, two designs of the same size can take different runtime for a job because the structure of one of the designs is more complex. Therefore, we designed an improved model that takes into account the structure of the design to be used as features.

Figure 5.8 shows the architecture of our model. The model takes as input the design in RTL or netlist, and performs an embedding operation using Graph Convolutional Networks [92]. After that, a fully-connected neural layer transforms the embedding into predictions for the runtime under different machine sizes (i.e. #vCPUs). This model is trained for each application separately. Using the predicted runtimes, we can calculate the speedup gains from using 2, 4 or 8 vCPUs as compared to using only 1 vCPU.

We used two GCN layers with 256 and 128 hidden units each, followed by 1 fully connected layer with 128 units. The model is trained for 200 epochs using Mean Square Error (MSE) as a loss function and Adam as the optimizer (lr=1e-4). The loss function calculates the combined prediction error for all four runtimes (i.e. 1, 2, 4 and 8 vCPUs).

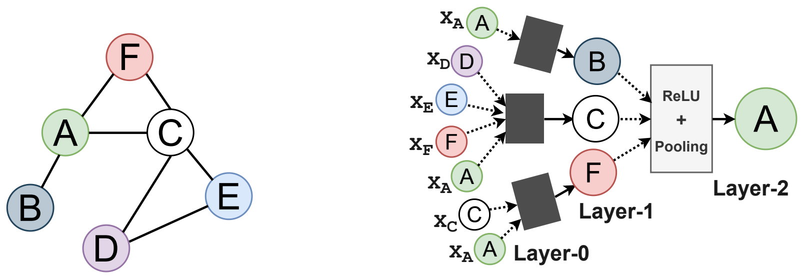
Processing Input Design. When building a model to predict synthesis runtime, the input is usually in RTL, which is not a graph. However, synthesis tools map the RTL into an intermediate representation such as And-Inverter Graphs (AIG) before synthesizing and mapping to a technology library. Therefore, our model can operate on the AIG representation of the design. The AIG is a Directed Acyclic Graph (DAG), which means it preserves edge directions for the GCN.

In the back-end EDA (P&R), the input is expected to be a netlist, which is represented as a hyper-graph. Figure 5.9 shows an example netlist (5.9-a) that has two input pins, four cells and three output pins. Cell *C1* is driving three other cells *C2, C3* and *C4* through the net marked in



(a) Netlist Example (b) Graph Representation

Figure 5.9: Graph extraction from design netlist.



(a) Input Graph (b) Convolution Operation

Figure 5.10: Graph Convolution Operation

red. It is natural to represent each cell and every IO pin as a node in the graph. Nets are converted to directed edges marked in red (5.9-b). This means that the feature vectors for cells *C2*, *C3* and *C4* will consider the feature vector of cell *C1* when it gets updated through the propagation rule defined in Equation 5.4.2. It also means that cell *C1* will not learn about cells *C2*, *C3* and *C4*. We can also generalize the edge directionality of a net by adding directed edges between both the driving cell and the load terminals (shown in dotted blue lines). This signifies that cell *C1* will consider the feature vectors of its loads.

Note that we only represent the nets connected to IO pins as *directed* edges to denote that nodes representing input pins have a blind receptive field, while nodes representing output pins communicate no feature vectors to other nodes. This restriction preserves important netlist features about IO pins.

For illustration, consider the adjacency matrix of the sub-graph that contains the cells (no self-loops). We also initialize a feature vector of length 2 for the purpose of illustration. We show a first iteration of the propagation rule below.

$$\mathcal{A} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad \mathcal{F} = \begin{bmatrix} 1 & -1 \\ 2 & -2 \\ 3 & -3 \\ 4 & -4 \end{bmatrix} \rightarrow \mathcal{X} = \mathcal{A} \times \mathcal{F} = \begin{bmatrix} 0 & 0 \\ 1 & -1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}$$

Notice how cells *C2*, *C3* and *C3* learned the feature vector of cell *C1*, but *C1* did not learn about its loads. Adding self-loops will allow nodes to include their own feature vectors in the calculation.

Graph Convolutions. In traditional Convolutional Neural Networks (CNNs), the idea is to apply a filter (also known as kernel) that produces a feature map from neighboring image pixels. In Graph Neural Networks (GNNs), the key idea is to generate node embeddings based on local neighborhoods. Figure 5.10 illustrates the *convolution* operation on graphs. Layer-0 embedding of a node represents its input feature vector, x_i. Nodes aggregate information from their neighbors in each convolutional layer. This aggregation is followed by an activation function, such as $ReLU$, and a pooling operation, such as *sum*-pooling. With that in-place, every layer is written as a non-linear function:

$$\mathcal{H}^{(l+1)} = f(\mathcal{H}^{(l)}, \mathcal{A})\,, \tag{5.1}$$

where $\mathcal{H}^{(l)}$ represents the activation at layer l, and $\mathcal{H}^{(0)}$ is the input feature matrix, $\mathcal{X}$. Looking at the embedding of each node, we can elaborate on Equation 5.1 as follows:

$$h_v^k = \sigma\ (W_k\ \sum\nolimits_{u \in N(v)} \frac{h_u^{k-1}}{|N(v)|}\ +\ B_k h_v^{k-1}) \quad \forall k \in \{1, ..., K\}$$

where, h_v^k represents kth-layer embedding of node v. W_k and B_k are the trainable matrices which are shared with all nodes of the graph. After K-layers of neighborhood aggregation, we get output embeddings for each node that can be fed into a loss function. We can then run stochastic gradient descent to train the aggregation parameters (i.e. W_k and B_k). A Graph Convolutional Network (GCN) that has an architecture of two convolutional layers allows a node to learn about the features of its neighbors in the first layer, and the features of its neighbors' neighbors in the second layer. In other words, increasing the number of layers in the GCN increases the receptive field size of each node.

5.4.3 Optimizing Virtual Machine Provisioning

Given the runtime estimates, we now address Problem C. Our proposed solution maps the problem to the multi-choice knapsack problem (MCKP) [85]. Using our predictions calculated in the previous section, each job can be run on a different machine configuration (i.e. #vCPUs), where each configuration completes the job in t time and costs p in total.

Formulation. Let $z_l(C)$ be an optimal solution defined on l applications and with total runtime constraint C:

$$z_l(C) := \min \sum_{i=1}^{l} \sum_{j=1}^{N_i} s_{ij} p_{ij} \tag{5.2}$$

such that,

$$\sum_{i=1}^{l} \sum_{j=1}^{N_i} s_{ij} t_{ij} \le C,$$

$$\sum\nolimits_{j \in N_i} s_{ij} = 1, i = 1, ..., l,$$

$$s_{ij} \in \{0, 1\}, i = 1, ..., l, j \in N_i$$

Algorithm 4: Optimizing Cloud Deployment Cost Subject to Time Constraint

```
Input  : Runtime constraint ($C$)
Input  : For each job, for each configuration ($N$):
    • $t$ - array of predicted job completion times
    • $p$ - array of job deployment costs
    • $l$ - array of job applications (i.e. classes)
Output: Selected configuration for each application ($s$)
1  $KC = [|l| \times C]$ lookup table. Initialize to $\infty$; $s = KC$
2  $w = [|l| \times N]$ weights table // represents runtime (sec.)
3  $v = [|l| \times N]$ values table   // represents cost ($)
4  for $i$ in $0, .., |l|$ do
5      for $j$ in $0, .., C$ do
6          $KC[i][j] = 0$ if $i = 0$
7          $KC[i][j] = \infty$ if $j = 0$
8          candidates = []; indices = []
9          for $k$ in $0, .., w[i-1]$ do
10             candidates = [candidates, $KC[i-1][j - w[i-1][k]] + v[i-1][k]$]
11             indices = [indices, $k$]
12         end
13         if candidates is not empty then
14             $KC[i][j]$ = min(candidates)
15             $s[i][j]$ = indices[index of (min(candidates))]
16         end
17     end
18 end
19 if $KC[|l|][C] = \infty$ then
20     No solution that satisfies time constraint. Abort.
21 end
22 $jobs$ = empty dictionary; $i = |l|$; $j = C$
23 while $i\,! = 0$ do
24     $jobs[i] = s[i][j]$
25     $j = j - w[i-1][s[i][j]]$
26     $i = i - 1$
27 end
28 $s = jobs$
```

where $s_{ij} \in \{0, 1\}$ denotes whether we select VM configuration j for stage i or not, and N_i denotes the number of configurations in a given stage. t_{ij} denotes the runtime of stage i when using j's configuration, which is obtained from the runtime predictions. Similarly, p_{ij} denotes the cost of running stage i when using j's configuration, which is obtained from the pricing table of the selected cloud vendor. We assume that $z_l(C) := \infty$ if no solution exists (i.e. the total runtime is not sufficient to complete all the stages using the fastest machine configuration).

To solve (2), we implemented a pseudopolynomial solution through dynamic programming utilizing Dudzinski and Walukiewicz approach [48]:

$$z_l(C) = \min \begin{cases} z_{l-1}(C - t_{l1}) + p_{l1} & \text{if } 0 \leq C - t_{l1}, \\ z_{l-1}(C - t_{l2}) + p_{l2} & \text{if } 0 \leq C - t_{l2}, \\ \vdots & \\ z_{l-1}(C - t_{l_{n_l}}) + p_{l_{n_l}} & \text{if } 0 \leq C - t_{l_{n_l}} \end{cases}$$

Algorithm 4 shows a detailed implementation of the above approach. This implementation provides an optimal solution provided that the runtime values are rounded to the nearest integer (second). This is an assumption that we can safely make in our case since cloud machines are billed per second (no fractions).

On-demand vs. Spot VMs. Costs used in our calculations can be obtained from the standard pricing table from public cloud vendors. Some cloud vendors offer a type of VMs called spot (or preemptible) instances. They are offered for a much lower price than the on-demand instances. The way spot instances work is that a user makes a request for a VM having a specific configuration (from Section 5.4.1), and includes a "maximum bid price" indicating the maximum that the user is willing to be charged for that VM configuration. The cloud vendor satisfies the requests of the highest bidders, and periodically recalculates a fair VM price (according to demand) and terminates those instances whose maximum bid is below the new price. In other words, the cloud resource allocator might stop (preempt) these instances if it requires access to those resources for other tenants. This means that an EDA job might be killed prematurely before it completes. However, we can take advantage of the huge cost saving that spot instances offer, by allowing EDA teams to determine "how high" they should bid in order to guarantee that their EDA job will complete before the VM is terminated.

In order to achieve this goal, we utilize the DrAFTS framework proposed in [182]. The framework analyzes published time-series data from a cloud vendor[1] and uses a non-parametric time series analysis method to predict an upper bound on the maximum bid price and a lower bound on the duration the bid will be competitive to prevent a termination due to market price. DrAFTS achieves this by tracking the history of market prices and their associated durations for each point where pricing data is available. Based on this price history, it then builds up a series of runtime durations for which the maximum bid price prediction would have remained above the market price. After that, it uses time-series analysis to predict a lower confidence bound for the requested duration.

In particular, given a time series of price history, a quantile for which a confidence bound should be predicted ($q \in (0, 1)$), and the confidence level of the prediction ($c \in (0, 1)$), each price history's data point in the time series is treated as a Bernoulli trial with a probability q of success. Assuming that data points are independent, the probability of getting exactly k successes is a Binomial distribution with parameters n (number of price history data points) and q (quantile). The probability that no more than k observations are greater than the q^{th} quantile of the distribution is given by:

[1]Amazon web services in this study

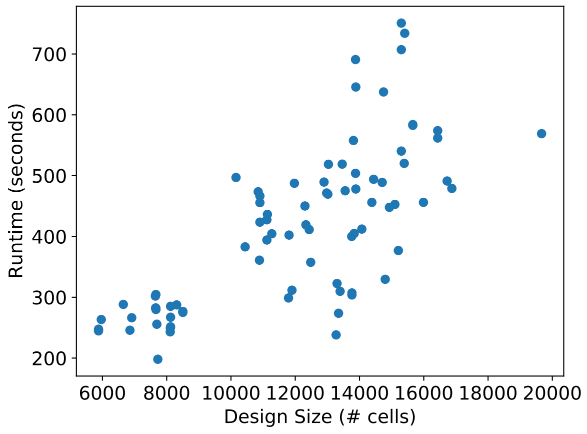


Figure 5.11: Runtime as a function of the design size.

$$\sum_{j=0}^{k} \binom{n}{j} (1-q)^j q^{n-j} \tag{5.3}$$

where taking k to be the largest integer for which this formula is smaller than $1-c$ gives the lower confidence bound. Our integration takes as input the required VM configuration (as recommended from Section 5.4.1) and the estimated runtime for a given job (as recommended from Section 5.4.2). In addition, it takes the required probability for successful completion before the spot VM possibly receives a termination signal by the cloud vendor (e.g. $p >= 0.95$). Then, we recommend a bid price that guarantees a minimum duration for the requested VM, using $c = 0.99$ and a lower confidence bound on the $(1-q)^{th}$ quantile. The higher the probability, the higher the recommended bid price. Obviously, if the recommended bid price is higher than the estimated cost of running the same job on an on-demand VM, a user should always choose the reliable execution of on-demand VMs.

5.5 Empirical Results

In this section, we describe our experimental setup and discuss the results.

Data Management. We implemented EDAAC as an open-source python package[2]. EDAAC connects to a non-structural database engine, MongoDB [131]. MongoDB can be deployed on a central cluster where all jobs in the EDA flow have access to it through EDAAC package. The package serves as a middle-ware providing log file parsers, standardized data model schema and a unified querying engine. These functionalities support the continuous optimization of the design, as well as enable machine learning applications inside and around tools [80]. The package offers a scalable solution to manage the future generation of data-driven EDA applications. We used this

[2]https://pypi.org/project/edaac/

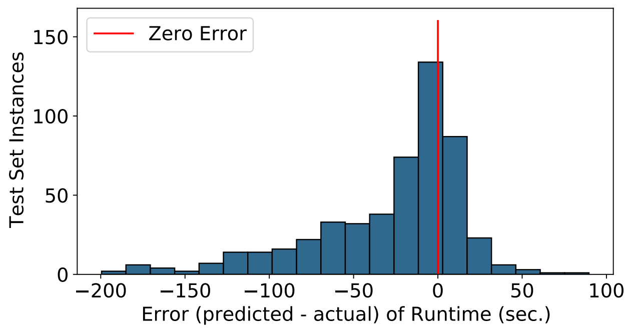


Figure 5.12: Runtime prediction errors. Avg. Error: 13%.

package for collecting runtime data in order to develop our proposed prediction model discussed next.

We demonstrate our predictions on GF 14nm technology node and commercial EDA tools. We implemented our model in Python and utilized Deep Graph Library for training.

Dataset. We use 18 representative benchmarks of different sizes and structures from EPFL benchmark suite [10] and OpenCores [116]. We synthesize each benchmark applying different logic optimizations to generate different netlists. The motivation is to challenge the GCN with netlists that have different physical structures, but perform the same logic function. In addition, the varying size of the netlists (#std_cells, #nets, #IOs) tests how wide and deep the graph convolutions can aggregate information from nodes. We have a total of 330 unique netlists, with 2,640 data points (runtimes) for different machine configurations. These designs range from a few hundred instances to 200k instances. We divide the dataset into training and test groups with 80% and 20% respectively, where netlists of the test set belong to unseen designs in the training set.

Prediction Accuracy. Figure 5.11 shows the runtime of the training set (routing job) as a function of the design size. We observe that designs with similar sizes take significantly different runtimes. In other words, the design size is not a differentiating feature of how long it would take a design to complete a specific job. When fitting a regression model, we found that the average error is 22.7%. While this prediction accuracy might be acceptable in certain use cases, it could make a huge difference when predicting cloud runtimes (and subsequently cloud deployment costs).

As discussed in Section 5.4, a graph-based neural network would be able to capture the inherent complexity of the design that could be a differentiating factor for prediction tasks. We used Adam optimizer [91] (learning rate $= 1e-4$) to train the model parameters. Figure 5.12 shows a histogram of model prediction errors for the routing job. Runtime predictions given a netlist (placement, routing, STA) achieves an average error of 13%. On AIGs (synthesis), the runtime prediction has an average error of 5%. This highlights the capability of graph-based neural networks in capturing the design complexity.

Optimization Results. Referring to Figure 5.2, our optimization module takes as input the predicted runtime for a given EDA job on certain machine size (from Section 5.4.2), and the cost of

Table 5.2: Minimizing total cloud deployment cost subject to a time constraint. The mark (x) denotes the recommended machine configuration. NA denotes Not Achievable.

Task	**vCPUs**	**Runtime (sec.)**	**Cost ($)**	**Total Runtime Constraint (sec.)**			
				10000	**6000**	**5645**	**5000**
Synthesis (general-purpose machine)	**1**	6100	0.1593				
	2	4342	0.1544	x			
	4	3449	0.1878		x		
	8	3352	0.3743			x	
Placement (memory-optimized machine)	**1**	1206	0.0370	x			
	2	905	0.0404				
	4	644	0.0468		x		
	8	519	0.0769			x	
Routing (memory-optimized machine)	**1**	10461	0.3208				
	2	5514	0.2463				
	4	2894	0.2103	x			
	8	1692	0.2506		x	x	
STA (general-purpose machine)	**1**	183	0.0048				
	2	119	0.0042	x	x		
	4	90	0.0049				
	8	82	0.0092			x	
Total Runtime				8561	5904	5645	NA
Minimum Cost				0.4059	0.4894	0.711	NA

running the job on a machine type recommended for that job (as shown in Section 5.4.1). In order to calculate the cost, we obtained the pricing table for the recommended machine types from AWS at the time of this writeup, and calculated the total cost for each EDA job (cost = runtime in hours × cost per hour).

To demonstrate our optimization, we applied different runtime constraints on predictions the of the *sparc_ core* design as shown in Table 5.2. Our algorithm outputs the recommended machine configurations for each task that minimizes the total cost subject to the given total runtime constraint (outputs in Figure 5.2). As we tighten the time constraint, we observe that the algorithm chooses higher machine configurations in some tasks (but not all). A very tight time constraint cannot be met and no solution is presented.

Figure 5.13 shows the cost savings that we get from running our optimization as compared to over-provisioning (using 8 vCPUs in all stages) or under-provisioning (using 1 vCPU in all stages) cloud instances. Under-provisioning cost is also relatively high although the per-second machine costs are cheaper. This is because the runtime of the individual stages are significantly longer. Our optimization offers an average of 35.29% cost saving with minimal overhead to the best runtime.

Furthermore, choosing to run the EDA jobs on spot VMs offers further cost savings. Figure 5.14 shows potential cost savings when bidding for spot VMs. A job completion probability of 95% offers huge cost savings up to 73%, while a job completion probability of 99% offers cost savings up to 44%. Depending on the market price at the time of the request, spot VMs might have a cost close to on-demand VMs; in which case, the EDA team should opt to run the jobs on on-demand VMs.

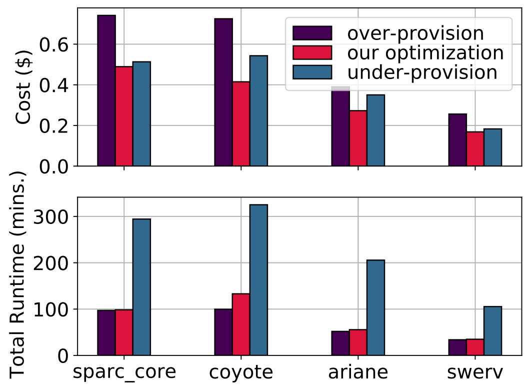


Figure 5.13: Cost savings from running our multi-choice knapsack optimization algorithm. Over-provisioning runs all stages on 8 vCPUs. Under-provisioning runs all stages on 1 vCPUs. Average cost saving: 35.29%

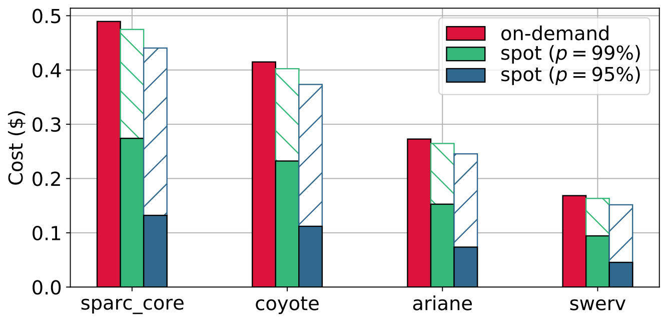


Figure 5.14: Further cost savings from choosing to run EDA jobs on spot VMs. p refers to the probability that the job will complete before the machine is terminated by the cloud provider. Solid bars indicate the minimum cost, and stacked dashed bars indicate the maximum cost; both depend on the market price at the time of the request.

In summary, our method for optimizing EDA jobs on the cloud is inspired by a deeper understanding of the performance characteristics presented in Section 5.4.1. We have offered a complete prediction and optimization pipeline for EDA teams aiming to migrate their workloads to the cloud while keeping the cost minimum and meeting tapeout deadlines. While all experiments are done for 1, 2, 4 and 8 vCPUs (due to license limits), the characterization and methods can be applied on larger designs and more CPU cores. The model and the optimization algorithm are both open-source under BSD-3 license and are available on GitHub [3].

[3]https://github.com/scale-lab/EDAonCloud

5.6 Conclusion

We present a detailed performance characterization of four EDA applications (synthesis, placement, routing, static timing analysis) on different cloud machine configurations. We recommend that placement and routing jobs run on cloud machines with higher memory-to-core ratio. In addition, placement jobs require that the underlying hardware supports Advanced Vector Extensions (AVX) for floating-point operations. Moreover, the performance of routing jobs scales well with more vCPUs added in the multi-tenancy environment. Based on our observations, we propose an integrated framework for analyzing and optimizing EDA flows on the cloud. The framework offers a data management solution and a predictive analytics module. Using this framework, we developed a novel model based on Graph Convolutional Networks (GCNs) that predicts the runtime of a given EDA job under different machine configurations. Our model achieves a runtime prediction accuracy of 87%. Furthermore, we formulated a new optimization problem for deploying EDA applications on the cloud, and presented a pseudo-polynomial optimal solution using a multi-choice knapsack mapping. Our dynamic programming implementation reduces the total deployment cost by 35.29%, while meeting tapeout deadline constraints. Costs costs could further be reduced by up to 73% when using cloud spot instances.

Chapter 6

Fast GPU-native Combinatorial Optimization

6.1 Introduction

Combinatorial optimization problems are a central and crucial class of problems in operations research, computer science, and applied mathematics, characterized by the need to find an optimal solution from a finite or countably infinite set of feasible solutions. These problems arise in numerous real-world applications, including but not limited to scheduling [46, 54], routing [114], resource allocation [41, 34], and network design [62, 1], where the goal is to optimize a given objective function subject to specific constraints. The complexity of combinatorial optimization problems is attributed to their discrete domains, which often leads to exponential growth in the number of potential solutions, rendering them computationally intractable for exact solution methods, particularly in large-scale instances. As such, the development of efficient and effective solution techniques for combinatorial optimization problems remains an ongoing and challenging research endeavor [21].

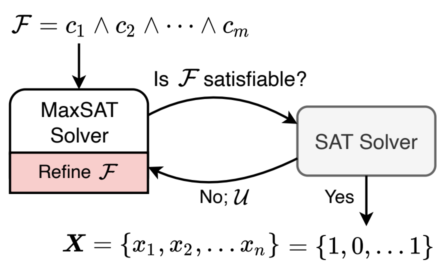
Discrete optimization can be conceptualized as a search process over a discrete solution space, where the primary objective is to identify an optimal or near-optimal solution from a set of feasible solutions. This search process can be approached via various methods, each offering unique trade-offs between computational efficiency and solution quality. Approximation algorithms, for instance, provide provably near-optimal solutions within a specified performance guarantee, typically expressed as a ratio or additive bound relative to the optimal solution [174]. Dynamic programming, on the other hand, leverages a recursive problem decomposition strategy to systematically solve a combinatorial optimization problem by solving its subproblems and storing their solutions to avoid redundant computation [19]. Branch and bound is another widely used approach, which employs a systematic exploration of the solution space using bounds and pruning techniques to eliminate suboptimal solutions, thus narrowing the search and reducing computational effort [3]. Each of these methods offers distinct advantages and limitations, and the choice of the most suitable method depends on the nature and requirements of the specific combinatorial optimization problem at hand.

The recent advancements in training neural networks have opened new avenues for addressing combinatorial optimization problems using deep neural networks. Here, the neural network does not predict solutions for the combinatorial optimization problem, rather the training process of the neural network is the solving process of the problem instance. This line of research aims to leverage the computational power of hardware accelerators, such as GPUs and TPUs, to efficiently search the solution space and discover more optimal solutions. [8] present a new approach that does not rely on data-driven training but instead utilizes backpropagation on a loss function defined by the neural network architecture itself. By reducing the Maximum Independent Set problem (MIS) to a neural network and employing a dataless training scheme, the proposed method updates the network parameters to produce a satisfiable solution. Similarly, [155] use a graph neural network (GNN) [92] to approximately solve canonical NP-hard problems that have a graph structure. By applying a relaxation strategy to the problem Hamiltonian, a differentiable loss function is generated, which is then used to train the graph neural network. The unsupervised training process is followed by a simple projection back to integer variables. These works highlight the potential of utilizing well-maintained and mature infrastructure of training deep neural networks in solving combinatorial optimization, providing scalable and efficient solutions for a wide range of problems.

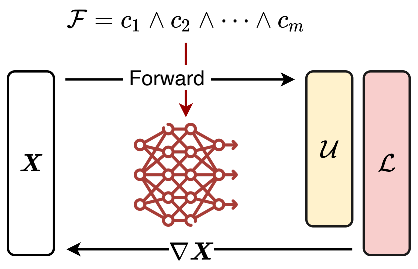
6.2 Motivation

Satisfiability (SAT) and Maximum Satisfiability (MaxSAT) problems represent two prominent classes of combinatorial optimization problems. In the case of SAT, the objective is to determine whether there exists an assignment of truth values to a given set of Boolean variables that satisfies a given set of logical clauses, typically expressed in conjunctive normal form (CNF). When the problem is non-satisfiable, MaxSAT extends the SAT problem by seeking an assignment that maximizes the number of satisfied clauses, rather than trying to satisfy all clauses. MaxSAT problems are of particular interest due to their ability to model optimization scenarios where some degree of constraint violation is tolerable, capturing a broader range of real-world applications [158, 161, 75]. Both SAT and MaxSAT problems have been extensively studied, giving rise to a variety of algorithmic techniques and heuristics aimed at efficiently solving or approximating solutions to these problems [12].

Existing MaxSAT solvers employ a diverse range of techniques to efficiently tackle the Maximum Satisfiability problem. These techniques can generally be classified into two main categories: *complete* and *incomplete* methods. Complete methods, such as Branch and Bound algorithms [105], search exhaustively for an optimal solution, guaranteeing the best possible assignment of truth values to satisfy the maximum number of clauses. Some complete solvers utilize SAT solvers as a core engine, iteratively refining their search space by tightening the upper bound of unsatisfied clauses, as seen in iterative SAT-based MaxSAT algorithms [132, 134]. In contrast, incomplete methods, like local search algorithms, do not guarantee optimality but aim to find high-quality solutions in less time. These methods involve exploring the solution space by iteratively making small changes to the


$\mathcal{F} = c_1 \wedge c_2 \wedge \cdots \wedge c_m$
Is $\mathcal{F}$ satisfiable?
MaxSAT Solver
Refine $\mathcal{F}$
SAT Solver
No; $\mathcal{U}$
Yes
$\boldsymbol{X} = \{x_1, x_2, \ldots x_n\} = \{1, 0, \ldots 1\}$

(a) Existing MaxSAT Solvers



(b) Our torchmSAT Solver

Figure 6.1: Existing MaxSAT solvers depend on a SAT oracle to iteratively evaluate a Boolean formula, $\mathcal{F}$, and update the clauses in $\mathcal{F}$ to reduce the number of unsatisfied clauses, $\mathcal{U}$. Our torchmSAT solver eliminates the need for a SAT oracle and encodes $\mathcal{F}$ in the architecture of a neural network. Maximizing satisfiability is performed using backpropagation on the neurons contributing to $\mathcal{U}$.

current assignment of truth values, guided by various heuristics and neighborhood search strategies [124, 133]. Both complete and incomplete MaxSAT solvers have been successfully applied to real-world problems, with the choice of solver being dependent on the problem size, required optimality guarantees, and available computational resources.

In this thesis, we present a novel MaxSAT solver, called *torchmSAT*, that leverages neural networks and falls within the incomplete category. Departing from the conventional practice of incrementally enhancing existing SAT solvers for MaxSAT resolution, we propose a completely new algorithm developed from scratch. As depicted in Figure 6.1, our method forgoes the necessity for a traditional SAT solver as a fundamental component of the search algorithm in incomplete techniques for MaxSAT. Rather than training a neural network to predict assignments that maximize satisfied clauses, an inherently complex task, we develop a novel neural network architecture with a differentiable loss function for solving MaxSAT. The key intuition is that by relaxing the binary constraint of the problem and allowing the Boolean variables to be represented in a continuous domain, advances in deep learning libraries would allow this optimization to be executed efficiently. Accordingly, the training process can be seen as a process that iteratively explores the solution space, generating progressively improved variable assignments. Consequently, our approach eliminates the need for labeled training data, or the need to call an underlying SAT solver to testify (un)satisfiability. Moreover, since our proposed solver is natively built using a reliable deep learning library [144], we are able to run the solver on GPUs without any change to the data structure, the solving algorithm or the optimization process. Therefore, we investigate the advantages of hardware acceleration for solving MaxSAT, demonstrating that the acceleration of such computations enables the solver to traverse the feasible solution space more rapidly. In essence, torchmSAT presents a fresh approach for solving MaxSAT that could potentially open doors for a new generation of combinatorial optimization solvers.

6.3 Preliminaries

MaxSAT Formulation. In propositional logic, a Boolean formula is composed of Boolean variables, $\mathbf{x} = \{x_1, x_2, \dots x_n\}$, and logical operators, including negations ($\neg$), conjunctions ($\wedge$), and disjunctions ($\vee$). A common representation for Boolean formulas is the conjunctive normal form (CNF), which is structured as a conjunction of multiple clauses, $\boldsymbol{C} = \{c_1, c_2, \dots, c_m\}$. Each clause is a disjunction of literals, where a literal can be either a variable or its negation.

$$\mathcal{F} = c_1 \wedge c_2 \wedge \cdots \wedge c_m$$

$$\mathcal{F} = (x_{1,1} \vee x_{1,2} \vee \cdots \vee x_{1,n_1}) \wedge (x_{2,1} \vee \cdots \vee x_{2,n_2}) \wedge \cdots \wedge (x_{m,1} \vee \cdots \vee x_{m,n_m}) \tag{6.1}$$

This format facilitates the systematic analysis and manipulation of Boolean expressions for various computational tasks. A formula is deemed satisfiable if there is at least one assignment of Boolean variables, x, that satisfies all clauses. In numerous applications, a formula may not be entirely satisfiable, and the objective of MaxSAT solvers is to identify an assignment that satisfies the greatest number of clauses. In this scenario, the number of unsatisfied clauses is referred to as **the cost** of the CNF – a lower cost corresponds to a better assignment. In some applications, (weighted) partial CNF formulas are considered. Clauses in a partial CNF formula are characterized as hard, $\mathcal{H}$, meaning that these must be satisfied, or soft, $\mathcal{S}$, meaning that these are to be satisfied, if at all possible.

SAT Oracle. Existing MaxSAT solvers employ SAT oracles to handle CNF formulas. A SAT oracle is any algorithm that can determine the satisfiability of any given Boolean formula in the conjunctive normal form (CNF). If the formula is satisfiable, the SAT oracle provides a satisfying assignment; if not, it returns additional information such as an unsatisfiable core, $\mathcal{U} \subseteq \mathcal{F}$. The unsatisfiable core, $\mathcal{U}$, represents a subset of $\mathcal{F}$'s clauses that are inherently unsatisfiable. The task of determining the satisfiability of a given CNF is indeed NP-complete [82].

Conflict-Driven Clause Learning (CDCL) is a method used by modern SAT solvers to efficiently search the solution space and resolve conflicts during the search process. State-of-the-art SAT solvers build upon the basic DPLL (Davis-Putnam-Logemann-Loveland) algorithm [139], which employs backtracking and unit propagation, by adding clause learning and non-chronological backtracking [26]. When a CDCL-based solver encounters a conflict, which occurs when the current partial assignment of $\boldsymbol{X}$ leads to an unsatisfiable core $\mathcal{U}$, it analyzes the conflict to generate a new learned clause. This learned clause represents the assignments of variables contributing to the conflict and helps in preventing the solver from introducing similar conflicts in the future. The solver then performs jumping back several levels in the search tree instead of just one, using the learned clause to guide the process. The main advantage of CDCL-based solvers is their ability to learn from conflicts and use that knowledge to prune the search space more effectively. This results in faster and more efficient SAT solving for many real-world problem instances.

MaxSAT Solvers. In an unweighted MaxSAT problem, a typical MaxSAT algorithm proceeds by making several calls to an underlying SAT oracle such as CaDiCaL [53], Glucose [14] or Minisat [168]. The difference amongst algorithms is how they orchestrate the calls to the SAT solver. For example, the RC2 algorithm sends $\mathcal{F}$ to the SAT solver, which reports that $\mathcal{F}$ is unsatisfiable and returns an unsatisfiable core, $\mathcal{U}$. At least one of the clauses of the core will have to be disregarded in order to fix the core. So, the algorithm proceeds by relaxing each clause in $\mathcal{U}$ (i.e. augmenting the clause with a fresh variable called relaxation variable) and constrains the sum of the relaxation variables to be at most one [72]. In other words, the algorithm starts by assuming all clauses can be satisfied and iteratively relaxes this assumption until it finds a satisfying assignment. Similarity, the FM algorithm [119] involves a sequence of calls to an unsatisfiability oracle, each of which generates an unsatisfiable core. The clauses that are part of this unsatisfiable core are then relaxed, followed by the introduction of a new constraint that pertains to the relaxation variables in the formula. On the other hand, the LSU algorithm runs a series of satisfiability oracle calls refining an upper bound on the MaxSAT cost, followed by one unsatisfiability call, which stops the algorithm [133]. In other words, it finds an initial assignment with a suboptimal cost, and iteratively search for better assignments to reduce the cost.

In our work, we present a novel MaxSAT algorithm that does not rely on a SAT oracle during the search process. Instead, it adopts a progressive strategy akin to the one described in [133] and relies entirely on the neural network for identifying unsatisfiable cores, $\mathcal{U}$, and discovering improved solutions through backpropagation.

6.4 Related Work

ML for MaxSAT. Several recent efforts have investigated the integration of machine learning in solving SAT and MaxSAT problems. [157] train a graph neural network classifier to predict satisfiability of random SAT problems. The model learns to search for satisfying assignments during inference for problem instances that are larger than the ones seen during training. Similarly, [107] present a framework for SAT solving utilizing Belief Propagation (BP). They introduce a Graph Neural Network (GNN) architecture that embeds BP in the latent space and use the trained model for marginal inference to obtain satisfying assignments for SAT. Other learning-based methods have been proposed for MaxSAT. For example, [102] propose a method for learning combinatorial optimization problems from contextual examples, which indicate whether solutions are adequate in specific contexts. The framework uses the MaxSAT formulation and considers a specific setting where example solutions and negative solutions are context-specific. In the same way, [22] train MaxSAT models from examples and use a genetic algorithm that decreases the number of evaluations needed to find good models. [122] uses a supervised learning approach to develop an algorithm that can fix Boolean variables based on local information from the Survey Propagation algorithm. In general, this line of research collects training data on MaxSAT instances, and trains a model that generalizes for bigger problems. Our method is different as it does not necessitate the gathering of any training data.

NN-based Combinatorial Optimization. An emerging approach to incorporating machine learning for tackling combinatorial optimization problems involves regarding the neural network's training process as the problem-solving procedure for a given instance. In this paradigm, a neural network is dynamically generated based on the problem instance, and through multiple iterations of forward and backpropagation, a series of learnable parameters embody the ultimate solution to the problem instance. The absence of data requirements for training makes this method particularly attractive for deployment in diverse settings. For instance, [8] introduce a technique that operates on graphs and addresses the Maximum Independent Set (MIS) problem [171]. The core concept involves representing the MIS problem as a single differentiable function, which facilitates the utilization of differentiable solutions. [155] present a Graph Neural Network (GNN)-based solver for approximately solving combinatorial optimization problems by encoding the optimization problem using a Hamiltonian (cost function) and associating binary decision variables with vertices in an undirected graph. A relaxation strategy is applied to generate a differentiable loss function for learning the GNN's node representations. After several iterations, a softmax activation is applied to obtain one-dimensional probabilistic node assignments, which are then mapped back to integer variables using a projection heuristic. Both work can be viewed as a Linear Programming (LP) relaxation of the Mixed-Integer Linear Programming (MILP) formulation of the investigated problems, i.e. Maximum Independent Set (MIS), and Maximim Cut (MaxCUT). While our approach is inspired by the same ideas, we address a different problem (i.e. MaxSAT), and eliminate the use of GNNs.

6.5 Method

Key Idea. Our proposed method, *torchmSAT*, is conceptually inspired by the technique of Linear Programming (LP) relaxations often utilized in the field of Mixed Integer Linear Programming (MILP) [5]. In MILP, certain variables are constrained to take only integer values which makes the optimization problem NP-hard. A common strategy to tackle this issue is to apply LP relaxation, where the integer constraints on the variables are relaxed, allowing them to take on continuous values. Similarly, in our method, we treat the binary variables of the MaxSAT problem as continuous, allowing us to leverage the power of differentiable optimization methods. The challenge we try to address becomes constructing *a single differentiable function* capable of approximating solutions for MaxSAT problems. Once derived, implementing the optimization process using a neural network allows us to capitalize on existing deep learning libraries, and their acceleration capabilities.

In what comes next, we begin by deriving a single differentiable function for MaxSAT using a novel neural network architecture. We describe the key components, their interactions, and their roles in the solving process. Following this, we show how our solver is different in finding unsatisfiable cores and how the solving process proceeds.

The Neural Network Architecture. Our proposed single differentiable function for MaxSAT

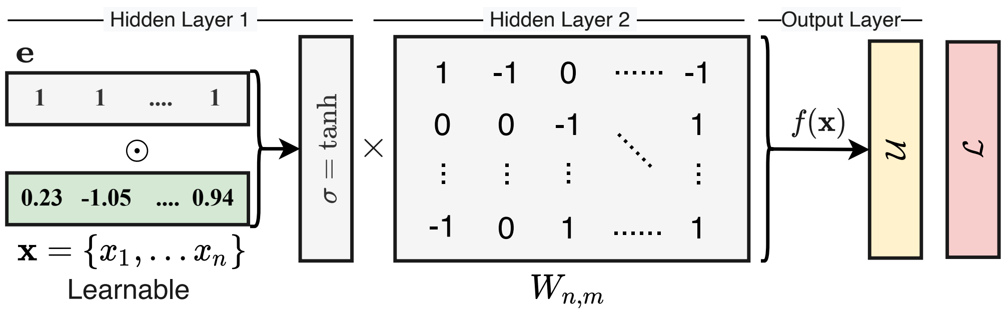


Figure 6.2: Overview of torchmSAT neural network architecture. The learnable vector $\mathbf{x}$ represents the assignments of the Boolean variables in a conjunctive normal form, $\mathcal{F}$. The W matrix is fixed and encodes a given MaxSAT instance represented in conjunctive normal form (CNF), i.e. Equation 6.1, where the rows represent boolean variables, and columns represent clauses. A value of 1 or -1 is assigned if a variable x_i or $\neg x_i$ appears in clause c_j respectively; and 0 otherwise. The output layer calculates the unsatisfied cores, $\mathcal{U}$. The loss function, $\mathcal{L}$, calculates gradients with respect to elements of $\mathbf{x}$ that are contributing to $\mathcal{U}$. The neural network requires no data for training. Rather, the training process functions as the solving process of maximizing the number of satisfied clauses in $\mathcal{F}$.

is modeled as a neural network. Figure 6.2 presents a high-level overview of its architecture. At the heart of our solving algorithm lies a vector, $\mathbf{x}$, which represents the relaxations of the Boolean variables in a given MaxSAT instance (refer to Equation 6.1). The vector $\mathbf{x}$ is the only trainable set of parameters within the neural network. Although $\mathbf{x}$ consists of real numbers, at any given point in time during the solving process, we can project it back into the binary domain to derive the variable assignments. To reverse the relaxation, we interpret $x_i = 1$ when $x_i > 0$, and 0 otherwise. At the onset of the solving process, $\mathbf{x}$ is initialized with random real values. Interpreting the values at this point merely corresponds to assigning random Boolean values to the variables. Throughout the solving process, the values of $\mathbf{x}$ are incrementally updated in a direction that minimizes the loss. This specific design prompts the network to progressively learn to satisfy an increasing number of clauses.

In the context of backpropagation and the chain rule, the derivative of a multiplication operation with respect to its inputs distributes the gradients. Therefore, the first hidden layer performs a point-wise multiplication between $\mathbf{x}$ and a vector of identical length containing all values set to 1, denoted as $\mathbf{e}$. This vector serves to propagate the gradient to the corresponding x_is during the backpropagation process. After that, and to stabilize backpropagation, a tanh activation function is applied to constrain the values of the first layer within the range of -1 to 1. The tanh activation prevents the output of the first layer from reaching excessively high or low values, which could potentially terminate the learning (i.e. solving) prematurely. The rest of the neural network is fixed (i.e. non-learnable) and architected to encode our novel single differentiable function.

The second layer uniquely encodes a given MaxSAT instance. A matrix $W_{n,m}$ is initialized at the start of the solving process, with rows representing variables x_i's and columns representing clauses c_j's. A value of 1 is set if a variable x_i appears in clause c_j, while a value of -1 is set if its negation $\neg x_i$ appears in c_j. All other entries in the matrix are set to 0. This formulation results in our unique differentiable function, $f(\mathbf{x})$, which generates a vector $\mathcal{U}$ of length m:

$$\mathcal{U} = f(\mathbf{x}) = \tanh(\mathbf{e} \odot \mathbf{x}) \cdot W$$

where $\odot$ is element-wise multiplication, and $\cdot$ is a matrix multiplication. The entries of this vector, $\mathcal{U}$, indicate the (un)satisfiability status of each clause, serving as a dual-purpose: an evaluator for SAT and a guide for variable gradients.

Example. Consider the following MaxSAT problem in CNF:

$$\mathcal{F} = (\neg x_1) \wedge (\neg x_2) \wedge (x_1 \vee x_2)$$

This formula is not satisfiable. It is evident that any combination of binary assignments for x_1 and x_2 can satisfy, at most, two clauses. Assume that $\mathbf{x}$ is initialized randomly as $[0.43, 1.27]$, which is interpreted as $x_1 = 1$ and $x_2 = 1$ since both values are positive. The output of the forward pass is:

$$f(\mathbf{x}) = \tanh\left(\begin{bmatrix} 1 & 1 \end{bmatrix} \odot \begin{bmatrix} 0.43 & 1.27 \end{bmatrix}\right) \cdot \begin{bmatrix} -1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} -0.41 & -0.85 & 1.26 \end{bmatrix}$$

Similar to the reverse relaxation operation of $\mathbf{x}$, the output of $f(\mathbf{x})$ is interpreted similarly, i.e. c_j is satisfied if its activation in f is positive, and vice versa. In this case, the output indicates that the first two clauses are unsatisfied, while only the last clause is satisfied. This represents the first role of $f(\mathbf{x})$ as a SAT evaluator. To satisfy the first two clauses, the values of $\mathbf{x}$ need to become negative so that the activations of the first two clauses in f are positive. This is where the role of the loss function comes in. The loss function, $\mathcal{L}$, computes the MSE between $f(x)$ and a zero vector for the unsatisfied clauses. A single iteration of backpropagation updates the values of x_1 and x_2 in the negative direction of the gradient, moving their values closer to zero. For instance, with a learning rate of 0.1, the new values of $\mathbf{x}$ will be $[0.34, 1.23]$. The process continues as the forward-loss-backward loop attempts to satisfy the first two clauses. However, once the first two clauses are satisfied (i.e., $\mathbf{x}$'s are negative, representing a Boolean zero when reversing the relaxation), the third clause becomes unsatisfied. This is when the training process explores other ways of satisfying more clauses.

Unsatisfiable Cores Contrary to conventional MaxSAT solvers discussed in Section 6.4 that rely on a SAT oracle to identify the set of unsatisfied clauses, $\mathcal{U}$, if the formula is determined to be unsatisfiable, our proposed neural network model directly infers the unsatisfied clauses from $f(\mathbf{x})$. Nevertheless, for larger problems, the activations of $f(\mathbf{x})$ alone might not be enough to ascertain satisfiability. The reason is that the matrix multiplication occurring in the first two layers is a non-injective operation. This implies that distinct assignments for $\mathbf{x}$ might yield identical activations

Algorithm 5: torchmSAT Algorithm for Solving MaxSAT

```
Input  : MaxSAT instance in conjunctive normal form (CNF). See Equation 6.1. Time limit ($T$)
Output: Assignment for Boolean variables ($\mathbf{x}$) that maximizes satisfiability
1  Init $\mathbf{x}$ randomly. Construct $W$ from CNF. Init $\mathbf{s}$ where $s_j = -len(c_j)$. Set best_cost = #clauses.
2  agent = Initialize(A2C);
3  while current solving time $< T$ do
4      Run forward pass, $f(\mathbf{x})$. Calculate $\mathbf{x}'$ where $x'_i = 1$ if $x_i > 1$, and 0 otherwise.
5      Calculate unsatisfied clauses, $\mathcal{U} = \mathbf{x}' \cdot W == \mathbf{s}$.
6      if # unsatisfied clauses ($\mathcal{U}$) $<$ best_ cost then
7          Save and output $\mathbf{x}'$, the Boolean assignments of $\mathbf{x}$. Update best_cost.
8      end
9      Calculate loss, $\mathcal{L}$, for $f(\mathbf{x})$ w.r.t variables contributing to the unsatisfied clauses, $\mathcal{U}$.
10     Run backpropagation. Run a single step of optimizer.
11 end
```

for $f(x)$. For instance, the inputs $\mathbf{x} = [0.43, -1.27]$ and $\mathbf{x} = [-1.27, 0.43]$ would result in the same negative activation for the aforementioned third clause, despite the fact that the de-relaxed variable assignments are totally different; one corresponds to $[1, 0]$ while the other corresponds to $[0, 1]$. To overcome this, we use an alternative method to establish (un)satisfiability of clauses. We construct a vector, $\mathbf{s}$, of length m, where each entry corresponds to the negative of the number of variables present in its respective clause, i.e., $s_j = -len(c_j)$. We also project $\mathbf{x}$ onto its Boolean domain, such that $x'_i = 1$ if $x_i > 1$, and 0 otherwise. This makes a clause in $\mathcal{U}$ unsatisfiable *if and only if* $\mathbf{x}' \cdot W = \mathbf{s}$. We use this projection to mask out satisfied clauses in the loss function, and only calculate it for unsatisfied clauses. Consequently, gradients are computed exclusively for those variables that contribute to the unsatisfied clauses, resulting in a more efficient optimization.

The Solving Process. In contrast to conventional MaxSAT solvers, which necessitate invoking a SAT oracle, pausing to await a result, and then coordinating the subsequent call, our algorithm operates differently. torchmSAT operates in a progressive manner, which implies that it incrementally improves the solutions as the search process unfolds. Algorithm 5 shows the main steps of our solving process. The solving process is an iterative procedure that alternates between a forward pass, loss calculation, and backward propagation. In Line 1, we construct the neural network layers based on the given problem instance. While the time limit has not been exhausted, the forward pass, in Line 3, takes the current assignments of the Boolean variables, $\mathbf{x}$, and computes the output of the network, $f(\mathbf{x})$. In addition, it calculates the reversed relaxation of $\mathbf{x}$, denoted as $\mathbf{x}'$, which is used to determine clauses are currently satisfied by the assignment in Line 4. In Lines 5-7, if the current assignment satisfies more clauses than previously found solutions, it outputs this result and updates the value of the best cost (the lower, the better). In Line 8, the loss function calculates the MSE between $f(\mathbf{x})$ and a zero vector of the same length, reflecting the number of unsatisfied clauses. The backpropagation step, in Line 9, adjusts the values of $\mathbf{x}$ according to the gradient calculated by backpropagation, moving the assignments in a direction that reduces the number of unsatisfied clauses. This process continues until all clauses are satisfied, as indicated by a zero loss, or a given time limit is reached.

6.6 Empirical Results

In this section, we present a series of experiments designed to evaluate the performance of our proposed algorithm and compare it with existing state-of-the-art MaxSAT solvers. Our experiments aim to demonstrate the effectiveness of the method in various scenarios and showcase its strengths and limitations. We describe the experimental setup, including the problem instances used, the choice of benchmarks, and the evaluation metrics employed. Additionally, we provide a detailed analysis of the experimental results, highlighting key observations and insights. Through these experiments, we aim to validate the applicability of our approach and its potential impact on combinatorial optimization.

Setup. We use PyTorch (v1.10.2) [144]. No other dependencies are required to run torchmSAT (e.g. no calls to a SAT oracle is made). We use Adam optimizer for backprogation with a learning rate of 1e-4. For reproducibility and extensibility of our work, we use the PySAT toolkit (v0.1.8.dev1) [71] to synthesize hard problem instances and compare against existing methods. The toolkit provides a unified interface to various state-of-the-art SAT and MaxSAT solvers. The experimental results are obtained using a machine with Intel Xeon E5-2680 2x14cores@2.4 GHz, 128GB RAM. For experiments on hardware acceleration, a Tesla P40 GPU is utilized to run torchmSAT.

Evaluation Criteria. We will use a multi-faceted approach to thoroughly assess the performance of our proposed method. First, we will investigate the cost obtained by our solver on problem instances of varying complexity and sizes, and under different time constraints. The imposition of a time limit in this scenario is crucial given the NP-hard nature of MaxSAT, which implies that exploring and assessing the entire feasible region would necessitate exponential time. Secondly, we will calculate the MaxSAT regret which quantifies the difference between the best solution found by any baseline solver and the solution obtained by our solver. Lastly, we will assess the impact of leveraging GPU acceleration on the performance of our solver.

Dataset. While the datasets of the popular MaxSAT evaluations [1] represent a set of non-trivial SAT instances, we opt to test our methods on smaller, yet hard, dataset to validate the applicability of the method. Although our method does not outperform one of the state-of-the-art SAT solvers, torchmSAT offers a completely revamped method to solve MaxSAT problems without the need for a SAT oracle. We employ PySAT to synthesize four representative datasets, encompassing combinatorial principles extensively examined in the context of propositional proof complexity. Specifically, they implement encodings for the pigeonhole principle (PHP) [40], the greater-than (ordering) principle (GT) [99], the mutilated chessboard principle (CB) [7], and the parity principle (PAR) [6]. For each principle, we synthesize 50 MaxSAT instances of increasing sizes where all problem instances are not satisfiable. In other words, there is at least one clause that cannot be satisfied, and the goal of MaxSAT solvers to find a feasible variable assignment that maximizes satisfiability. Synthesis scripts are available in Appendix B.

Comparing with Existing MaxSAT Solvers. As discussed in Section 6.3, existing MaxSAT algorithms proceed by making several calls to an underlying SAT oracle (i.e. solver). In PySAT,

[1] `https://maxsat-evaluations.github.io/`

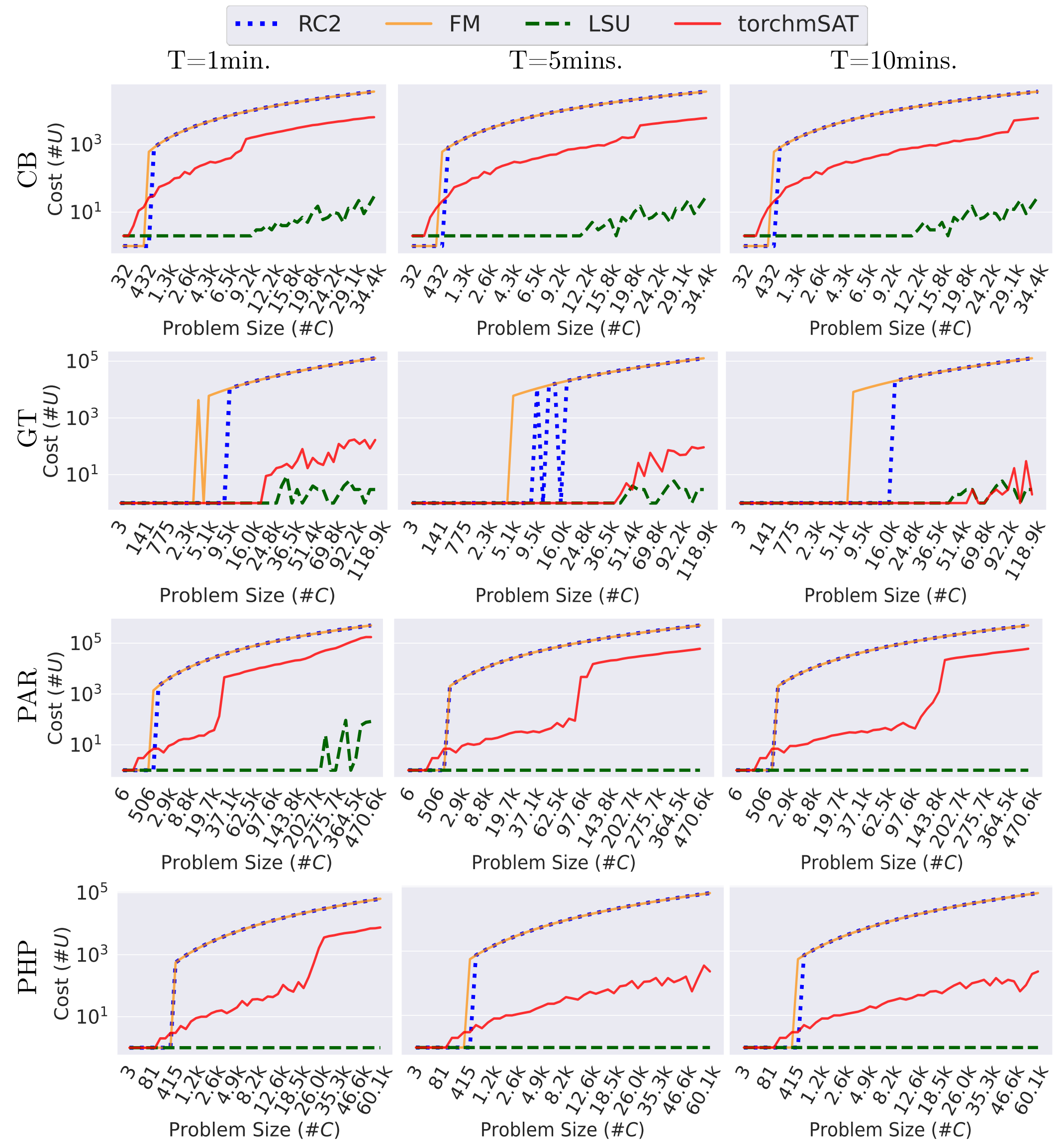


Figure 6.3: Performance of torchmSAT as compared to the state-of-the-art MaxSAT solvers in PySAT [71], namely FM, RC2 and LSU (refer to Section 6.3 for detailed descriptions of algorithms). Each row represents one of the datasets and each column is the time limit given to the solver. Each dataset contains 50 problem instances of increasing size (number of Boolean variables and number of clauses). In the plots, the problem size is parameterized by its total number of clauses on the x-axis. On the y-axis, the cost represents the number of unsatisfiable clauses ($\mathcal{U}$) by the end of the time limit.

both the RC2 and the FM algorithms start their solving process by making a call to the underlying SAT solver, which reports that $\mathcal{F}$ is unsatisfiable and returns an unsatisfiable core, $\mathcal{U}$. The algorithm proceeds by alternating between relaxing clauses and calling the underlying SAT solver. As depicted in Figure 6.3, both methods struggle to find any assignment for moderately large problem

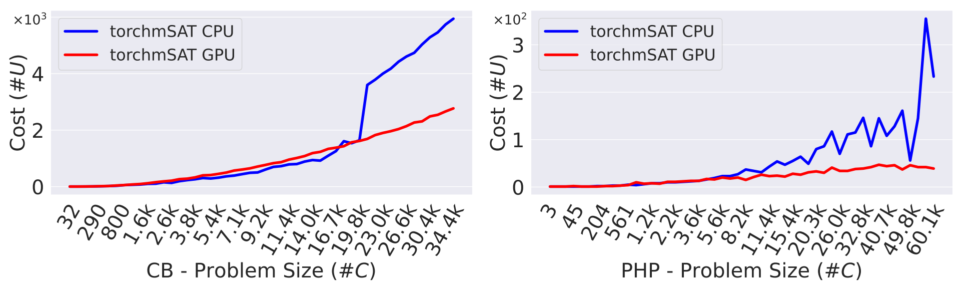


Figure 6.4: Running torchmSAT on CPU vs. GPU, where it is capable of taking advantage of GPU acceleration, and finds better MaxSAT solutions within the same time limit (5mins). For complete results on scalability under different time limits, and on GT and PAR datasets, see Appendix D.

instances. The reason is that their solving process starts by calling the SAT oracle to establish the unsatisfiability of the given formula, which is NP-hard. LSU is the best-performing algorithm since it solves a given problem incrementally, delaying expensive calls to the SAT solver to the end. Our proposed approach, torchmSAT, is designed to be progressive like LSU, but without calling a SAT oracle. As evident from the plots, given more time, it becomes increasingly effective at finding solutions with lower costs. Conversely, the LSU algorithm identifies an initial assignment carrying a suboptimal cost, then incrementally searches for superior assignments to lower the cost. This explains its position as the top-performing algorithm on the dataset. Our proposed method, torchmSAT, excels at progressively generating viable solutions that surpass those of FM and RC2. Indeed, when considering the GT dataset, torchmSAT's performance approaches that of LSU.

In Table 6.1, we compute the average regret of RC2, FM, and torchmSAT relative to LSU, the top-performing MaxSAT solver. For every problem instance, we calculate a solver's regret as the difference between the cost it achieves and the cost LSU achieves. As the results clearly show, torchmSAT generally demonstrates a markedly lower average regret. Raw results for individual problem instances can be found in Appendix C.

GPU Acceleration. The main premise of our method is the novel application of contemporary GPUs and the evolving ecosystem of deep learning libraries and accelerators in solving MaxSAT instances. In this section, we aim to demonstrate that by merely altering where the neural network is initialized, without any modifications to its structure or the solving process described earlier. As shown in Figure 6.4, executing torchmSAT on a GPU yields solutions to larger MaxSAT instances with lower costs within the same time limit. This can be attributed to the enhanced speed of GPU computations, which accelerates the forward-loss-backward loop (Algorithm 5), thereby enabling more extensive exploration of the feasible region of variable assignments.

The use of GPUs in torchmSAT advances MaxSAT solving, offering an accelerated and efficient exploration of the solution space. This acceleration is particularly transformative given that the algorithm's progressive nature means it benefits directly from more rapid computations, enabling it to find better solutions within the same time frame. In contrast, traditional MaxSAT solvers are inherently sequential and cannot take advantage of GPU acceleration.

Table 6.1: The average regret of the solvers. The regret(s, i) of a solver s on instance i is the difference between the cost of the best solution found by s and the cost of best known solution: regret(s, i) $= cost_{s,i} - cost_{best,i}$. Cost is defined as the number of unsatisfied clauses, $\mathcal{U}$. Considering LSU is the the optimal solver (i.e. regret = 0), the table presents the average regret of torchmSAT as compared to FM and RC2. In torchmSAT, regret decreases as the solving time limit increases. See Appendix C.

	RC2			FM			torchmSAT		
Dataset	1min	5mins	10mins	1min	5mins	10mins	1min	5mins	10mins
CB	12396	12396	12396	12408	12408	12408	**1981**	**1496**	**1148**
GT	31852	31245	30425	32552	32467	32206	**30**	**13**	**2**
PAR	130013	130013	130013	130040	130013	130013	**28272**	**13363**	**11777**
PHP	16688	16676	16676	16688	16688	16688	**1384**	**55**	**47**

6.7 Conclusion

Limitations. While our proposed method presents several notable advantages, it also has some limitations. First, it currently only works for unweighted MaxSAT instances, meaning it cannot handle instances where different clauses have different weights or importance. Secondly, the algorithm lacks a definitive stopping criterion for backpropagation unless the optimal solution is zero. This means it can be difficult to determine when the algorithm has reached an optimal solution or when to halt the process except using timeouts. Lastly, the memory requirement for the method is $\mathcal{O}(nm)$, which can be prohibitive for large instances. Although the matrix W is sparse, which could potentially be leveraged to save memory, our current implementation does not take advantage of this sparsity.

Conclusions and Future Work. In conclusion, we have presented a new method for MaxSAT solving that capitalizes on the use of neural networks. Our method, named torchmSAT, is a progressive approach that continually refines and improves its solutions over time. One of the core advantages of torchmSAT is its independence from a SAT oracle, a feature that differentiates it from traditional MaxSAT solvers. This makes our method more self-sufficient and less reliant on external components. Experimental results show that our method outperforms two existing MaxSAT solvers, and is on par with another state-of-the-art solver for small to medium problem sizes. Additionally, torchmSAT is able to benefit from GPU acceleration, allowing for more rapid exploration of feasible solution regions. Despite some limitations, torchmSAT represents a promising step forward in MaxSAT problem solving. For future work, we aim to extend the capabilities of our method to handle weighted MaxSAT instances, develop a clear stopping criteria for backpropagation, and optimize memory usage by leveraging the sparsity of the matrix W. This could lead to a more efficient, versatile and powerful solver that can tackle even more complex problems.

Chapter 7

Summary and Possible Extensions

This dissertation has aimed at improving the solving process of combinatorial optimization problems. The main hypothesis is that looking historical data can indeed improve on classic optimization techniques. We have explored multiple strategies to integrate machine learning models into the solving process of combinatorial optimization problems. We have demonstrated these approaches through various contributions, covering reinforcement learning for logic synthesis, cloud-based optimization of EDA flows, deep metric learning for automatic configuration of Mixed Integer Linear Programming solvers, and a progressive neural network approach for the Maximum Satisfiability Problem. These studies underscore the potential of machine learning to significantly enhance optimization algorithms, whether it be via in-loop, out-of-loop, or end-to-end modeling.

Key findings of our work include successful application of a reinforcement learning agent to minimize area in chip design with specific timing constraints, effective cloud-based optimization of EDA flows that achieved a 35.29% reduction in deployment costs, the use of deep metric learning to improve solutions' costs by up to 38% in MILP, and the creation of a novel neural network model to solve the MaxSAT problem, outperforming two existing solvers. Our efforts thus demonstrate the broad applicability of machine learning techniques in addressing complex optimization problems.

In conclusion, this dissertation showcases the exciting promise of combining machine learning models with optimization problems. Through a multitude of methods and applications, we demonstrate that the adaptability and performance of optimization solvers can be significantly improved. As the challenges of real-world applications continue to grow in complexity, the fusion of machine learning and optimization heralds a new era of innovative problem-solving approaches. In the next sub-sections, we give a brief synopsis of the contributions and discuss future research directions.

7.1 Summary of Contributions

In Chapter 3, we have developed a framework based on reinforcement learning that can explore the optimization space for a specific circuit design and attain high Quality of Result (QoR) autonomously. This problem is modeled within the domain of reinforcement learning, granting the machine a process of learning through trial and error akin to how humans become experts in optimization. This approach essentially converts the complex search space into a "game", where an advantage actor-critic (A2C) agent learns to maximize its reward (minimize area under a delay constraint) by consistently selecting elementary transformations that promise the highest expected reward. We have successfully formulated an And-Inverter Graph (AIG) state representation that effectively delineates the feature set of a design state. Furthermore, we have introduced an innovative multi-objective reward function that directs the agent's exploration process, enabling it to discover a minimal design area while adhering to a delay constraint. Upon evaluating ten representative benchmarks, our suggested methodology outperforms existing methods in terms of effectiveness. DRiLLS demonstrates the feasibility of employing Reinforcement Learning in the combinatorial optimization of hardware circuit designs, which reduces a design area by 13% while meeting delay constraints. It shows immense promise for utilization in related physical synthesis tasks, potentially removing the necessity for human expertise.

In Chapter 4, we have addressed the challenge of optimizing Mixed Integer Linear Programs (MILPs) by predicting configuration parameters that lead to lower-cost solutions. We have demonstrated that the choice of solver configuration significantly impacts solution quality and runtime, even for problem instances with the same number of variables and constraints. By default, using the solver's default configuration often results in suboptimal solutions. To overcome this limitation, our approach focuses on predicting the optimal configuration for unseen problem instances without the time-consuming process of exhaustive search and evaluation. We first examined the correlation of costs between MILP instances from the same distribution but solved using different configurations. Our analysis revealed that instances with similar costs using one configuration tend to exhibit similar costs when using another configuration in the same runtime environment.

Building on this insight, we have proposed a methodology based on Deep Metric Learning to learn similarities among MILP instances that are indicative of their final solution costs. By training a model to project problem instances into a learned metric space, we were able to capture the underlying relationships among instances in terms of their solution quality. This learned embedding space allowed us to leverage information from previously explored configurations to predict suitable parameters for a new problem instance. During the inference phase, when presented with a new MILP instance, our method instantly projects it into the learned metric space using the trained model. By identifying the nearest neighbor instance in the embedding space, we can leverage the configuration parameters from that neighbor to predict the optimal parameters for the new instance. Our empirical results, based on real-world problem benchmarks, clearly demonstrate the effectiveness of our approach. We observed improvements in solution costs of up to 38% compared to existing approaches.

Overall, our study provides a promising solution to the challenge of configuration parameter prediction for MILP solvers. By harnessing the power of deep metric learning, we effectively capture the cost correlation among problem instances and leverage this knowledge to guide the choice of solver configurations. The proposed methodology reduces the time overhead associated with searching and evaluating configurations, while also delivering superior solution quality. These findings open up new avenues for optimizing real-world optimization problems and can have significant practical implications across various domains.

In Chapter 5, we have addressed the challenges faced by semiconductor and electronics companies in leveraging cloud computing for design space exploration in logic synthesis and parameter tuning in physical design. Cloud computing offers scalable compute resources, enabling companies to meet tapeout schedules efficiently. However, deploying Electronic Design Automation (EDA) jobs on the cloud requires a deep understanding of job characteristics in cloud environments, which is currently lacking in public information. To tackle this problem, we first formulated the task of migrating EDA jobs to the cloud. We then conducted a comprehensive characterization of four key EDA applications: synthesis, placement, routing, and static timing analysis. Through our analysis, we established that each EDA job requires specific compute configurations to achieve optimal performance. This insight guided our subsequent efforts.

Based on the observations from our characterization, we introduced a novel model utilizing Graph Convolutional Networks (GCNs) to predict the total runtime of a given EDA stage on different compute configurations. Our model achieved an impressive prediction accuracy of 87%, providing valuable insights for resource allocation and job scheduling. Furthermore, we presented a novel formulation for optimizing cloud deployments to minimize costs while meeting deadline constraints. Our approach utilized a multi-choice knapsack mapping, yielding a pseudo-polynomial optimal solution. By employing this method, we achieved a significant reduction of 35.29% in deployment costs, with minimal impact on the overall runtime. Additionally, we introduced EDA Analytics Central, a cloud-ready solution designed for the continuous optimization of a design throughout the entire EDA flow. This system served as the foundation for building our runtime prediction model, enabling real-time analysis and optimization of EDA jobs.

In summary, this chapter contributes to the advancement of cloud-based EDA by providing insights into the characteristics of EDA jobs in cloud environments. Our proposed model based on GCNs demonstrates high prediction accuracy, facilitating efficient resource allocation. Moreover, our optimization formulation significantly reduces deployment costs by while ensuring deadline compliance. The introduction of EDA Analytics Central enhances the overall EDA flow, enabling continuous optimization and improved design outcomes. These findings have practical implications for semiconductor and electronics companies, empowering them to leverage cloud computing effectively in their design processes and meet critical tapeout schedules.

In Chapter 6, we established a novel approach towards integrating machine learning techniques, specifically a novel neural network architecture, into the solving process of combinatorial optimization algorithms. This approach targets the Maximum Satisfiability Problem (MaxSAT), and rather than enhancing existing solvers, it introduces a single differentiable function capable of approximating solutions for MaxSAT. Remarkably, this function is progressively improved via backpropagation, which eliminates the need for conventional training phases or labeled data, thereby reframing the network training process as the actual solving algorithm. Our proposed methodology not only demonstrates feasibility but also exhibits exceptional performance. Indeed, our experimental findings reveal that this approach significantly outperforms two of the existing MaxSAT solvers, while matching the solution cost of another, without the necessity of using an underlying SAT solver. Furthermore, the potential of leveraging the computational power of GPUs for accelerating these computations has been explored, presenting an enticing avenue for further optimization and speedups.

Given that a multitude of NP-hard problems can be translated into the MaxSAT format, the implications of this methodology are far-reaching. We believe this work to lay the groundwork for a new generation of solvers that are poised to harness the capabilities of neural network hardware acceleration, pushing the boundaries of efficiency and performance in combinatorial optimization problem-solving. Our findings encourage further investigation into the utilization of such techniques for other NP-hard problems, foreseeing a potential paradigm shift in the way we approach these complex computational challenges.

7.2 Potential Future Research

Building upon the contributions of this dissertation, there are several promising directions for future research. These possibilities span across the fields of machine learning, optimization, and hardware acceleration, with potential applications in various real-world scenarios.

Universal Embedding of Optimization Problems: Through all the methods we have developed, it has become apparent that learning problem representations is a critical component in developing ML methods for combinatorial optimization. In Natural Language Processing (NLP), tremendous research efforts led to the globally available embedding APIs (Application Programming Interfaces) for natural language text. Developers and researchers alike are able to build numerous downstream applications using the learned embeddings. In combinatorial optimization, there is a need for such an embedding API that takes an optimization problem in its most general form (e.g. integer programs) and produces meaningful embeddings. Such an API could be "fine-tuned" for different problem domains and downstream applications. Investigating this direction could lead to revolutionary ML-based methods for combinatorial optimization.

Defining Problem Instance Similarities: Being able to accurately define a subjective measure for defining optimization problem similarities would open doors for systems that can be practically deployed on production environments. However, our definition of similarity has been based on solving problem instances to get some accurate measures about the solution quality within a time limit. This could introduce friction in training such models. Therefore, there is an immense need to find more efficient ways to define similarities between problem instances. This would lead to being able to train larger models on larger datasets, ultimately improving downstream applications that use such similarities (e.g. hyperparaemter tuning).

In-place Reinforcement Learning for Optimization: With our initial success in applying reinforcement learning (RL) in the context of chip design optimization, there is a fiction in applying the model to large problem instances due to the communication overhead between the RL agent and the logic synthesis environment. Building an in-place RL agent within the logic synthesis environment would make such models hugely practical to incorporate in practice. This might require a fresh look on how we design RL algorithms in a new format different from how they are currently established (separate model and game environment).

Hardware Acceleration: As we demonstrated with torchmSAT, leveraging ML acceleration hardware can provide significant computational benefits. Future work could explore the use of large-scale distributed acceleration technologies, such as GPU clusters. These infrastructures offer a fundamentally advanced computing paradigm that could potentially lead to breakthroughs in optimization problem solving. These future directions demonstrate the vast landscape for further exploration, promising exciting opportunities to expand the horizons of machine learning in optimization problems.

Appendix A

Data Management in Metric Learning

In order to offer a seamless integration of our method in existing environments, a data store is required to save the results from the offline configuration space search. In this work, we use MongoDB[1] for that purpose. For each benchmark, we create a collection that contains records for each problem instance in that dataset. Listing A.1 shows the schema used for each instance. It keeps track of configurations explored for that instance along with their costs. In addition, it records the embedding vector of the instance in order to be searched later with the nearest neighbor algorithm. The parameters presented in the listing are the ones that were used for the configuration space exploration using SMAC [111]. A detailed description of the definition of these parameters can be found in their official documentation[2]. As discussed in Section 4.6, the metric learning approach does not limit the number of configuration parameters explored offline. It also does not limit which parameters are explored since it focuses on learning an embedding space where similarity between instances can be quantified reliably. Thus, it is possible to learn a model for similarity once and keep expanding the offline configuration space search without requiring to re-train the model.

```
instance_record = {
    "configs": [
        {
            "seed": 0,
            "cost": 0,
            "time": 0,
            "params": {
                "branching/scorefunc": "s",
                "branching/scorefac": 0.167,
                "branching/preferbinary": False,
                "branching/clamp": 0.2,
                "branching/midpull": 0.75,
                "branching/midpullreldomtrig": 0.5,
                "branching/lpgainnormalize": "s",
```

[1]Link: https://www.mongodb.com/

[2]Link: https://www.scipopt.org/doc/html/PARAMETERS.php

```
                "lp/pricing": "l",
                "lp/colagelimit": 10,
                "lp/rowagelimit": 10,
                "nodeselection/childsel": "h",
                "separating/minortho": 0.9,
                "separating/minorthoroot": 0.9,
                "separating/maxcuts": 100,
                "separating/maxcutsroot": 2000,
                "separating/cutagelimit": 80,
                "separating/poolfreq": 10
            }
        },
        :   # all configurations explored offline
    ],
    "bipartite": {
        "vars_features": [...],
        "cons_features": [...],
        "edge_features": [...]
    },
    "embedding": [...]
}
```

Listing A.1: Problem Instance Record

Appendix B

Dataset Details in torchmSAT

We employ PySAT to synthesize four representative datasets, encompassing combinatorial principles extensively examined in the context of propositional proof complexity. Specifically, they implement encodings for the pigeonhole principle (PHP) [40], the greater-than (ordering) principle (GT) [99], the mutilated chessboard principle (CB) [7], and the parity principle (PAR) [6]. For each principle, we synthesize 50 MaxSAT instances of increasing sizes where all problem instances are not satisfiable. In other words, there is at least one clause that cannot be satisfied, and the goal of MaxSAT solvers to find a feasible variable assignment that maximizes satisfiability.

Below is the synthesis script.

```python
from pysat.examples.genhard import CB, GT, PAR, PHP


def gen_php():
    for n_holes in range(1, 51):
        cnf = PHP(n_holes)
        cnf.to_file(fname=f"data/php/{cnf.nv}_{len(cnf.clauses)}_{n_holes}.zip")

def gen_cb():
    for size in range(1, 51):
        cnf = CB(size)
        cnf.to_file(fname=f"data/cb/{cnf.nv}_{len(cnf.clauses)}_{size}.zip")

def gen_gt():
    for size in range(1, 51):
        cnf = GT(size)
        cnf.to_file(fname=f"data/gt/{cnf.nv}_{len(cnf.clauses)}_{size}.zip")

def gen_par():
    for size in range(1, 51):
        cnf = PAR(size)
        cnf.to_file(fname=f"data/par/{cnf.nv}_{len(cnf.clauses)}_{size}.zip")
```

Listing B.1: Synthesis script for the datasets

For each dataset, we provide detailed statistics on the number of Boolean variables and clauses in Table B.1.

Table B.1: Number of Boolean variables and clauses in each problem instance in the datasets. This indicates the difficulty level of solving each instance.

	CB [7]		**GT [99]**		**PAR [6]**		**PHP [40]**	
	# vars	# clauses	# vars	# clauses	# vars	# clauses	# vars	# clauses
1	20	32	2	3	3	6	2	3
2	56	90	6	12	10	35	6	9
3	108	176	12	34	21	112	12	22
4	176	290	20	75	36	261	20	45
5	260	432	30	141	55	506	30	81
6	360	602	42	238	78	871	42	133
7	476	800	56	372	105	1380	56	204
8	608	1026	72	549	136	2057	72	297
9	756	1280	90	775	171	2926	90	415
10	920	1562	110	1056	210	4011	110	561
11	1100	1872	132	1398	253	5336	132	738
12	1296	2210	156	1807	300	6925	156	949
13	1508	2576	182	2289	351	8802	182	1197
14	1736	2970	210	2850	406	10991	210	1485
15	1980	3392	240	3496	465	13516	240	1816
16	2240	3842	272	4233	528	16401	272	2193
17	2516	4320	306	5067	595	19670	306	2619
18	2808	4826	342	6004	666	23347	342	3097
19	3116	5360	380	7050	741	27456	380	3630
20	3440	5922	420	8211	820	32021	420	4221
21	3780	6512	462	9493	903	37066	462	4873
22	4136	7130	506	10902	990	42615	506	5589
23	4508	7776	552	12444	1081	48692	552	6372
24	4896	8450	600	14125	1176	55321	600	7225
25	5300	9152	650	15951	1275	62526	650	8151
26	5720	9882	702	17928	1378	70331	702	9153
27	6156	10640	756	20062	1485	78760	756	10234
28	6608	11426	812	22359	1596	87837	812	11397
29	7076	12240	870	24825	1711	97586	870	12645
30	7560	13082	930	27466	1830	108031	930	13981
31	8060	13952	992	30288	1953	119196	992	15408
32	8576	14850	1056	33297	2080	131105	1056	16929
33	9108	15776	1122	36499	2211	143782	1122	18547
34	9656	16730	1190	39900	2346	157251	1190	20265
35	10220	17712	1260	43506	2485	171536	1260	22086
36	10800	18722	1332	47323	2628	186661	1332	24013
37	11396	19760	1406	51357	2775	202650	1406	26049
38	12008	20826	1482	55614	2926	219527	1482	28197
39	12636	21920	1560	60100	3081	237316	1560	30460
40	13280	23042	1640	64821	3240	256041	1640	32841
41	13940	24192	1722	69783	3403	275726	1722	35343
42	14616	25370	1806	74992	3570	296395	1806	37969
43	15308	26576	1892	80454	3741	318072	1892	40722
44	16016	27810	1980	86175	3916	340781	1980	43605
45	16740	29072	2070	92161	4095	364546	2070	46621
46	17480	30362	2162	98418	4278	389391	2162	49773
47	18236	31680	2256	104952	4465	415340	2256	53064
48	19008	33026	2352	111769	4656	442417	2352	56497
49	19796	34400	2450	118875	4851	470646	2450	60075
50	20600	35802	2550	126276	5050	500051	2550	63801

Appendix C

Raw Results in torchmSAT

In Table C.1, we provide the raw results discussed in the Experiments Section 6.6. It is a detailed expansion of Table 6.1. The goal is to give a deeper look into Figure 6.3, and see how numbers relate to each other.

Table C.1: Raw results for costs (i.e. number of unsatisfied clauses) obtained by different solvers.

		RC2			FM			LSU			torchmSAT		
		1m	5m	10m	1m	5m	10m	1m	5m	10m	1m	5m	10m
CB	1	1	1	1	1	1	1	2	2	2	2	2	2
	2	1	1	1	1	1	1	2	2	2	2	2	2
	3	1	1	1	1	1	1	2	2	2	4	2	2
	4	1	1	1	1	1	1	2	2	2	11	7	6
	5	1	1	1	1	1	1	2	2	2	14	13	13
	6	1	1	1	602	602	602	2	2	2	27	21	21
	7	800	800	800	800	800	800	2	2	2	30	30	30
	8	1026	1026	1026	1026	1026	1026	2	2	2	55	54	53
	9	1280	1280	1280	1280	1280	1280	2	2	2	63	63	63
	10	1562	1562	1562	1562	1562	1562	2	2	2	74	74	74
	11	1872	1872	1872	1872	1872	1872	2	2	2	99	99	99
	12	2210	2210	2210	2210	2210	2210	2	2	2	105	105	105
	13	2576	2576	2576	2576	2576	2576	2	2	2	153	153	153
	14	2970	2970	2970	2970	2970	2970	2	2	2	132	132	132
	15	3392	3392	3392	3392	3392	3392	2	2	2	193	193	193
	16	3842	3842	3842	3842	3842	3842	2	2	2	230	230	230
	17	4320	4320	4320	4320	4320	4320	2	2	2	260	260	260
	18	4826	4826	4826	4826	4826	4826	2	2	2	304	304	304
	19	5360	5360	5360	5360	5360	5360	2	2	2	288	288	288
	20	5922	5922	5922	5922	5922	5922	2	2	2	317	317	317
	21	6512	6512	6512	6512	6512	6512	2	2	2	364	364	364
	22	7130	7130	7130	7130	7130	7130	2	2	2	391	391	391
	23	7776	7776	7776	7776	7776	7776	2	2	2	550	441	441
	24	8450	8450	8450	8450	8450	8450	2	2	2	661	489	489
	25	9152	9152	9152	9152	9152	9152	2	2	2	1426	502	502
	26	9882	9882	9882	9882	9882	9882	2	2	2	1569	605	605
	27	10640	10640	10640	10640	10640	10640	3	2	2	1687	698	698
	28	11426	11426	11426	11426	11426	11426	3	2	2	1833	725	725
	29	12240	12240	12240	12240	12240	12240	4	2	2	1995	787	787
	30	13082	13082	13082	13082	13082	13082	3	3	3	2116	799	799
	31	13952	13952	13952	13952	13952	13952	5	5	5	2280	887	887
	32	14850	14850	14850	14850	14850	14850	4	3	3	2423	941	941
	33	15776	15776	15776	15776	15776	15776	4	4	3	2614	920	920
	34	16730	16730	16730	16730	16730	16730	6	6	5	2790	1093	1050
	35	17712	17712	17712	17712	17712	17712	5	2	2	3030	1257	1117
	36	18722	18722	18722	18722	18722	18722	7	7	7	3227	1608	1265
	37	19760	19760	19760	19760	19760	19760	5	5	5	3433	1536	1234
	38	20826	20826	20826	20826	20826	20826	10	10	9	3669	1641	1364
	39	21920	21920	21920	21920	21920	21920	15	15	15	3797	3599	1426
	40	23042	23042	23042	23042	23042	23042	6	6	6	4036	3785	1502
	41	24192	24192	24192	24192	24192	24192	7	7	7	4268	4002	1688
	42	25370	25370	25370	25370	25370	25370	10	10	10	4465	4176	1849
	43	26576	26576	26576	26576	26576	26576	9	9	9	4701	4424	2097
	44	27810	27810	27810	27810	27810	27810	5	5	5	4855	4606	2242
	45	29072	29072	29072	29072	29072	29072	13	13	13	5109	4738	2307
	46	30362	30362	30362	30362	30362	30362	12	12	12	5449	5034	5025
	47	31680	31680	31680	31680	31680	31680	22	22	22	5621	5286	5243
	48	33026	33026	33026	33026	33026	33026	9	9	9	5857	5460	5460
	49	34400	34400	34400	34400	34400	34400	16	16	16	6182	5740	5736
	50	35802	35802	35802	35802	35802	35802	29	29	29	6297	5945	5936

Continued on next page

		RC2			FM			LSU			torchmSAT		
		1m	5m	10m	1m	5m	10m	1m	5m	10m	1m	5m	10m
GT	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1	1	1	1	1	1
	3	1	1	1	1	1	1	1	1	1	1	1	1
	4	1	1	1	1	1	1	1	1	1	1	1	1
	5	1	1	1	1	1	1	1	1	1	1	1	1
	6	1	1	1	1	1	1	1	1	1	1	1	1
	7	1	1	1	1	1	1	1	1	1	1	1	1
	8	1	1	1	1	1	1	1	1	1	1	1	1
	9	1	1	1	1	1	1	1	1	1	1	1	1
	10	1	1	1	1	1	1	1	1	1	1	1	1
	11	1	1	1	1	1	1	1	1	1	1	1	1
	12	1	1	1	1	1	1	1	1	1	1	1	1
	13	1	1	1	1	1	1	1	1	1	1	1	1
	14	1	1	1	1	1	1	1	1	1	1	1	1
	15	1	1	1	1	1	1	1	1	1	1	1	1
	16	1	1	1	4233	1	1	1	1	1	1	1	1
	17	1	1	1	1	1	1	1	1	1	1	1	1
	18	1	1	1	6004	6004	1	1	1	1	1	1	1
	19	1	1	1	7050	7050	1	1	1	1	1	1	1
	20	1	1	1	8211	8211	8211	1	1	1	1	1	1
	21	1	1	1	9493	9493	9493	1	1	1	1	1	1
	22	10902	10902	1	10902	10902	10902	1	1	1	1	1	1
	23	12444	1	1	12444	12444	12444	1	1	1	1	1	1
	24	14125	14125	1	14125	14125	14125	1	1	1	1	1	1
	25	15951	15951	1	15951	15951	15951	1	1	1	1	1	1
	26	17928	1	1	17928	17928	17928	1	1	1	1	1	1
	27	20062	20062	20062	20062	20062	20062	1	1	1	1	1	1
	28	22359	22359	22359	22359	22359	22359	1	1	1	1	1	1
	29	24825	24825	24825	24825	24825	24825	1	1	1	9	1	1
	30	27466	27466	27466	27466	27466	27466	1	1	1	10	1	1
	31	30288	30288	30288	30288	30288	30288	1	1	1	17	1	1
	32	33297	33297	33297	33297	33297	33297	4	1	1	19	1	1
	33	36499	36499	36499	36499	36499	36499	9	1	1	24	1	1
	34	39900	39900	39900	39900	39900	39900	1	1	1	17	1	1
	35	43506	43506	43506	43506	43506	43506	3	1	1	31	1	1
	36	47323	47323	47323	47323	47323	47323	1	1	1	80	2	1
	37	51357	51357	51357	51357	51357	51357	2	2	2	17	5	1
	38	55614	55614	55614	55614	55614	55614	4	4	2	39	3	1
	39	60100	60100	60100	60100	60100	60100	3	3	3	26	26	1
	40	64821	64821	64821	64821	64821	64821	3	3	3	22	9	3
	41	69783	69783	69783	69783	69783	69783	1	1	1	59	59	1
	42	74992	74992	74992	74992	74992	74992	1	1	1	28	27	1
	43	80454	80454	80454	80454	80454	80454	2	2	2	121	13	2
	44	86175	86175	86175	86175	86175	86175	4	4	4	85	73	3
	45	92161	92161	92161	92161	92161	92161	6	6	6	157	67	2
	46	98418	98418	98418	98418	98418	98418	3	3	3	173	49	3
	47	104952	104952	104952	104952	104952	104952	3	3	3	121	51	17
	48	111769	111769	111769	111769	111769	111769	1	1	1	167	94	1
	49	118875	118875	118875	118875	118875	118875	3	3	3	84	84	30
	50	126276	126276	126276	126276	126276	126276	3	3	3	167	92	2

Continued on next page

		RC2			FM			LSU			torchmSAT		
		1m	5m	10m	1m	5m	10m	1m	5m	10m	1m	5m	10m
PAR	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1	1	1	1	1	1
	3	1	1	1	1	1	1	1	1	1	1	1	1
	4	1	1	1	1	1	1	1	1	1	3	1	1
	5	1	1	1	1	1	1	1	1	1	3	3	3
	6	1	1	1	1	1	1	1	1	1	5	3	3
	7	1	1	1	1380	1	1	1	1	1	7	7	7
	8	2057	2057	2057	2057	2057	2057	1	1	1	7	7	7
	9	2926	2926	2926	2926	2926	2926	1	1	1	5	5	5
	10	4011	4011	4011	4011	4011	4011	1	1	1	9	9	9
	11	5336	5336	5336	5336	5336	5336	1	1	1	11	11	9
	12	6925	6925	6925	6925	6925	6925	1	1	1	15	10	10
	13	8802	8802	8802	8802	8802	8802	1	1	1	17	11	11
	14	10991	10991	10991	10991	10991	10991	1	1	1	17	17	15
	15	13516	13516	13516	13516	13516	13516	1	1	1	19	17	17
	16	16401	16401	16401	16401	16401	16401	1	1	1	23	19	19
	17	19670	19670	19670	19670	19670	19670	1	1	1	23	23	23
	18	23347	23347	23347	23347	23347	23347	1	1	1	32	28	25
	19	27456	27456	27456	27456	27456	27456	1	1	1	38	32	28
	20	32021	32021	32021	32021	32021	32021	1	1	1	134	33	31
	21	37066	37066	37066	37066	37066	37066	1	1	1	4553	30	30
	22	42615	42615	42615	42615	42615	42615	1	1	1	5085	34	34
	23	48692	48692	48692	48692	48692	48692	1	1	1	5745	31	31
	24	55321	55321	55321	55321	55321	55321	1	1	1	6344	38	38
	25	62526	62526	62526	62526	62526	62526	1	1	1	7490	45	40
	26	70331	70331	70331	70331	70331	70331	1	1	1	8380	72	46
	27	78760	78760	78760	78760	78760	78760	1	1	1	9294	51	38
	28	87837	87837	87837	87837	87837	87837	1	1	1	10574	107	55
	29	97586	97586	97586	97586	97586	97586	1	1	1	11300	89	73
	30	108031	108031	108031	108031	108031	108031	1	1	1	12677	4705	53
	31	119196	119196	119196	119196	119196	119196	1	1	1	14268	4693	44
	32	131105	131105	131105	131105	131105	131105	1	1	1	15063	15063	126
	33	143782	143782	143782	143782	143782	143782	1	1	1	17086	17086	258
	34	157251	157251	157251	157251	157251	157251	1	1	1	18917	18917	466
	35	171536	171536	171536	171536	171536	171536	1	1	1	20864	20864	1249
	36	186661	186661	186661	186661	186661	186661	1	1	1	21759	21759	21759
	37	202650	202650	202650	202650	202650	202650	1	1	1	24458	24458	24458
	38	219527	219527	219527	219527	219527	219527	1	1	1	29072	26754	26754
	39	237316	237316	237316	237316	237316	237316	1	1	1	37080	28506	28506
	40	256041	256041	256041	256041	256041	256041	1	1	1	44488	30846	30846
	41	275726	275726	275726	275726	275726	275726	24	1	1	51984	32926	32926
	42	296395	296395	296395	296395	296395	296395	1	1	1	57826	34824	34824
	43	318072	318072	318072	318072	318072	318072	1	1	1	63468	37285	37285
	44	340781	340781	340781	340781	340781	340781	12	1	1	75878	41174	41174
	45	364546	364546	364546	364546	364546	364546	92	1	1	92245	43744	43744
	46	389391	389391	389391	389391	389391	389391	1	1	1	111230	46042	46042
	47	415340	415340	415340	415340	415340	415340	3	1	1	130513	49045	49045
	48	442417	442417	442417	442417	442417	442417	58	1	1	157500	52467	52467
	49	470646	470646	470646	470646	470646	470646	77	1	1	174228	55854	55854
	50	500051	500051	500051	500051	500051	500051	83	1	1	173900	60406	60406

Continued on next page

		RC2			FM			LSU			torchmSAT		
		1m	5m	10m	1m	5m	10m	1m	5m	10m	1m	5m	10m
PHP	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1	1	1	1	1	1
	3	1	1	1	1	1	1	1	1	1	1	1	1
	4	1	1	1	1	1	1	1	1	1	1	1	1
	5	1	1	1	1	1	1	1	1	1	1	1	1
	6	1	1	1	1	1	1	1	1	1	1	1	1
	7	1	1	1	1	1	1	1	1	1	2	2	2
	8	1	1	1	1	1	1	1	1	1	2	2	2
	9	1	1	1	1	1	1	1	1	1	3	3	3
	10	561	1	1	561	561	561	1	1	1	3	3	3
	11	738	738	738	738	738	738	1	1	1	5	5	5
	12	949	949	949	949	949	949	1	1	1	4	4	4
	13	1197	1197	1197	1197	1197	1197	1	1	1	7	6	6
	14	1485	1485	1485	1485	1485	1485	1	1	1	9	8	8
	15	1816	1816	1816	1816	1816	1816	1	1	1	10	8	8
	16	2193	2193	2193	2193	2193	2193	1	1	1	10	10	10
	17	2619	2619	2619	2619	2619	2619	1	1	1	13	10	10
	18	3097	3097	3097	3097	3097	3097	1	1	1	15	11	11
	19	3630	3630	3630	3630	3630	3630	1	1	1	16	12	12
	20	4221	4221	4221	4221	4221	4221	1	1	1	13	13	13
	21	4873	4873	4873	4873	4873	4873	1	1	1	16	16	15
	22	5589	5589	5589	5589	5589	5589	1	1	1	20	19	19
	23	6372	6372	6372	6372	6372	6372	1	1	1	32	23	17
	24	7225	7225	7225	7225	7225	7225	1	1	1	23	23	22
	25	8151	8151	8151	8151	8151	8151	1	1	1	36	27	27
	26	9153	9153	9153	9153	9153	9153	1	1	1	37	37	30
	27	10234	10234	10234	10234	10234	10234	1	1	1	34	34	34
	28	11397	11397	11397	11397	11397	11397	1	1	1	45	31	31
	29	12645	12645	12645	12645	12645	12645	1	1	1	43	43	35
	30	13981	13981	13981	13981	13981	13981	1	1	1	54	54	43
	31	15408	15408	15408	15408	15408	15408	1	1	1	106	47	42
	32	16929	16929	16929	16929	16929	16929	1	1	1	75	55	55
	33	18547	18547	18547	18547	18547	18547	1	1	1	64	64	57
	34	20265	20265	20265	20265	20265	20265	1	1	1	130	49	49
	35	22086	22086	22086	22086	22086	22086	1	1	1	84	80	66
	36	24013	24013	24013	24013	24013	24013	1	1	1	191	86	85
	37	26049	26049	26049	26049	26049	26049	1	1	1	538	117	103
	38	28197	28197	28197	28197	28197	28197	1	1	1	1651	70	70
	39	30460	30460	30460	30460	30460	30460	1	1	1	3596	111	100
	40	32841	32841	32841	32841	32841	32841	1	1	1	3985	115	111
	41	35343	35343	35343	35343	35343	35343	1	1	1	4231	146	128
	42	37969	37969	37969	37969	37969	37969	1	1	1	4607	86	86
	43	40722	40722	40722	40722	40722	40722	1	1	1	4936	145	145
	44	43605	43605	43605	43605	43605	43605	1	1	1	5171	108	98
	45	46621	46621	46621	46621	46621	46621	1	1	1	5399	128	128
	46	49773	49773	49773	49773	49773	49773	1	1	1	5924	161	115
	47	53064	53064	53064	53064	53064	53064	1	1	1	6370	56	56
	48	56497	56497	56497	56497	56497	56497	1	1	1	7023	145	88
	49	60075	60075	60075	60075	60075	60075	1	1	1	7133	355	194
	50	63801	63801	63801	63801	63801	63801	1	1	1	7552	233	233

Appendix D

GPU Acceleration in torchmSAT

The application of GPU acceleration in computational tasks offers remarkable advantages, particularly when it comes to large-scale computations, such as those involved in combinatorial optimization. The parallel processing capabilities of GPUs allow for significant speed enhancements, enabling faster computation and consequently more rapid exploration of solution spaces. For our torchmSAT method, which is reliant on neural networks and iterative computation, GPU acceleration can significantly enhance the efficiency of the forward-loss-backward loop, enabling more rapid exploration of the feasible region of variable assignments. This implies that given a time limit, we can expect torchmSAT running on a GPU to find solutions of lower cost than it would when executed on a CPU. Future exploration of GPU acceleration could bring about further advancements in MaxSAT solvers and other similar combinatorial optimization problems, leading to more efficient and effective solutions. Figures D.1, D.2 and D.3 show the scalability of our method under different time limits. As expected, the use of GPUs pays off on larger problem instances, especially when the time limit is tight.

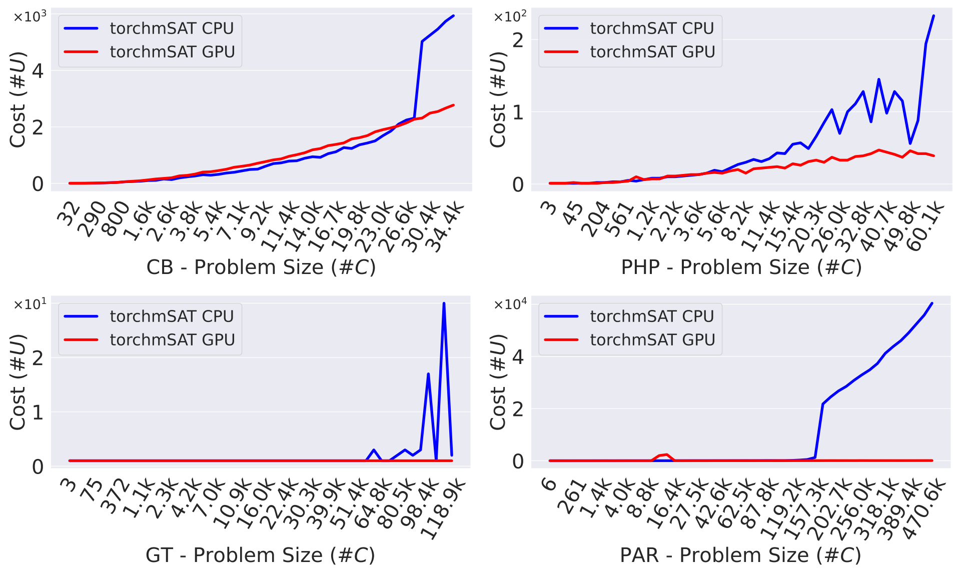


Figure D.1: Running torchmSAT on CPU vs. GPU, where it is capable of taking advantage of GPU acceleration, and finds better MaxSAT solutions within the same time limit **(10mins)**.

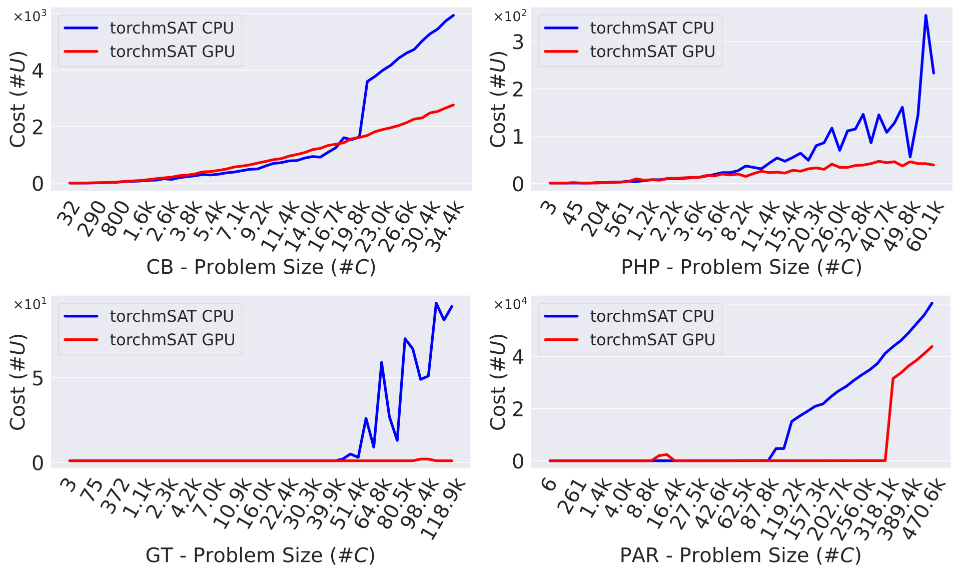


Figure D.2: Running torchmSAT on CPU vs. GPU, where it is capable of taking advantage of GPU acceleration, and finds better MaxSAT solutions within the same time limit **(5mins)**.

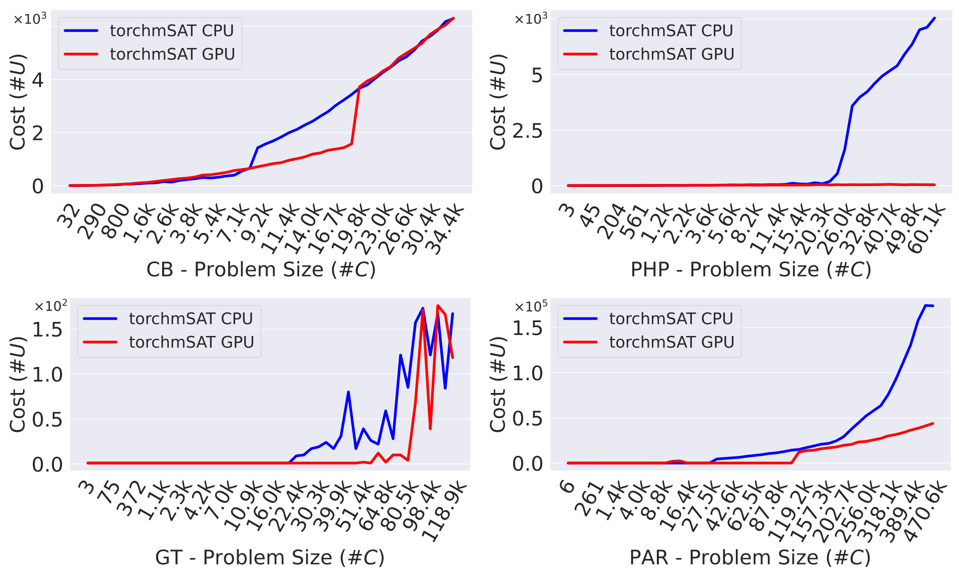


Figure D.3: Running torchmSAT on CPU vs. GPU, where it is capable of taking advantage of GPU acceleration, and finds better MaxSAT solutions within the same time limit **(1min)**.

Bibliography

[1] On the minimization of traffic congestion in road networks with tolls. *Annals of Operations Research*, 249:119–139, 2017.

[2] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[3] Tobias Achterberg. *Constraint Integer Programming.* PhD thesis, 2009.

[4] Tobias Achterberg, Timo Berthold, Thorsten Koch, and Kati Wolter. Constraint integer programming: A new approach to integrate cp and mip. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 5th International Conference, CPAIOR 2008 Paris, France, May 20-23, 2008 Proceedings 5*, pages 6–20. Springer, 2008.

[5] Shmuel Agmon. The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, 6:382–392, 1954.

[6] Miklos Ajtai. Parity and the pigeonhole principle. In *Feasible Mathematics: A Mathematical Sciences Institute Workshop, Ithaca, New York, June 1989*, pages 1–24. Springer, 1990.

[7] Michael Alekhnovich. Mutilated chessboard problem is exponentially hard for resolution. *Theoretical Computer Science*, 310(1-3):513–525, 2004.

[8] Ismail R Alkhouri, George K Atia, and Alvaro Velasquez. A differentiable approach to the maximum independent set problem using dataless neural networks. *Neural Networks*, 155:168–176, 2022.

[9] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The epfl combinational benchmark suite. In *IWLS*, 2015.

[10] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The epfl combinational benchmark suite. In *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, 2015.

[11] Amzon Web Services. Aws high performance computing: Virtually unlimited infrastructure and fast networking for scalable hpc. `https://aws.amazon.com/hpc/`. Accessed: 2021-07-10.

[12] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. SAT-based MaxSAT algorithms. *Artificial Intelligence*, 196:77–105, 2013.

[13] Carlos Ansótegui, Joel Gabas, Yuri Malitsky, and Meinolf Sellmann. MaxSAT by improved instance-specific algorithm configuration. *Artificial Intelligence*, 235:26–39, 2016.

[14] Gilles Audemard and Laurent Simon. On the glucose SAT solver. *International Journal on Artificial Intelligence Tools*, 27(01):1840001, 2018.

[15] Prasanna Balaprakash, Mauro Birattari, and Thomas Stützle. Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. In *Hybrid Metaheuristics: 4th International Workshop, HM 2007, Dortmund, Germany, October 8-9, 2007. Proceedings 4*, pages 108–122. Springer, 2007.

[16] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, et al. Openpiton: An open source manycore research framework. *ACM SIGPLAN Notices*, 51(4):217–232, 2016.

[17] Henrique Becker, Olinto Araujo, and Luciana S Buriol. Extending an integer formulation for the guillotine 2d bin packing problem. *Procedia Computer Science*, 195:499–507, 2021.

[18] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[19] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[20] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[21] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[22] Senne Berden, Mohit Kumar, Samuel Kolb, and Tias Guns. Learning MAX-SAT models from examples using genetic algorithms and knowledge compilation. In *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[23] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[24] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[25] Dimitri P Bertsekas. *Constrained optimization and Lagrange multiplier methods*. Academic press, 2014.

[26] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[27] Mauro Birattari. *Tuning Metaheuristics.* Studies in Computational Intelligence. Springer, Berlin, Heidelberg, 2009.

[28] Mauro Birattari, Thomas Stützle, Luis Paquete, Klaus Varrentrapp, et al. A racing algorithm for configuring metaheuristics. In *Gecco*, volume 2. Citeseer, 2002.

[29] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. F-race and iterated f-race: An overview. *Experimental methods for the analysis of optimization algorithms*, pages 311–336, 2010.

[30] Bob Bixby. The Gurobi Optimizer. *Transp. Re-search Part B*, 41(2):159–178, 2007.

[31] Pierre Bonami, Andrea Lodi, and Giulia Zarpellon. Learning a classification of mixed-integer quadratic programming problems. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 595–604. Springer, 2018.

[32] Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint arXiv:2102.09544*, 2021.

[33] Augustin Cauchy et al. Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538, 1847.

[34] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. Optimization of resource provisioning cost in cloud computing. *IEEE Transactions on Services Computing*, 5(2):164–177, 2012.

[35] Shaoming Chen, Samuel Irving, and Lu Peng. Operational cost optimization for cloud computing data centers using renewable energy. *IEEE Systems Journal*, 10(4):1447–1458, 2016.

[36] X. Chen, L. Wang, A. Y. Zomaya, L. Liu, and S. Hu. Cloud computing for vlsi floorplanning considering peak temperature reduction. *IEEE Transactions on Emerging Topics in Computing*, 3(4):534–543, 2015.

[37] Zong-Gan Chen, Ke-Jing Du, Zhi-Hui Zhan, and Jun Zhang. Deadline constrained cloud computing resources scheduling for cost optimization based on dynamic objective genetic algorithm. In *2015 IEEE Congress on Evolutionary Computation (CEC)*, pages 708–714, 2015.

[38] Yi-Ju Chiang, Yen-Chieh Ouyang, and Ching-Hsien Hsu. An efficient green control algorithm in cloud computing for cost optimization. *IEEE Transactions on Cloud Computing*, 3(2):145–155, 2015.

[39] Edwin KP Chong and Stanislaw H Zak. *An introduction to optimization.* John Wiley & Sons, 2004.

[40] Stephen A Cook and Robert A Reckhow. The relative efficiency of propositional proof systems. *The journal of symbolic logic*, 44(1):36–50, 1979.

[41] George B Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity analysis of production and allocation*, 13:339–347, 1951.

[42] George B Dantzig and John H Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[43] Jason V Davis and Inderjit S Dhillon. Structured metric learning for high dimensional problems. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 195–203, 2008.

[44] Roy De Maesschalck, Delphine Jouan-Rimbaud, and Désiré L Massart. The mahalanobis distance. *Chemometrics and intelligent laboratory systems*, 50(1):1–18, 2000.

[45] Jiankang Deng, Jia Guo, Niannan Xue, and Stefanos Zafeiriou. Arcface: Additive angular margin loss for deep face recognition. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4690–4699, 2019.

[46] Muhammet Deveci and Nihan Çetin Demirel. A survey of the literature on airline crew scheduling. *Engineering Applications of Artificial Intelligence*, 74:54–69, 2018.

[47] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

[48] Krzysztof Dudziński and Stanisław Walukiewicz. Exact methods for the knapsack problem and its generalizations. *European Journal of Operational Research*, 28(1):3–21, 1987.

[49] Omar El-Sewefy. Calibre in the cloud: Unlocking massive scaling and cost efficiencies, 2019.

[50] Yasemin Eryoldaş and Alptekin Durmuşoglu. A literature survey on offline automatic algorithm configuration. *Applied Sciences*, 12(13):6316, 2022.

[51] Stephen Fenstermaker, David George, Andrew B Kahng, Stefanus Mantik, and Bart Thielges. Metrics: a system architecture for design process optimization. In *Proceedings of the 37th Annual Design Automation Conference*, pages 705–710, 2000.

[52] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[53] ABKFM Fleury and Maximilian Heisinger. CaDiCaL, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. *SAT COMPETITION*, 2020:50, 2020.

[54] Christodoulos A Floudas and Xiaoxia Lin. Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Annals of Operations Research*, 139:131–162, 2005.

[55] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, March 2020.

[56] Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

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

[58] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[59] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

[60] Google Cloud. Gcp high performance computing. `https://cloud.google.com/solutions/hpc`. Accessed: 2021-07-10.

[61] Hadi Goudarzi, Mohammad Ghasemazar, and Massoud Pedram. Sla-based optimization of power and migration cost in cloud computing. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 172–179. Ieee, 2012.

[62] Evren Güney. An efficient linear programming based method for the influence maximization problem in social networks. *Information Sciences*, 503:589–605, 2019.

[63] Abhishek Gupta, Laxmikant V. Kale, Filippo Gioachin, Verdi March, Chun Hui Suen, Bu-Sung Lee, Paolo Faraboschi, Richard Kaufmann, and Dejan Milojicic. The who, what, why, and how of high performance computing in the cloud. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 1, pages 306–314, 2013.

[64] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[65] S Hashemi, CT Ho, AB Kahng, HY Liu, and S Reda. Metrics 2.0: A machine-learning based optimization system for ic design. In *Workshop on Open-Source EDA Technology*, page 21, 2018.

[66] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press, 1992.

[67] Holger H Hoos. Automated algorithm configuration and parameter tuning. *Autonomous search*, pages 37–71, 2012.

[68] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM (JACM)*, 21(2):277–292, 1974.

[69] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[70] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Kevin Murphy. Time-bounded sequential parameter optimization. In *Learning and Intelligent Optimization: 4th International Conference, LION 4, Venice, Italy, January 18-22, 2010. Selected Papers 4*, pages 281–298. Springer, 2010.

[71] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

[72] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. RC2: an efficient maxsat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, 2019.

[73] Michael D Intriligator. *Mathematical optimization and economic theory.* SIAM, 2002.

[74] Engin Ïpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. *SIGPLAN Not.*, 41(11):195–206, October 2006.

[75] Franjo Ivančić, Zijiang Yang, Malay K Ganai, Aarti Gupta, and Pranav Ashar. Efficient sat-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008.

[76] Max Jaderberg, Wojciech M Czarnecki, Dunning, et al. Human-level performance in first-person multiplayer games with population-based deep reinforcement learning. *arXiv preprint arXiv:1807.01281*, 2018.

[77] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455, 1998.

[78] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC–instance-specific algorithm configuration. In *ECAI 2010*, pages 751–756. IOS Press, Lisbon, Portugal, 2010.

[79] Andrew B Kahng, Jens Lienig, Igor L Markov, and Jin Hu. *VLSI physical design: from graph partitioning to timing closure.* Springer Science & Business Media, 2011.

[80] Andrew B Kahng and Stefanus Mantik. A system for automatic recording and prediction of design quality metrics. In *Proceedings of the IEEE 2001. 2nd International Symposium on Quality Electronic Design*, pages 81–86. IEEE, 2001.

[81] V. Kamath, R. Giri, and R. Muralidhar. Experiences with a private enterprise cloud: Providing fault tolerance and high availability for interactive eda applications. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 770–777, 2013.

[82] Richard M. Karp. Reducibility among Combinatorial Problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations*, pages 85–103. Springer US, Boston, MA, 1972.

[83] Richard M. Karp. Reducibility Among Combinatorial Problems. In *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*, pages 219–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[84] Mahmut Kaya and Hasan Şakir Bilge. Deep metric learning: A survey. *Symmetry*, 11(9):1066, 2019.

[85] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *The Multiple-Choice Knapsack Problem*, pages 317–347. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[86] Hans Kellerer, Ulrich Pferschy, David Pisinger, Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Multidimensional knapsack problems.* Springer, 2004.

[87] Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary computation*, 27(1):3–45, 2019.

[88] Elias B Khalil, Bistra Dilkina, George L Nemhauser, Shabbir Ahmed, and Yufen Shao. Learning to run heuristics in tree search. In *Ijcai*, pages 659–666, 2017.

[89] Elias B Khalil, Bistra Dilkina, George L Nemhauser, Shabbir Ahmed, and Yufen Shao. Learning to run heuristics in tree search. In *Ijcai*, pages 659–666, 2017.

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

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

[92] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[93] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[94] Gregory Koch, Richard Zemel, Ruslan Salakhutdinov, et al. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2. Lille, 2015.

[95] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[96] Vijay R Konda and John N Tsitsiklis. Onactor-critic algorithms. *SIAM journal on Control and Optimization*, 42(4):1143–1166, 2003.

[97] Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

[98] James Kotary, Ferdinando Fioretto, Pascal Van Hentenryck, and Bryan Wilder. End-to-end constrained optimization learning: A survey. *arXiv preprint arXiv:2103.16378*, 2021.

[99] Balakrishnan Krishnamurthy. Short proofs for tricky formulas. *Acta informatica*, 22:253–275, 1985.

[100] Markus Kruber, Marco E Lübbecke, and Axel Parmentier. Learning when to use a decomposition. In *International conference on AI and OR techniques in constraint programming for combinatorial optimization problems*, pages 202–210. Springer, 2017.

[101] Brian Kulis et al. Metric learning: A survey. *Foundations and Trends® in Machine Learning*, 5(4):287–364, 2013.

[102] Mohit Kumar, Samuel Kolb, Stefano Teso, and Luc De Raedt. Learning MAX-SAT from contextual examples for combinatorial optimisation. *Artificial Intelligence*, 314:103794, 2023.

[103] Joonseok Lee, Sami Abu-El-Haija, Balakrishnan Varadarajan, and Apostol Natsev. Collaborative deep metric learning for video understanding. In *Proceedings of the 24th ACM SIGKDD International conference on knowledge discovery & data mining*, pages 481–490, 2018.

[104] Sangho Lee, Jeongsub Choi, and Youngdoo Son. Efficient visibility algorithm for high-frequency time-series: application to fault diagnosis with graph convolutional network. *Annals of Operations Research*, pages 1–21, 2023.

[105] Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Boosting branch-and-bound MaxSAT solvers with clause learning. *AI Communications*, 35(2):131–151, 2022.

[106] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[107] Zhaoyu Li and Xujie Si. NSNet: A General Neural Probabilistic Framework for Satisfiability Problems. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors,

Advances in Neural Information Processing Systems, volume 35, pages 25573–25585. Curran Associates, Inc., 2022.

[108] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *Advances in neural information processing systems*, 31, 2018.

[109] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[110] X. Lin, Y. Li, H. Dai, and D. Guo. Architecture of web-eda system based on cloud computing and application for project management of ic design. In *2010 International Conference on Anti-Counterfeiting, Security and Identification*, pages 150–153, 2010.

[111] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022.

[112] Hung-Yi Liu and L. P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *Design Automation Conference*, pages 1–7, May 2013.

[113] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[114] Ali Louati, Rahma Lahyani, Abdulaziz Aldaej, Racem Mellouli, and Muneer Nusir. Mixed Integer Linear Programming Models to Solve a Real-Life Vehicle Routing Problem with Pickup and Delivery. *Applied Sciences*, 11(20):9551, 2021.

[115] Stephen Maher, Matthias Miltenberger, João Pedro Pedroso, Daniel Rehfeldt, Robert Schwarz, and Felipe Serrano. PySCIPOpt: Mathematical programming in python with the SCIP optimization suite. In *Mathematical Software – ICMS 2016*, pages 301–307. Springer International Publishing, Cham, 2016.

[116] The OpenCores maintainers. Opencores.

[117] Yuri Malitsky and Meinolf Sellmann. Instance-specific algorithm configuration as a method for non-model-based portfolio generation. In *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28–June1, 2012. Proceedings 9*, pages 244–259. Springer, 2012.

[118] C. Man, Z. Shi, Z. Xu, Y. Zong, K. Pang, and Y. Li. Cloud-eda:a paas platform architecture and application development for ic design test. In *Proceedings of 2014 International Conference on Cloud Computing and Internet of Things*, pages 1–4, 2014.

[119] Vasco Manquinho, Joao Marques-Silva, and Jordi Planes. Algorithms for weighted boolean optimization. In *Theory and Applications of Satisfiability Testing-SAT 2009: 12th International Conference, SAT 2009, Swansea, UK, June 30-July 3, 2009. Proceedings 12*, pages 495–508. Springer, 2009.

[120] CPLEX User's Manual. IBM ILOG CPLEX optimization studio. *Version*, 12:1987–2018, 2018.

[121] Aniruddha Marathe, Rachel Harris, David K Lowenthal, Bronis R De Supinski, Barry Rountree, Martin Schulz, and Xin Yuan. A comparative study of high-performance computing on the cloud. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 239–250, 2013.

[122] Raffaele Marino. Learning from survey propagation: a neural network for max-e-3-sat. *Machine Learning: Science and Technology*, 2(3):035032, 2021.

[123] Oden Maron and Andrew W Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11:193–225, 1997.

[124] Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for maxsat. In *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings 20*, pages 531–548. Springer, 2014.

[125] Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-wbo: A modular maxsat solver. In *Theory and Applications of Satisfiability Testing–SAT 2014: 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings 17*, pages 438–445. Springer, 2014.

[126] Microsoft Azure. Azure high performance computing. `https://azure.microsoft.com/en-us/solutions/high-performance-computing/`. Accessed: 2021-07-10.

[127] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

[128] Alan Mishchenko et al. Abc: A system for sequential synthesis and verification. *URL http://www. eecs. berkeley. edu/alanmi/abc*, pages 1–17, 2007.

[129] ML4CO. Machine learning for combinatorial optimization - neurips 2021 competition. ml4co competition. `https://www.ecole.ai/2021/ml4co-competition/`, 2021. Accessed: 2022-05-16.

[130] Volodymyr Mnih, Koray Kavukcuoglu, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[131] Inc MongoDB. Mongodb. *URL https://www. mongodb. com/. Cited on (2014)*, 9, 2014.

[132] António Morgado, Carmine Dodaro, and Joao Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings 20*, pages 564–573. Springer, 2014.

[133] António Morgado, Federico Heras, Mark Liffiton, Jordi Planes, and Joao Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18:478–534, 2013.

[134] Antonio Morgado, Alexey Ignatiev, and Joao Marques-Silva. MSCG: Robust core-guided MaxSAT solving. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):129–134, 2014.

[135] 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 AAAI conference on artificial intelligence*, volume 33, pages 4602–4609, 2019.

[136] Kevin Musgrave, Serge Belongie, and Ser-Nam Lim. Pytorch metric learning, 2020.

[137] Saurav Nanda, Ganapathy Parthasarathy, Parivesh Choudhary, and Arun Venkatachar. Resource aware scheduling for eda regression jobs. In *European Conference on Parallel Processing*, pages 639–651. Springer, 2019.

[138] Isaac Newton. *De analysi per aequationes numero terminorum infinitas.* 1711.

[139] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[140] Randal S Olson, Nathan Bartley, Ryan J Urbanowicz, and Jason H Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the genetic and evolutionary computation conference 2016*, pages 485–492, 2016.

[141] Patryk Osypanka and Piotr Nawrocki. Resource usage cost optimization in cloud computing using machine learning. *IEEE Transactions on Cloud Computing*, pages 1–1, 2020.

[142] B. Ozisikyilmaz, G. Memik, and A. Choudhary. Efficient system design space exploration using machine learning techniques. In *45th ACM/IEEE Design Automation Conference*, pages 966–969, June 2008.

[143] Vangelis Th Paschos. *Applications of combinatorial optimization*, volume 3. John Wiley & Sons, United Kingdom, 2014.

[144] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[145] Antoine Petitet. Hpl-a portable implementation of the high-performance linpack benchmark for distributed-memory computers. *http://www. netlib. org/benchmark/hpl/*, 2004.

[146] Sven Peyer, Dieter Rautenbach, and Jens Vygen. A generalization of dijkstra's shortest path algorithm with applications to vlsi routing. *Journal of Discrete Algorithms*, 7(4):377–390, 2009.

[147] Antoine Prouvost, Justin Dumouchelle, Lara Scavuzzo, Maxime Gasse, Didier Chételat, and Andrea Lodi. Ecole: A gym-like library for machine learning in combinatorial optimization solvers. In *Learning Meets Combinatorial Algorithms at NeurIPS*, 2020.

[148] Pongtorn Prukkantragorn and Kitt Tientanopajai. Price efficiency in high performance computing on amazon elastic compute cloud provider in compute optimize packages. In *2016 International Computer Science and Engineering Conference (ICSEC)*, pages 1–6, 2016.

[149] Haiyang Qian and Deep Medhi. Server operational cost optimization for cloud computing service providers over a time horizon. In *Hot-ICE*, 2011.

[150] Joseph Raphson. *Analysis aequationum universalis.* Typis TB prostant venales apud A. and I. Churchill, 1702.

[151] Ronald L Rardin and Ronald L Rardin. *Optimization in operations research*, volume 166. Prentice Hall Upper Saddle River, NJ, 1998.

[152] Maria Alejandra Rodriguez and Rajkumar Buyya. Deadline based resource provisioningand scheduling algorithm for scientific workflows on clouds. *IEEE transactions on cloud computing*, 2(2):222–235, 2014.

[153] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science*, 40(4):455–472, 2006.

[154] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[155] Martin JA Schuetz, J Kyle Brubaker, and Helmut G Katzgraber. Combinatorial optimization with physics-inspired graph neural networks. *Nature Machine Intelligence*, 4(4):367–377, 2022.

[156] Naresh Sehgal, John M Acken, and Sohum Sohoni. Is the eda industry ready for cloud computing? *IETE Technical Review*, 33(4):345–356, 2016.

[157] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *International Conference on Learning Representations*, 2019.

[158] Ahmad Shabani and Bijan Alizadeh. PMTP: A MAX-SAT-based approach to detect hardware trojan using propagation of maximum transition probability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(1):25–33, 2018.

[159] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[160] Q. Shen, C. Yu, J. Xiao, S. Tang, X. Meng, and J. Li. Dynamic scheduling of eda scientific workflows in hybrid computing environments. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 313–320, 2019.

[161] Xujie Si, Xin Zhang, Radu Grigore, and Mayur Naik. Maximum satisfiability in software analysis: Applications and techniques. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, pages 68–94. Springer, 2017.

[162] David Silver, Julian Schrittwieser, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[163] Jose Luis Lucas Simarro, Rafael Moreno-Vozmediano, Ruben S. Montero, and I. M. Llorente. Dynamic placement of virtual machines for cost optimization in multi-cloud environments. In *2011 International Conference on High Performance Computing Simulation*, pages 1–7, 2011.

[164] Aarti Singh, Dimple Juneja, and Manisha Malhotra. A novel agent based autonomous and service composition framework for cost optimization of resource provisioning in cloud computing. *Journal of King Saud University-Computer and Information Sciences*, 29(1):19–28, 2017.

[165] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[166] Jan A Snyman, Daniel N Wilke, et al. *Practical mathematical optimization.* Springer, 2005.

[167] Thamarai Selvi Somasundaram and Kannan Govindarajan. Cloudrb: A framework for scheduling and managing high-performance computing (hpc) applications in science cloud. *Future Generation Computer Systems*, 34:47–65, 2014.

[168] Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 53(2005):1–2, 2005.

[169] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[170] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[171] Robert Endre Tarjan and Anthony E Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

[172] Romeo Valentin, Claudio Ferrari, Jérémy Scheurer, Andisheh Amrollahi, Chris Wendler, and Max B. Paulus. Instance-wise algorithm configuration with graph neural networks, 2022.

[173] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of machine learning research*, 9(11), 2008.

[174] Vijay V Vazirani. *Approximation algorithms*, volume 1. Springer, 2001.

[175] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

[176] Dujuan Wang, Jiacheng Zhu, Yunqiang Yin, Joshua Ignatius, Xiaowen Wei, and Ajay Kumar. Dynamic travel time prediction with spatiotemporal features: using a gnn-based deep learning method. *Annals of Operations Research*, pages 1–21, 2023.

[177] Feng Wang and Huaping Liu. Understanding the behaviour of contrastive loss. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2495–2504, 2021.

[178] Hao Wang, Yitong Wang, Zheng Zhou, Xing Ji, Dihong Gong, Jingchao Zhou, Zhifeng Li, and Wei Liu. Cosface: Large margin cosine loss for deep face recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5265–5274, 2018.

[179] Runzhong Wang, Zhigang Hua, Gan Liu, Jiayi Zhang, Junchi Yan, Feng Qi, Shuang Yang, Jun Zhou, and Xiaokang Yang. A bi-level framework for learning to solve combinatorial optimization on graphs. *Advances in Neural Information Processing Systems*, 34:21453–21466, 2021.

[180] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[181] Sanford Weisberg. *Applied linear regression*, volume 528. John Wiley & Sons, 2005.

[182] Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. Probabilistic guarantees of execution duration for amazon spot instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2017.

[183] Weili Wu and Zhongnan Zhang. *Combinatorial Optimization and Applications: 14th International Conference, COCOA 2020, Dallas, TX, USA, December 11–13, 2020, Proceedings*, volume 12577. Springer Nature, 2020.

[184] Junyuan Xie, Ross Girshick, and Ali Farhadi. Unsupervised deep embedding for clustering analysis. In *International conference on machine learning*, pages 478–487. PMLR, 2016.

[185] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Hydra-mip: Automated algorithm configuration and selection for mixed integer programming. In *RCRA workshop on experimental evaluation of algorithms for solving problems with combinatorial explosion at the international joint conference on artificial intelligence (IJCAI)*, pages 16–30, 2011.

[186] Wenlong Yang, Lingli Wang, and Alan Mishchenko. Lazy man's logic synthesis. In *ICCAD*, pages 597–604. IEEE, 2012.

[187] X Yang. Introduction to mathematical optimization. *From linear programming to metaheuristics*, 2008.

[188] Cunxi Yu, Houping Xiao, and Giovanni De Micheli. Developing synthesis flows without human knowledge. In *Design Automation Conference*, DAC '18, pages 50:1–50:6. ACM, 2018.

[189] Gang Yu. *Industrial applications of combinatorial optimization*, volume 16. Springer Science & Business Media, 2013.

[190] Matthew M Ziegler, Hung-Yi Liu, and Luca P Carloni. Scalable auto-tuning of synthesis parameters for optimizing high-performance processors. In *ACM International Symposium on Low Power Electronics and Design*, pages 180–185, 2016.

[191] Matthew M Ziegler, Hung-Yi Liu, et al. A synthesis-parameter tuning system for autonomous design-space exploration. In *DATE*, pages 1148–1151, 2016.