# Irregular Deep Data Embedding and Learning

**LI, Wei**

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
July 2021

Abstract of thesis entitled:
Irregular Deep Data Embedding and Learning
Submitted by LI, Wei
for the degree of Master of Philosophy
at The Chinese University of Hong Kong in July 2021

Many research objects are organized under the non-Euclidean structure and often called irregular data. Irregular data, such as graphs and point clouds, often vary in terms of the size and scale, making some analytics methods infeasible due to its un-fixed scale and possible prohibitive cost for some huge cases. To overcome these issues, the study of data embedding method is essential for irregular data. Among all data embedding methods, deep learning is the most successful one.

However, it is hard to directly apply neural networks that perform well on the Euclidean domain to the non-Euclidean domain. Compared with regular data under an Euclidean or grid-like structure, irregular data loses some critical properties like shift invariance that are one of the key reasons for the success of deep learning. The thesis will discuss the applications of deep learning models for two representative irregular data representations, graph and point cloud.

For the graph embedding, we study its applications in multiple patterning lithography decomposition (MPLD) problem and the graph coloring problem. MPLD has been widely investigated, but so far there is no decomposer that dominates others in terms of both the optimality and the efficiency. This observation motivates us exploring how to adaptively select the most suitable MPLD strategy for a given layout graph, which is non-trivial and still an open problem. We propose a layout decomposition framework based on graph neural networks (GNNs) to obtain the graph embeddings of the layout. The graph embeddings are used for graph library construction, decomposer selection and graph matching. Besides the applications in the industrial workflow, we study the power of GNNs for a pure graph coloring problem from three perspectives. First, we extend the theoretical analysis of GNNs from tasks under homophily to heterophily, and prove that previous definitions on the power of GNNs cannot generalize to a task under heterophily including the coloring problem. Furthermore, we show that any AC-GNN is a local coloring method, and any local coloring method is non-optimal by exploring the limits of local methods over sparse random graphs, thereby demonstrating the non-optimality of AC-GNNs due to its local property. Moreover, we discuss the color equivariance for the coloring problem. Following the discussions above, we develop a global GNN-based approach by un-supervised learning, which proves to enhance the discrimination power and

retain the color equivariance.

For the point cloud embedding, we focus on its application on the routing tree construction problem. In the routing tree construction, both wirelength (WL) and pathlength (PL) are of importance. Among all methods, PD-II and SALT are the two most prominent ones. However, neither PD-II nor SALT always dominates the other one in terms of both WL and PL for all nets. In addition, estimating the best parameters for both algorithms is still an open problem. We model the pins of a net as point cloud and formalize a set of special properties of such point cloud. Considering these properties, we propose a novel deep neural net architecture, TreeNet, to obtain the embedding of the point cloud. Based on the obtained cloud embedding, an adaptive workflow is designed for the routing tree construction. In the workflow, the cloud embedding is used to select the algorithm and predict the balance parameter.

# 摘要

许多研究对象都是在非欧几里得结构下组织的，因此也被称为不规则数据。不规则数据，如图和点云，往往在大小和规模上存在差异。这种差异导致数据大小的不固定，甚至会存在超大数据。这些潜在的问题使得一些分析方法不可行，为了克服这些问题，对于不规则数据来说，数据嵌入方法的研究变得至关重要。在所有的数据嵌入方法中，深度学习是最成功的方法。

然而，我们很难将在欧氏领域表现良好的神经网络直接应用到非欧氏领域中。与欧几里得或网格状结构下的规则数据相比，不规则数据失去了一些关键的特性，如移位不变性，而这些特性是深度学习发挥作用的关键之一。本论文主要将研究深度学习在两种代表性的不规则数据表征，图和点云，中的应用。

对于图嵌入，我们分别探讨其在多重图案光刻分解及图着色问题中的应用。多重图案光刻分解问题已经被广泛研究，然而至今也没有一个分解器可以在性能和效率上都优于其他的分解器。因此，给定一个布局图，我们试图自适应地选择最合适的分解器。我们提出了一个基于图神经网络的布局分解框架。首先我们用图神经网络获得布局的图嵌入，图嵌入接着被用于图库构建、分解器选择以及图匹配。除了在工业流程中的应用，我们从三个角度研究了 GNNs 在纯图着色问题上的能力。首先，我们将图神经网络的理论分析从同质性下的任务扩展到异质性下的任务，并证明了之前关于图神经网络能力的定义不能推广到异质性下的任务。此外，我们通过探索稀疏随机图上的局部方法的极限，证明任何基于消息传输的图神经网络都是一种局部着色方法，而任何局部着色方法都是非最优的，从而证明基于消息传输的图神经网络在图着色问题上是非最优的。最后，我们讨论了着色问题的颜色等值性。根据上述探究，我们通过无监督学习开发了一种全局的图神经网络，该方法被证明可以提高着色能力并保留颜色的等值性。

对于点云嵌入，我们研究其在布线树问题上的应用。在布线树的构建中，浅度（路径长度）和轻度（线路长度）都很重要。同时，对于大多数构建布线树的算法，都会存在一个参数来平衡这两个指标。在所有方法中，PD-II 和 SALT 是最突出的两种。然而，无论是 PD-II 还是 SALT 都不能完全打败另一个算法。我们将网的针脚建模为点云，并证明了这种点云存在一系列特殊属性。利用这些属性，我们提出了新的深度神经网络架构以获得点云的嵌入。基于获得的点云嵌入，我们设计了一个自适应的工作流程来构建布线树。在这个流程种，点云嵌入被用于选择算法以及预测平衡参数。

# Acknowledgement

First and foremost, I would like to express my greatest thanks to my advisor Prof. Bei Yu for his continuous support, for countless inspiring discussions ranging from mathematical proofs to the design of the whole framework, for giving me the freedom to do the research I am interested in, and for always believing my talents in research even when I was skeptical about my ability.

Many thanks also go to Prof. Michael R. Lyu, Prof. Yuqun Zhang, Prof. Lingming Zhang, Prof. Yibo Lin, and Prof. David Z. Pan (in chronological order) who led me to the path of research and provided me with unselfish support and guidance during my undergraduate and MPhil career.

Being part of the CUDA group was an unforgettable and truly enriching experience. I would like to thank Dr. Yuzhe Ma, Dr. Haoyu Yang, Hao Geng, Tinghuan Chen, Ran Chen, Lu Zhang, Qi Sun, Wanli Chen, Zhuolun (Leon) He, Peiyu Liao, Wenqian Zhao, Yang Bai, Binwu Zhu, Chen Bai, Siting Liu, Ziyang Yu, Guojin Chen, and Xufeng Yao for their inspiring group presentations and wonderful brainstorms.

My sincere thanks also go to all mates in the lab 913: Dr. Jie Geng, Dr. Jordan Chak-Wa Pui, Dr. Haocheng Li, Jingsong Chen, Bentian Jiang (aka Bro Mega), Jinwei Liu, Xiaopeng Zhang, Dan Zheng, Fangzhou Wang (aka involution king), Lixin Liu, Xinshi Zang, and Shiju Lin. I could not have hoped for a more welcoming and happy environment for my MPhil study.

Last but not least, I would like to thank my parents, Yuebing Li and Shengmei Wei: Thank you for your endless love, unconditional support for my decisions, and forgiving my absence in the past few years. You are always my motivation to be a great man.

# Contents

**Abstract**

**Acknowledgement**

**1 Introduction** **1**
- 1.1 Irregular Data and Its Embedding . . . . . 1
- 1.2 Deep Learning . . . . . 2
- 1.3 Thesis Overview . . . . . 3

**2 Literature Review** **5**
- 2.1 Graph Neural Networks . . . . . 5
  - 2.1.1 Analysis on the power of GNNs. . . . . . 5
  - 2.1.2 GNNs for NP problem. . . . . . 6
  - 2.1.3 GNNs for tasks under heterophily. . . . . . 7
- 2.2 Neural Networks for Point Clouds . . . . . 7
  - 2.2.1 Multi-view based Methods. . . . . . 8
  - 2.2.2 Volumetric-based Methods. . . . . . 8
  - 2.2.3 Point-based Methods. . . . . . 9
- 2.3 Graph Coloring Problems . . . . . 10
  - 2.3.1 Non-learning-based methods. . . . . . 11
  - 2.3.2 Learning-based methods. . . . . . 11

**3 Multiple Patterning Lithography** **13**
- 3.1 OpenMPL: An Open Source Layout Decomposer . . . . . 13
  - 3.1.1 Preliminary . . . . . 14
  - 3.1.2 The OpenMPL Framework . . . . . 16
  - 3.1.3 Stitch Redundancy Removal . . . . . 19
  - 3.1.4 Optimized ILP-based algorithm . . . . . 23
  - 3.1.5 Flexible Exact cover-based algorithm . . . . . 26
  - 3.1.6 Experimental Results . . . . . 33
- 3.2 Adaptive Layout Decomposition with Graph Embedding Neural Networks 41
  - 3.2.1 Preliminaries . . . . . 44
  - 3.2.2 Layout Decomposition Algorithms . . . . . 46

3.2.3 Adaptive Decomposition Framework . . . 51
3.2.4 Experimental Results . . . 57

**4 Graph Coloring** **64**
4.1 Preliminaries . . . 66
4.2 Powerful GNNs for Graph Coloring . . . 67
4.2.1 Discrimination power under heterophily . . . 68
4.2.2 Locality . . . 69
4.3 Color Equivariance . . . 71
4.4 Our Method . . . 72
4.5 Experiments . . . 74
4.5.1 Experimental setup . . . 74
4.5.2 Comparison with other AC-GNN variations . . . 75
4.5.3 Comparison with other graph coloring methods . . . 75

**5 Routing Tree Construction** **77**
5.1 Introduction . . . 77
5.2 Preliminaries . . . 79
5.2.1 Routing Tree . . . 79
5.2.2 Point Cloud . . . 80
5.2.3 Problem Formulation . . . 80
5.3 The Proposed Approaches . . . 80
5.3.1 Property Analysis . . . 80
5.3.2 Cloud embedding by TreeNet . . . 83
5.3.3 Routing Tree Construction based on Point Cloud Embedding . . . 84
5.4 Experimental Results . . . 86
5.4.1 Comparison with other DL models . . . 87
5.4.2 Comparison with routing tree constructors . . . 88

**6 Appendix** **93**
.1 Graph Terminology . . . 93
.2 Proofs in Chapter 4 . . . 93
.2.1 Proof of Property 1 . . . 93
.2.2 Proof of Property 2 . . . 94
.2.3 Proof of Corollary 1 . . . 94
.2.4 Proof of Corollary 2 . . . 96
.2.5 Proof of Property 3 . . . 96
.2.6 Proof of Theorem 2 . . . 97
.3 Preproces & Postprocess in Chapter 4 . . . 98
.4 Global method . . . 100
.4.1 Distance encoding [63]. . . . 100
.4.2 Readout function [11]. . . . 101
.4.3 Identity-aware Graph Neural Networks [125]. . . . 102

.4.4 Randomness. . . . . 104
.5 Supplementary Experiments in Chapter 4 . . . . . 104
.5.1 Experiment & model settings. . . . . 104
.5.2 Comparison with other methods. . . . . 106
.5.3 Ablation study . . . . . 106
.5.4 Other discussion. . . . . 108

**Bibliography** **111**

# List of Figures

1.1 Examples of graph embeddings 1.1(a) and point cloud embeddings 1.1(b). 2

2.1 Three major kinds of deep learning based methods for point cloud. . . . . 9

3.1 An example of TPLD with stitches. . . . . . . . . . . . . . . . . . . . . . 15
3.2 The workflow of OpenMPL. . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.3 Projection results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.4 An TPLD example of stitch redundancy removal (SRR) . . . . . . . . . . 21
3.5 An example of the non-optimality of the original ILP-based algorithm. . . 25
3.6 Double patterning instance. . . . . . . . . . . . . . . . . . . . . . . . . 28
3.7 Double patterning instance with stitches. . . . . . . . . . . . . . . . . . 29
3.8 Original EC-based algorithm vs. our flexible EC-based algorithm. . . . . 30
3.9 The non-optimal case of the original EC-based algorithm . . . . . . . . . 32
3.10 An example of the decomposition results . . . . . . . . . . . . . . . . . 36
3.11 An example of graph embeddings of layout graphs . . . . . . . . . . . . . 42
3.12 An example of the routed layout and its graph representations. . . . . . 42
3.13 The histogram of the number of layout graphs and graphs that need not stitches. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
3.14 The example of redundant stitch candidates. . . . . . . . . . . . . . . . 49
3.15 The example of alterative solutions for stitches. . . . . . . . . . . . . . 49
3.16 An example on how AC-GNN gives the coloring results directly. . . . . . 51
3.17 The workflow of adaptive layout decomposition framework. . . . . . . . . 52
3.18 Overview of the process for graph embedding . . . . . . . . . . . . . . . 54
3.19 Runtime breakdown of adaptive layout decomposition framework. . . . . 61

4.1 Examples of graph structures in which some AC-GNNs fail to discriminate the equivalent node pair. . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.2 Comparison among AC-GNN variations. . . . . . . . . . . . . . . . . . . 74

5.1 Cloud embeddings for tree construction. . . . . . . . . . . . . . . . . . . 78
5.2 Examples of the down-sampling. . . . . . . . . . . . . . . . . . . . . . . 79
5.3 Examples of the routing trees. . . . . . . . . . . . . . . . . . . . . . . . 81
5.4 Examples of the node with the same coordinates. . . . . . . . . . . . . . 82
5.5 Comparison among different grouping methods . . . . . . . . . . . . . . 82

5.6 Illustration of TreeConv. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
5.7 Illustration of TreeNet. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
5.8 The workflow for the routing tree construction. . . . . . . . . . . . . . . . . 85
5.9 Runtime comparison with SALT and SALT$^*$. . . . . . . . . . . . . . . . . . . 89
5.10 Runtime breakdown of the routing tree construction framework. . . . . . . 92

1 A contradiction example to prove that the distance encoding makes an AC-GNN global. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
2 Comparison with other heuristic methods. . . . . . . . . . . . . . . . . . . . 107
3 Comparison with different model depth and mean aggregation. . . . . . . 108

# List of Tables

2.1 Summary of existing point-based methods which follow a sampling-grouping-encoding scheme . . . 8

3.1 Effective of stitch redundancy removal (SRR) . . . 34
3.2 Our ILP vs. Original ILP [131] on ISCAS benchmarks . . . 34
3.3 Our ILP vs. Original ILP [131] on ISPD benchmarks . . . 35
3.4 Our EC vs. Original EC [17] on ISCAS benchmarks . . . 37
3.5 Our EC vs. Original EC [17] on ISPD benchmarks . . . 37
3.6 Non-stitch Decomposition Cost Comparison on ISCAS benchmarks . . . 39
3.7 Decomposition Cost Comparison on ISCAS benchmarks . . . 39
3.8 Decomposition comparison on ISPD benchmarks . . . 40
3.9 Decomposition Runtime (s) Comparison on ISCAS benchmarks . . . 40
3.10 Comparsion among different decomposers . . . 51
3.11 F1 score comparison of (a) proposed RGCN and (b) conventional GCN. . . 59
3.12 Decomposition Cost Comparison . . . 59
3.13 Decomposition Runtime Comparison . . . 60
3.14 Results of stitch redundancy prediction. . . . 60
3.15 Layout statistics and results by GNN decomposer . . . 62

4.1 Solved ratio of existing GNNs and the simplest greedy algorithm on layout dataset. . . . 68
4.2 Graph datasets information and results by different coloring methods. . . . 76

5.1 Comparison to existing methods. . . . 85
5.2 ICCAD 2015 Benchmark Label Statistics (part) . . . 87
5.3 Algorithm selection results . . . 88
5.4 Results on shallowness . . . 90
5.5 Results on normalized PL . . . 91

1 The results of our method without postprocess on the normal dataset. . . 109
2 The results of our methods without postprocess and with concatenation on the layout dataset. . . . 109

# Chapter 1

# Introduction

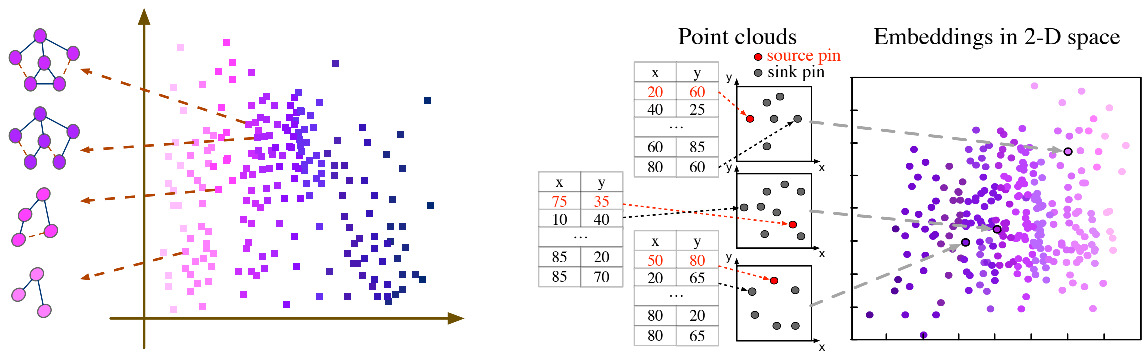
## 1.1 Irregular Data and Its Embedding

Although data under a regular Euclidean or grid-like structure seems to be easier and more straightforward to use for research purpose, we live in a world that is not perfectly regular most of the time: from natural compositions and interactions to humankind activities and products such as molecules, proteins, living organisms, integrated circuit (IC), social networks, planetary systems, and many more examples across a wide range of scales in the universe [54].

As a result, many scientific fields study data with an underlying structure that is non-Euclidean [16], where two representative ones are graph and point cloud. A graph is a classic data structure defined by a collection of discrete objects (nodes), and their pairwise relationships (edges). Different with the graph, a point cloud only represents a set of data points in the Euclidean space such as our 3D real world, without containing any connection information.

The study and analysis of the irregular data becomes increasingly crucial because of not only its ubiquitousness but also its importance in various research tasks, such as node classification and graph matching. For example, by analyzing the hidden information of previous preferences of users in the website, we can provide a relatively accurate recommendation based on the preference graph, an graph abstraction of the preference information.

However, the analytics of irregular data is inhibited by two factors. First, most traditional analytics methods for irregular data suffer from the high computation and space cost, especially for large data. Second, irregular data instances usually vary in terms of size and scale, making them hard to digest for those methods that require fixed-size input such as Multilayer Perceptron (MLP). To overcome these issues, the data embedding method is discussed and studied widely. Basically, a data embedding method is to generate an embedding of a data instance such that the original instance with various size is transformed into a vector space in a lower but unified dimension with maximal representation capability, where the obtained embedding can be further used in downstream



(a) (b)

Figure 1.1: Examples of graph embeddings 1.1(a) and point cloud embeddings 1.1(b).

tasks. Figure 1.1 shows toy examples of graph embeddings and point cloud embeddings, where the graphs and point clouds are converted into a vector in the 2-D space, no matter how large or different the graph/point cloud sizes are.

## 1.2 Deep Learning

*Deep learning* refers to learning complicated concepts by building them from simpler ones in a hierarchical or multi-layer manner [16]. A *deep neural network* is a widely-used realization of the abstract multi-layer hierarchies. For a long time, training a deep learning model is almost impossible due to the lack of training data and the prohibitive computational complexity caused by deep multilayer hierarchies and the explosion of data dimensions. However, over the past few years, owing to the advance of big data technology and the development of Graphic Processing Units (GPUs) and AI-chips which are highly optimized for parallel computing and neural network computing, a deep neural network becomes feasible and gains overwhelming success as an effective yet efficient data embedding method, leading to considerable breakthroughs across many different fields such as computer vision, natural language processing, medicine, and VLSI design.

One of the crucial reasons for the success of deep learning is their ability to leverage statistical properties of the data [16], especially for regular data such as images, videos, and audios. For example, an image can be also regarded as a series of data points sampled on a grid in the Euclidean space. Following this formalization, some meaningful statistical properties like stationarity, locality, and compositionality correspond to the shift invariance, local connectivity, and the multi-resolution structure of grid respectively [1]. All of these properties are well-leveraged by convolutional neural networks (CNNs), consisting of the convolution and down-sampling operations. The convolution operation only

[1]This statement does not indicate that any data under a grid structure has these properties. For example, stationarity can be found in the natural images but does not exist in a randomly generated pixel sequences.

contains filters which are usually small spatially. At each step, the convolution computation is conducted by the dot-product between the filter entries and the input entries in the corresponding small receptive field, which utilizes the local connectivity. A complete computation in the convolutional layer is composed of repeated steps above, that is, the filters will be slidden across the whole image while the filters keep unchanged. Through this way, the convolutional layer is naturally shift equivariant. Between successive convolutional layers, pooling Layer is sometimes inserted to reduce the size of the feature map, which not only keeps the shift equivariance but also makes the model sensitive to the multi-resolution structure.

Many neural network e.g., CNNs for images and videos, Recurrent Neural Networks (RNNs) for sequences show their considerable power by considering some priors about these data. Nevertheless, most previous models are particularly designed for regular data, which are organized under the Euclidean structure. When directly transferring these models for irregular data, most of them will lose the effectiveness since those priors for regular data do not hold anymore. For example, on non–Euclidean domain, we even cannot find a suitable definition of the basic convolution operation even though it is very common on the Euclidean domain. There are some attempts to apply deep learning in the irregular data, such as Graph Convolutional Network (GCN) [55] for graphs and PointNet for point clouds [88]. However, unlike regular images which are mostly from nature and own consistent properties, irregular data usually varies largely based on the subject. For example, in the recommendation system, the target graph is bipartite, i.e., whose nodes can be divided into user node set and product node set and usually contains billion of nodes, moreover, two close user nodes (which may represent a relationship like friends in the real-world) tend to have a more similar preferences, called homophily. On the contrary, in the graph coloring problem, the target graph is relatively small in terms of the size, requiring connected nodes being assigned different colors, known as heterophily. These tremendous differences among various tasks make it hard to find a universal deep learning model which performs well for all kinds of irregular data representations. One of the main targets of this thesis is to provide a general idea on what a good deep learning model is and how to design such a model for a specific task under the non-Euclidean sturcture.

## 1.3 Thesis Overview

Rather than designing models by empirical intuition and experimental trials, we try to propose solution from a theoretical perspective, e.g., analyzing the structural properties and proving its limitation bounds. In the thesis, we focus on the key question: *What is a good irregular data embedding for a specific task, and how to obtain it by deep learning?* We try to answer the question by learning two representative irregular data representations, graph and point cloud. Specifically, for the graph embedding, we study its applications in multiple patterning lithography decomposition (MPLD) problem and the graph coloring problem. For the point cloud embedding, we focus on its application on the routing tree construction problem.

In Chapter 2, we will review the literature on deep learning methods for graphs and point clouds. Moreover, we review previous methods for the graph coloring methods.

In Chapter 3, we discuss the applications of graph embeddings in the MPLD problem. Before stepping into the discussion, we first introduce our proposed open-source layout decomposition framework: OpenMPL. Then, we introduce how layout graph embedding is obtained and integrated into our adaptive layout decomposition framework.

In Chapter 4, we analyze the power of GNNs for a pure graph coloring problem from three perspectives: heterophily, locality, and color equivariance. Following the analysis, we develop a global GNN-based approach by un-supervised learning, which proves to enhance the coloring ability and retain the color equivariance.

In Chapter 5, we study point cloud embedding method and its application in the routing tree construction problem. We model the pins of a net as point cloud and formalize a set of special properties of such point cloud. Considering these properties, we propose a novel deep neural net architecture, TreeNet, to obtain the embedding of the point cloud.

In the appendix shown in Chapter 6, we list basic graph terminologies, proofs, more detailed discussions and analysis, and incremental experimental results of Chapter 4.

---

□ **End of chapter.**

# Chapter 2

# Literature Review

This chapter reviews the literature about deep learning methods for irregular data embeddings and previous methods for the graph coloring problem.

## 2.1 Graph Neural Networks

With the development and further study of Neural Network, Graph Neural Networks, as a branch of Deep Neural Networks, has shown promising results in many domains such as the graph embedding. Generally speaking, GNN takes the graph as input and returns the node embeddings or graph embeddings. Usually, GNN is composed of two modules, aggregator and encoder, which exploit the neighborhood information and node attributes respectively. Specifically, for each node $u$ in graph $G$, the aggregator is to aggregate neighbor $v$'s representations $\boldsymbol{h}_v$ and obtain an intermediate representation $\hat{\boldsymbol{h}}_u$ such that the final graph embedding is able to contain graph structure information. Especially, one virtual additional edge is added to each node, i.e. a single self-connection whose weight is defined as 1 to guarantee that the latter layer's node representation can also be informed by the corresponding representation at the previous layer besides neighbors. Encoder is to multiply the aggregated representation $\hat{\boldsymbol{h}}_u$ with a learnable matrix followed with a non-linear activation function. GNN can be also explained in a message-passing way where the intermediate representations can be viewed as messages. The aggregation is the actual message-passing phase and each node passes its message to its neighbors along the edge. The encoder is served as the integration phase, in which each node integrates received the message and reduces it into its new message. Each message-pass and integration phase formulate one GNN layer. The representation after the final layer is called the node embedding of each node and the graph embedding by GNN is usually obtained by a summation or mean operation using node embeddings.

### 2.1.1 Analysis on the power of GNNs.

With the overwhelming success of GNNs in various fields ranging from recommendation system and VLSI design, recently, the study on the power of GNNs becomes more and more important and necessary, and has attracted extensive interest. The two recent papers [82, 117] formalize the power as the capability to map two equivalent nodes to the same node embedding. They explore the power by establishing a close connection between GNNs and 1-Weisfeiler-Lehman (WL) test, a classical algorithm for the graph isomorphism test. More specifically, they independently showed that every time when two nodes are assigned the same embedding by any GNN, the two nodes will always be labeled the same by the 1-WL test, which means that GNNs are upper-bounded by 1-WL test in terms of the representation power. To develop a more powerful GNN that breaks through the limit by 1-WL test, many attempts are made from different perspectives. Some GNNs [22, 77, 78, 82] are proposed by mimicking a higher-order-WL test based on higher-order tensors. Another direction is to introduce more informative features/operations to make the model sensitive to the substructure [63] or global structure [11, 126, 125]. We leave the detailed discussion of such a non-local scheme in Appendix .4. Besides the study on the comparison with WL-test, many other works investigate the power of GNNs from different angles and a lot of interesting conclusions are obtained. Xu et al. [119] shows that GNNs align with DP and thus are expected to solve tasks that are solvable by DP. This interesting conclusion leaves us a future work to study GNN in the coloring problem by learning previous DP-based coloring algorithms. Loukas et al. [73] concludes that the product of the GNN’ s depth and width must exceed a polynomial of the graph size to obtain the optimal solution of some problems, e.g., Maximum Independent Set (MIS) problem, and coloring problem. This conclusion motivates our experiments on the model depth, which is covered in Appendix .5.3. Another work [127] explores the design space for GNNs and gives some best parameters in various design dimensions, where best means the selected parameters make the corresponding GNNs more effective than others. We follow the guidance of this work to select most hyper-parameters and model architectures, as shown in Appendix .5.1. GeomGCN [87] points the limits of AC-GNNs from the perspective of network geometry, the node can only exchange information with its neighbors, while the long-range dependencies are missed and similar nodes (may be very distant) are more likely to be proximal. To overcome the issues, a novel geometric aggregation scheme was proposed. Generally, instead of aggregating information from graph neighborhoods directly, the original graph is mapped to a latent continuous space according to pre-calculated node embedding. Then, a structural neighbor relation is constructed based on the distance and relative direction in the latent space. However, their motivation is not applicable for the coloring problem: the coloring results are totally not relevant with the similarity of nodes.

### 2.1.2 GNNs for NP problem.

Recently, the applications of GNNs on NP problems received great attention. Some works integrate GNNs to a sophisticated heuristic algorithm designed for a specific NP problem. Li et al. [69] proposes a GNN-based framework to solve the MIS problem, where the adopted GNN generates multiple probability maps to represent the likelihood of each vertex being in the optimal solution. However, the following heuristic algorithm to handle the *multiple* probability maps is time-consuming. In their experiments, a graph with 1,000 vertices will yield up to 100K diverse solutions and the heuristic algorithm is processed up to 10 minutes. Not saying that the runtime may explode when applied in the $k$-coloring problem. Another work [72] uses GNNs to solve the subgraph matching problem, a problem of determining whether a given query graph is a subgraph of a large target graph. They designed a particular loss function, to ensure that the subgraph relations are preserved in the embedding space. Besides these direct applications, some theoretical works discussed the power of GNNs to solve the NP problem. If P $\neq$ NP, GNNs cannot exactly solve these problems. Under this assumption, Sato [93] demonstrates the approximation ratios of GNNs for some combinatorial problems such as the minimum vertex cover problem. They study the ratio by building the connection between GNNs and distributed local algorithms. Specifically, they show that the set of graph problems that GNN classes can solve is the same as the one that distributed local algorithm classes can solve. Besides a pure GNN, Dai et al. [26] develops a framework that combines reinforcement learning and graph embedding to address some NP problems. Simply speaking, the reinforcement learning model uses the graph embedding obtained by Structure2Vec [25].

### 2.1.3 GNNs for tasks under heterophily.

To the best of our knowledge, Zhu et al. [139] is the first and only work that formally addressed the drawbacks of previous GNNs on tasks under heterophily. Beyond homophily, they proposed three designs that can be beneficial for the learning under heterophily: 1) The node embedding and aggregated embeddings should be separated. This statement aligns with our Proposition 2, addressing the limitation of an integrated AC-GNN in the coloring problem. Although most previous works focus on the homophily scenario, some of them [127, 41] also pay attention to the separation of neighbor embeddings and ego-embedding (i.e., a node' s embedding). 2) The aggregation function should involve higher-order neighborhoods. The intuition is that higher-order neighborhoods may be homophily-dominant. Take the coloring problem as an example, if two nodes, say $u, v$, are connected with another same node $t$, then $u, v$ are more likely to be assigned the same color. This design is also employed in previous works [7, 28] for homophily, considering that a higher order polynomials of the normalized adjacency matrix indicates a low-pass filter. 3) The final results should combine intermediate representations from all layers. The design is originally introduced in jumping knowledge networks [118] and motivated by the fact that each layers contains information from neighborhoods of different depth.

## 2.2 Neural Networks for Point Clouds

Point cloud, as its name suggests, is the set of data points in space. Each point in a point cloud contains its location information, for example, a set of $x, y$ and $z$ coordinates if it is in a 3D space. The study of point clouds becomes increasingly important mainly because of the rapid development of 3D acquisition technologies. The emergence of affordable

Table 2.1: Summary of existing point-based methods which follow a sampling-grouping-encoding scheme

| Methods | | *Sampling* | *Grouping* | *Encoding* |
|---|---|---|---|---|
| MLP-based methods | PointNet [88] | - | - | $v'_{ic} = \sigma(\boldsymbol{\theta}_c \boldsymbol{v}_i)$ |
| | PointNet++ [89] | FPS | Ball query | $v'_{ic} = \max_{j \in E_i} \sigma(\boldsymbol{\theta}_c \boldsymbol{v}_j)$ |
| | Yang *et al.* [122] | FPS/GSS | - | $v'_{ic} = \sigma(\boldsymbol{\theta}_c \boldsymbol{v}_i)$ |
| Conv-based methods | PointCNN [68] | Random/FPS | KNN | $\boldsymbol{v}'_i = Conv(X \times \boldsymbol{\theta}(\boldsymbol{v}_i - \boldsymbol{v}_j))$ |
| | RS-Conv [71] | FPS | Ball query | $v'_{ic} = \sigma(\frac{1}{\lvert E_i \rvert} \sum_{j \in E_i} \boldsymbol{v}_i \times MLP(CONCAT(\boldsymbol{v}_i - \boldsymbol{v}_j, \boldsymbol{v}_i, \boldsymbol{v}_j)))$, |
| Graph-based methods | ECC [100] | VoxelGrid | KNN | $v'_{ic} = \frac{1}{\lvert E_i \rvert} \sum_{j \in E_i} F(\boldsymbol{v}_j)$, |
| | FoldingNet [123] | Random | KNN | $v'_{ic} = \boldsymbol{\theta}_c \cdot \max_{j \in E_i} \sigma(\boldsymbol{v}_j)$, |
| | KCNet [99] | Poisson disk | KNN | $v'_{ic} = \max_{j \in E_i}(\boldsymbol{\theta}_c \cdot \boldsymbol{v}_j)$, |
| | DGCNN [111] | - | KNN | $v'_{ic} = \max_{j \in E_i} \sigma(\boldsymbol{\theta}_c \cdot CONCAT(\boldsymbol{v}_i - \boldsymbol{v}_j, \boldsymbol{v}_i))$, |

and available sensors brings about various types of data, in which point cloud is one of the most essential representations since it directly preserves the original geometric information without any discretization. Due to the increasing importance of point clouds and the presence of massive data, a great number of works [88, 89, 111, 68, 102, 113, 124, 34, 80, 91, 110] are developed to explore the possibility of deep learning on point clouds.

Guo *et al.* [39] categorize deep learning based methods for point clouds into three major types: multi-view based methods, volumetric-based methods, and point-based methods. We give a simple illustration of these three methods in Figure 2.1, where Figure 2.1(a) is the original point cloud, and Figures 2.1(b), 2.1(c), 2.1(d) correspond to View-based, Volumetric-based, and Point-based methods respectively.

### 2.2.1 Multi-view based Methods.

Multi-view based methods [102, 113, 124, 34] are designed for 3D point cloud. These methods first transform a 3D point cloud into multiple views through projection and extract view-wise features. Finally, extracted features are fused together to generate a cloud embedding. Among these works, the major discrepancy locates in the fusion of multiple view-wise features, which is also the key challenge. For example, MVCNN [102] uses a straightforward max-pooling operator to aggregate features; Yang *et al.* [124] fuse these features based on a relation network, which includes inter-relationships among regions and views. GVCNN [34] proposes a hierarchical view-group-shape architecture to obtain the cloud embedding from the view level, the group level, and the shape level.

### 2.2.2 Volumetric-based Methods.

Volumetric-based methods also use the idea of transformation to solve the irregularity of regular data. Instead of views, these methods voxelize a point cloud into regular grids, and then a conventional CNN is compatible with the volumetric data for feature extraction. VoxNet [80], one of the pioneer works using volumetric data, uses a volumetric occupancy grid representation and fed it into a 3D Convolutional Neural Network (3D CNN).

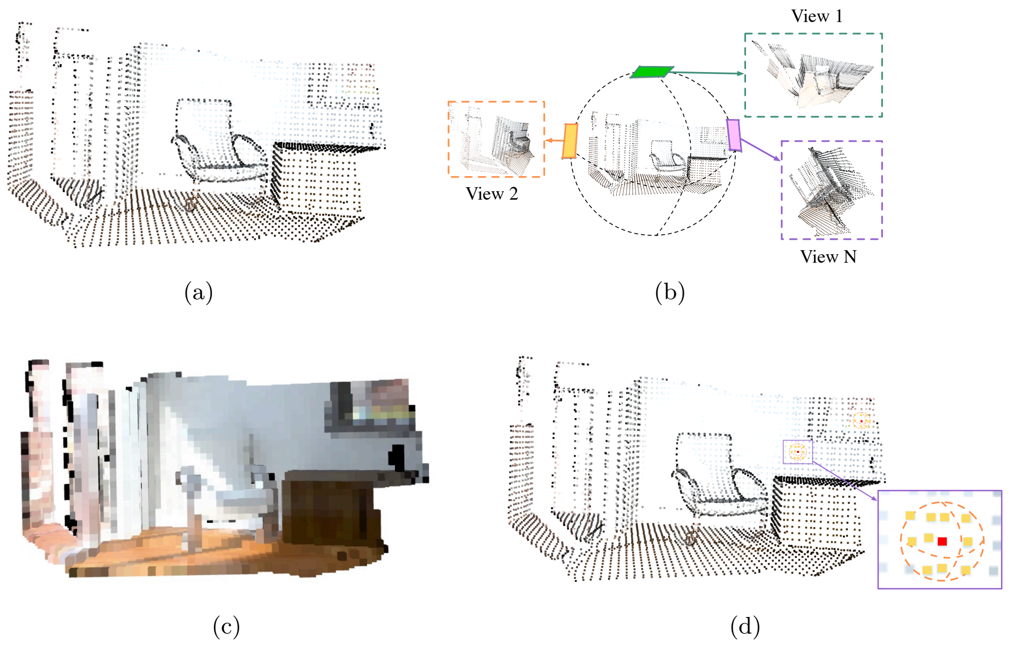


Figure 2.1: Three major kinds of deep learning based methods for point cloud: a) the original point cloud b) multi-view based method c) volumetric-based method d) point-based method.

### 2.2.3 Point-based Methods.

Compared to view-based methods and volumetric-based methods, which generate unavoidable information loss, point-based methods skip any preprocess techniques such as voxelization and projection and directly handle with raw point clouds. Typical point-based methods usually include three procedures to obtain the embedding: *Sampling*, *Grouping* and *Encoding*. *Sampling* is to select a set of centroids from the original point cloud to reduce the memory cost. *Grouping* is to select a set of neighbors (also called agglomerates) for each centroid, which represents a local information and works like the local region constrained by a convolution kernel in the original convolution. *Encoding* is to encode the new centroid feature using the original one and the local feature aggregated from the neighbors of the centroid. In *Sampling* phase, widely used sample rules include *Farthest Point Sampling* (FPS) [88, 111, 122, 71], random sampling [68], Poisson disk sampling [43, 99], and VoxelGrid sampling [92, 100]. In *Grouping* phase, ball query grouping [89, 59, 43, 71] and k nearest neighbors (KNN) [68, 111, 121, 37] are the two dominant methods. As for methods used in *Encoding* phase, previous survey [39] lists three main types: MLP-based, convolution-based, and graph-based methods.

*MLP-based methods* use Multi-Layer Perceptrons (MLPs) as the backbone to extract hidden features. Among all these methods, PointNet [88] is the pioneering work which simply calculates point-wise features independently, causing a loss of neighborhood information. Following this way, many efforts have been made to further increase its representation power. For example, PointNet++ [89], the extension work of PointNet, captures neighborhood information by a hierarchical model on the basis of MLP. Besides solely using a MLP module, some works use attention technique to explore the relational information, i.e., relations between neighbors and centroid [122], and between local structures [30].

*Convolution-based methods* process the centroid along with the neighborhood by a continuous [71, 14] or discrete [68, 47] convolutional kernel, i.e., a weighted sum over a given subset related to the centroid and the neighborhood. Among all convolution-based methods, PointCNN [68] is the most representative one. It does not use a symmetric function to keep the order invariance property. Instead, the relative coordinates are adopted to obtain a transformation matrix $X$, which transforms the relative coordinates into a latent canonical form and thus achieves the order invariance. Then the transformed input is processed by a convolution operation.

*Graph-based methods* treat each point in the point cloud as the vertex in the graph and connects each centroid with its selected neighbors. In this way, the original point cloud is viewed and processed as a graph, in which GNNs can be utilized. Similar with GNNs, graph-based methods also develop two different directions for feature learning: spatial domain and spectral domain. Methods in spatial domain obtain point embeddings following a convolution philosophy, i.e., recursively updating node features by aggregating information from neighbors repeatedly. The pioneer work [100] simply updates node feature by calculating the average value of all neighbor features processed by a filter-generating network $F$, e.g. MLP. Another work [123] designs an auto-encoder, in which the graph-based encoder replaces the average term as a max-pooling operation. Evolved from the static graph, DGCNN [111] proposes a dynamic graph construction method that updates the neighbors after each layer of the network. Methods in spectral domain implements the *Encoding* from a spectra perspective. For example, RGCNN [104] defines the convolution over graph by Chebyshev polynomial approximation and Wang *et al.* [107] update node features by standard unparametrized Fourier kernels.

For a clear comparison, we list most state-of-the-art point-based methods in Table 2.1. Some methods are not covered in the table since they do not follow a typical sampling-grouping-encoding scheme. The method shown in the table may only represent a part of the method. For example, FoldingNet [123] described in Table 2.1 solely stands for the encoder in the work.

## 2.3 Graph Coloring Problems

The graph coloring problem is crucial in domains ranging from network science and database systems to VLSI design. Here, we classify previous coloring methods as learning-based methods and non-learning-based methods.

### 2.3.1 Non-learning-based methods.

As a classical problem in the NP-hard classes and graph theory, graph coloring problem has received considerable attention in past decades. Here, we only cover some representative non-learning-based methods that are related to our method from some perspectives. Braunstein et al. [15] proposed Belief propagation (BP) and Survey propagation (SP) to solve the $k$-coloring problem, where both methods belong to a message-passing scheme. The key idea is that, each node is randomly assigned a probability distribution of colors, then the probability is updated based on the probabilities of neighbors. Formally, define $\eta^k_{e\to u}$ as the probability that edge $e = \{u, v\}$ refutes $u$ as the color $k$, for the $k$-coloring problem, $\eta^k_{e\to u}$ is updated by:

$$\eta^k_{e\to u} = \frac{\prod_{v'\in\mathcal{N}(v)/\ u}(1 - \eta^k_{\{v,v'\}\to v})}{\sum_{r=1}^{k}\prod_{v'\in\mathcal{N}(v)/\ u}(1 - \eta^r_{\{v,v'\}\to v})} \tag{2.1}$$

The numerator indicates the possibility that $v$ is colored by the color $k$ (without considering node $u$). And the fraction is for normalization, making that the sum equals to one. The SP procedure is a little different, it did not normalize the probability directly, but introduced a joker state, $\eta^\star_{e\to u}$, representing that the edge can not refute any colors, i.e., $1 = \eta^\star_{e\to u} + \sum_{r=1}^{k}\eta^r_{e\to u}$. The method is simple and very close to our non-training version. However, the method is more theoretical and less practical because it is easy to fall into a trivial solution, i.e., all edges are assigned into the joker state. Even though a non-trivial solution can be found, a large number of iterations may be required due to its randomness. Apart from message passing, Takefuji [103] proposed an Artificial Neural Network (ANN) based method for the four-coloring problem. The basic conclusion is that the probability distribution[1] can be updated by subtracting the aggregated probability distributions of neighbors, which aligned with the intuition for our parameter initialization.

### 2.3.2 Learning-based methods.

Although our work is the first one that tries and analyzes the power of GNNs in the graph coloring problem, there is a surge of learning-based methods for coloring. Lemos et al. [60] integrated Recurrent Neural Networks (RNNs) into the message passing framework, i.e., two RNNs were employed to computes the embedding update from aggregated messages for each vertex and color. Finally, the graph embedding was used to predict the chromatic number, and the node embedding was used to predict exact node color by clustering. However, the clustering-based method generated a prohibitive conflict number as shown in Table 1 of our paper, making the method not practical. Huang et al. [48] introduced a fast heuristics coloring algorithm using deep reinforcement learning. For each step (state), the model predicts the next node and its best color solution with a win/lose feedback. The prediction depends on previous coloring results and a graph embedding generated

[1]In their work, the node attribute is not probability distribution but a binary vector indicating the selected colors. Nevertheless, the conclusion still holds for a probability case.

by an LSTM. The method can give relatively accurate coloring results, but still only comparable with some simple heuristic algorithms such as dynamic order coloring. On the contrary, our supplementary results in Appendix .5.2 demonstrates that our method outperforms these simple heuristic algorithms significantly. Zhou et al. [137] also borrowed the idea of reinforcement learning. However, the proposed method did not use a training scheme: the actions towards the environment is defined by a deterministic update function of the coloring probability distributions. The method achieves a state-of-the-art result quality. However, the framework contains a descent-based local search with a portion of randomness, which requires a repeated execution with different random seeds. Moreover, the local search algorithm may require extensive iterations to find a local optimum. Due to these limitations, the method suffers from the runtime, in their experiments, the coloring process for a 500-node graph costs more than 100 seconds.

□ **End of chapter.**

# Chapter 3

# Multiple Patterning Lithography

## 3.1 OpenMPL: An Open Source Layout Decomposer

Multiple patterning lithography has been widely adopted in advanced technology nodes of VLSI manufacturing. As a key step in the design flow, multiple patterning layout decomposition (MPLD) is critical to design closure. Due to the $\mathcal{NP}$-hardness of the general decomposition problem, various efficient algorithms have been proposed with high-quality solutions. However, with increasingly complicated design flow and peripheral processing steps, developing a high-quality layout decomposer becomes more and more difficult, slowing down further advancement in this field.

To reduce the repeated effort in the reimplementation of the whole decomposition framework and lower the bar of research on MPLD, we present OpenMPL as an open platform for developing MPLD algorithms. OpenMPL contains efficient implementations of widely adopted graph simplification techniques and state-of-the-art layout decomposition algorithms. We carefully design the software architectures and APIs to decouple the innovations on the core optimization steps. For example, one can focus on developing novel graph simplification or decomposition techniques without worrying about the peripheral processing issues as the platform provides clean and well-defined APIs for the kernel optimization engines.

Moreover, considering that the framework is well decoupled, which makes each step separated clearly, we can inspect individual algorithm or technique easily. Through the inspections, a set of issues are discovered and corresponding solutions to these issues are proposed in OpenMPL. Specifically, there are three possible issues which can be further improved: (1) There exist some redundant stitches which can be removed without decomposition quality loss; (2) The original problem formulation and corresponding ILP method cannot quantify the cost accurately, which makes the previous ILP-based algorithm sub-optimal; (3) The original exact cover (EC)-based algorithm fails to obtain the optimal solution in some cases. All these issues are well described and solved in this paper. Our contributions are highlighted as follows:

- We present OpenMPL [6], an open-source layout decomposition framework, with efficient implementations of various state-of-the-art simplification and decomposition algorithms.

- We prove the stitch candidate redundancy in the state-of-the-art stitch generation algorithm and propose a corresponding solution.
- We find the sub-optimality in the widely-adopted ILP formulation and propose an optimized ILP-based algorithm with improved performance.
- We improve the exact cover (EC)-based algorithm by some techniques which were not revealed and studied in the previous work.
- We conduct experiments on widely-recognized benchmarks and new large-scale designs derived from the latest ISPD'19 benchmark suites. The results demonstrate the effectiveness of our proposed algorithms and techniques.

The rest of this section is organized as follows. Section 3.1.2 gives the problem formulation and discusses the design principles, the workflow, and some other properties of OpenMPL. Section 3.1.3 discusses the redundancy of the stitch candidate and gives the corresponding stitch redundancy removal algorithm. Section 3.1.4 provides the non-optimal cases generated by the previous ILP-based algorithm and the updated optimized ILP-based algorithm is proposed. Section 3.1.5 introduces the drawbacks of the previous EC-based algorithm in some cases and proposes the optimized EC-based algorithm. Section 3.1.6 lists comprehensive experimental results.

### 3.1.1 Preliminary

Multiple patterning layout decomposition (MPLD) has been adopted to enhance the lithography resolution. The key idea of MPLD is to assign features that are close to each other to different masks, such that these features are far away enough to be printed with existing lithography techniques. MPLD can be divided into double patterning layout decomposition (DPLD), triple patterning layout decomposition (TPLD) and quadruple patterning layout decomposition (QPLD), according to the number of masks. This problem is difficult since it is a variation of the graph coloring problem, which is $\mathcal{NP}$-hard for $k \geq 3$, where $k$ is the number of colors (masks).

Figure 3.1 is an example of TPLD, where different colors represent different masks and the stitch candidates are highlighted in blue.. Figure 3.1(a) is the input layout feature; Figure 3.1(b) is the constructed layout graph without stitch candidate generation, which is a 4-clique and therefore not 3-colorable; Figure 3.1(c) is the constructed layout graph with stitch candidate generation. Two stitch candidates are introduced and the original 4-clique is dismissed; Figure 3.1(c) is the coloring result on the layout graph with stitch candidate generation. The final decomposed layout with three masks (each color corresponds to one mask).

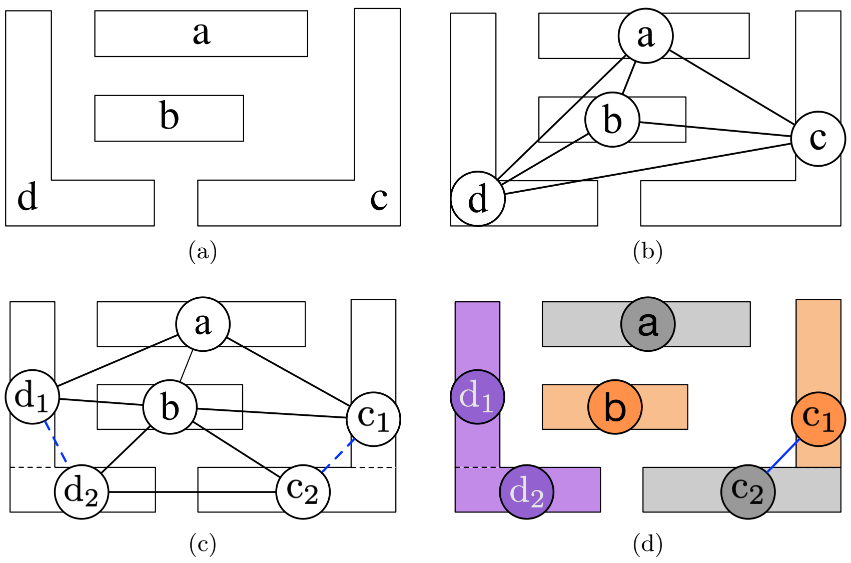


Figure 3.1: An example of TPLD with stitches.

Unlike the classical graph coloring problem, the MPLD problem has several unique characteristics. 1) Stitch: a polygon feature is allowed to be split into multiple overlapping segments to resolve coloring conflicts, as shown by the dashed edge in Figure 3.1(c). 2) Special patterns: there are different kinds of special features in a circuit layout, e.g., alternative power and ground lines, which may help to simplify the graph. 3) Complex rules: besides the widely adopted spacing constraint for the same color, there are also other rules. The different color spacing constraints [20] are related to the ordering of masks. That is, these constraints pre-determine the colors of some features before decomposition. All above characteristics impose different challenges to the MPLD problem, thus specialized algorithms are in demand to solve the MPLD problem effectively and efficiently.

To achieve high efficiency and to maintain high solution quality, a variety of decomposition algorithms have been proposed. These algorithms can be roughly categorized into three types [84, 76]: mathematical programming and relaxation, graph-theoretical approaches, and search-based approaches. Mathematical programming solves the MPLD problem by formulating it into a standard optimization model, such as integer linear programming (ILP) for DPLD [120, 51, 133] and TPLD [131, 128, 130]. Due to the $\mathcal{NP}$-hardness of TPLD and QPLD, a set of relaxation techniques such as semidefinite programming (SDP) [131], linear programming (LP) [70], and discrete relaxation method [67] are proposed based on ILP. Another category is to directly perform color assignment based on a set of graph-theoretical algorithms, e.g., the maximal independent set (MIS) [32], the shortest-path [23, 105], and fixed-parameter tractable (FPT) algorithms [58]. Search-based algorithms follow a divide-and-conquer principle with each sub-graph containing a small number of nodes, e.g., less than 20. Then a search procedure is applied to find the optimal solutions for small sub-graphs [57, 32, 135, 131, 17, 33]. Besides the researches on the single layout decomposition stage, recent work [75, 136] pioneers a new direction that integrates layout decomposition and mask optimization seamlessly, achieving compelling results from a global view of the solution space.

No matter how efficient the decomposition algorithm is, the $\mathcal{NP}$-hardness of TPLD and QPLD still makes the problem suffer from the runtime issue, especially when the graph size is large. Therefore, many graph simplification techniques have been developed to reduce problem size. The representative techniques include independent component computation (ICC) [131], iterative vertex removal (IVR) [57, 131], biconnected component extraction (BCE) [51, 133] and sub-K4 structure merging for TPLD [70].

### 3.1.2 The OpenMPL Framework

In this subsection, we first formulate the MPLD problem, which is the target of OpenMPL. Then, we introduce OpenMPL by covering the design principles, workflows, and functionalities. Finally, some additional features of OpenMPL are discussed.

#### Problem Formulation

The general MPLD problem can be formulated as follows:

**Problem 1** (MPLD)**.** *Given 1) a routed layout which is a set of polygonal features; 2) the number of masks $k$; 3) the minimal conflict space $d$; 4) other constraints like pre-coloring constraints, the goal is to assign one or more masks (if the stitch is enabled) to each feature so that the weighted sum of conflict cost and stitch cost is minimized.*

#### Design Principles

OpenMPL is designed for end-users, developers, and researchers as a general platform for the MPLD algorithms. Therefore, we emphasize usability, efficiency, and extensibility during development. The core design principles are highlighted as follows. (1) **Decoupled design stages**. The implementation clearly separates different optimization stages, as shown in Figure 3.2. Therefore, the interdependence between them is minimized. In this way, developers can focus on verifying individual stages without worrying about cross-stage impacts. (2) **Graph representations throughout the core stages**. After layout graph construction, the graph simplification, decomposition solver, and the simplified graph recovery stages use pure graphs as input/output, without involving mask data. This design leads to well-defined and highly separable core algorithms, making the framework highly extensible. (3) **Efficiency and generality for different mask data**. As a mask layer can be a contact layer or a metal layer, the processing efficiency varies significantly for different types of layers. We design a general mask database with separate processing routines for contact layers and metal polygon layers for efficiency enabled by C++ polymorphism since contact layers can be processed in a much simpler way.

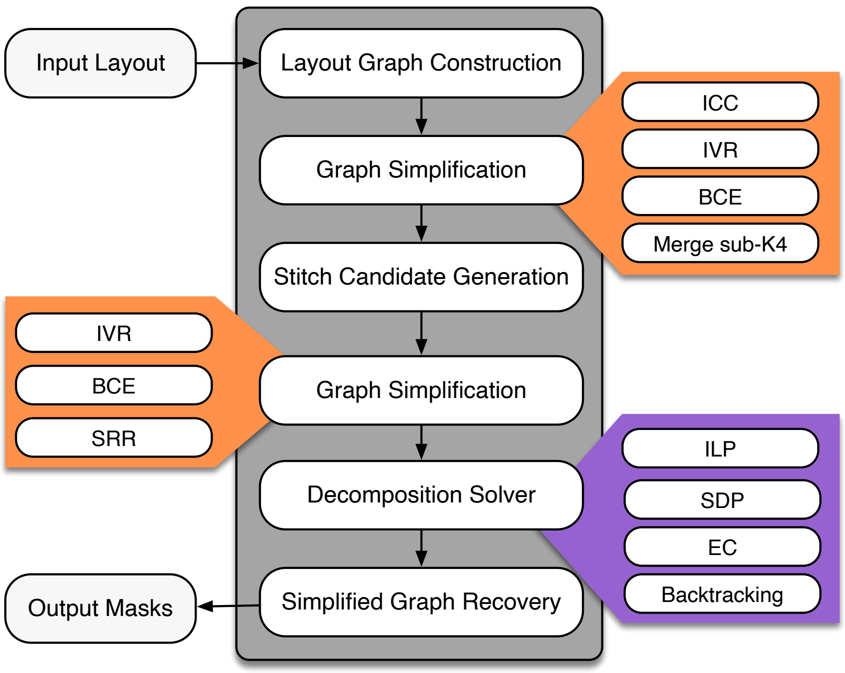


Figure 3.2: The workflow of OpenMPL.

### Workflow and Functionalities

The workflow of OpenMPL is illustrated in Figure 3.2. Firstly, one chip layout information (in GDS format) file is loaded and transformed into a layout graph (LG), which is represented by a vector of rectangle pointers, where the rectangles are defined in Boost [1]. Secondly, LG is simplified by some optional graph simplification techniques, where some of them are implemented in a third-party library Limbo [4]. Then, if stitch is enabled, the stitch insertion process [131] is executed to generate a decomposed graph (DG) with stitches. DG is further simplified by several simplification techniques. After the simplification, a coloring solver is called for each component in DG to solve the component coloring problem. Finally, our framework recovers nodes removed in the simplification step and assigns legal color for each removed node. In the following sub-subsections, we are going to introduce all of the functionalities in two crucial procedures of OpenMPL: graph simplification and decomposition.

Graph simplification techniques can be used to reduce the graph size and therefore reduce the computational complexity. Through layout graph simplification, we only need to deal with the smaller graph without affecting the final result. All of the simplification techniques mentioned in Section 3.1.1 are supported in our framework, including independent component computation (`ICC`), iterative vertex removal (`IVR`), biconnected component extraction (`BCE`), and sub-K4 structure merging for TPLD (`Merge sub-K4`). `ICC` is proposed based on the fact that there are many isolated clusters in a real layout, which enables `ICC` to break down the layout graph into several independent components.

IVR temporarily removes the nodes whose degree is less than the number of colors in an iterative manner. BCE simplifies the graph by duplicating the bridge vertices and then removing the bridge edges. Merge sub-K4 detects and merges specific structures whose number of edges is exactly one less than four-clique structures and thus is only applicable for TPLD. Except Merge sub-K4, other implemented simplification techniques support any number of masks. Besides these simplification methods, we develop a simplification method which focuses on the removal of redundant stitches. The details are shown in Section 3.1.3. Different simplification techniques require different recovery methods. However, those nodes which are shared among different components may be assigned different colors after recovery. To tackle this, color rotation [51] is implemented in our framework. Specifically, color rotation is to rotate the color assignments of the sub-graphs to avoid unnecessary conflict when coloring the whole layout graph from the sub-graphs.

Graph color assignment is the most crucial step in the flow, which impacts the final coloring results directly. In the graph color assignment, a simplified graph is provided and each vertex in the graph should be assigned one color by the specified algorithm. OpenMPL has supported all of the commonly-used algorithms in the layout decomposition and some updated algorithms are also implemented. The algorithms are briefly introduced in the following context:

- **Original Integer Linear Programming**: The details are covered in Section 3.1.4. We use Gurobi [40], Lemon [3], and CBC [2] as the ILP solvers.
- **Optimized Integer Linear Programming**: The details are covered in Section 3.1.4.
- **Semidefinite Programming**: The discrete integer programming solving process of Equation (3.9) is $\mathcal{NP}$-hard, thus it may suffer from run-time overhead for practical designs. As shown in [131, 79, 129], the color assignment can be formulated as a vector programming and then relaxed and solved by semidefinite programming in polynomial time. Given the solutions of SDP, a mapping process is used to map the solutions to coloring results. CSDP [13] is used as the SDP solver.
- **Backtracking**: Backtracking [131] is a DFS fashion algorithm used to find solutions in the whole solution space. Especially, we use a simple but effective heuristic technique to speed up the backtracking process. We set the upper bound of the cost as 0 at the beginning to cut branches more frequently and thus speed up the process. If no feasible solution is found under such an upper bound constraint, we relax the constraint by adding the bound to 1 and repeat the procedure until finding the optimal solution.
- **Original Exact cover-based algorithm**: The details are covered in Section 3.1.5. We implement the dancing links data structure and EC solver, instead of calling the third-party solver like ILP and SDP. Therefore, the runtime of the EC-based algorithm can be optimized further compared to other algorithms.

- **Flexible Exact cover-based algorithm**: The details are covered in Section 3.1.5.

OpenMPL also supports decomposition algorithms like maximal independent set (MIS) [32], linear programming (LP) [70], etc., which cannot decompose the graph containing stitch edges while working well on stitch-free graphs. Due to the page limit, we leave the details on the tool release page [6].

### Additional Features

Some additional features are supported for better usability, efficiency and extensibility. 1) OpenMPL supports **multi-threading** operations by OpenMP [5] and users can specify the number of threads. Graph components are solved in parallel and layout decomposition algorithms also support multi-threading computations; 2) We can identify all the possible positions of stitches through pattern projections [131] in **stitch insertion**, which is one of the most critical steps to parse a layout. One example of the stitch is shown in Figure 3.1. There are lots of candidate positions to insert a stitch, and only some are chosen as the final stitches. 3) In practice, a pattern in the layout may be a polygon or rectangle. Consequently, the storage may vary from case to case. OpenMPL provides a **shape-friendly** system considering this case and users can specify the shape, `POLYGON` or `RECTANGLE`, to guarantee the performance to avoid unnecessary calculations. For polygonal inputs, to simplify the storage structure design and save space, OpenMPL first decomposes the polygons to rectangles. After reading the whole input file, DFS is utilized to find connected components and re-union rectangles into polygons. For rectangle circuits, we directly store these patterns without further operations.

## 3.1.3 Stitch Redundancy Removal

In this subsection, we briefly introduce the widely used stitch candidate generation method and then propose an algorithm for stitch redundancy removal (SRR) with mathematical proof.

### Stitch Candidate Generation

The original layout does not contain stitch information, thus the framework for MPLD problem should determine the positions to insert stitches. One example of stitch can be found in Figure 3.1(c), where $c_1$-$c_2$ and $d_1$-$d_2$ are two generated stitch candidates. Previous works proposed solutions to generate candidate stitches for DPL [51, 120] and TPL [57, 128]. The key idea of stitch candidate generation is to project each feature into its neighbor features, where the projection results are then used to determine stitch candidates. For example, [57] proposed a heuristic algorithm to find all legal stitch positions in TPL using the projection results. Kahng *et al.* [51] used the projection sequence to directly carry out stitch candidate generation by some simple rules. One example of the stitch candidate insertion by projection sequence is shown in Figure 3.3, where the middle feature $a$ has three conflict features, $b$, $c$,and $d$. Based on the projection indicated by the

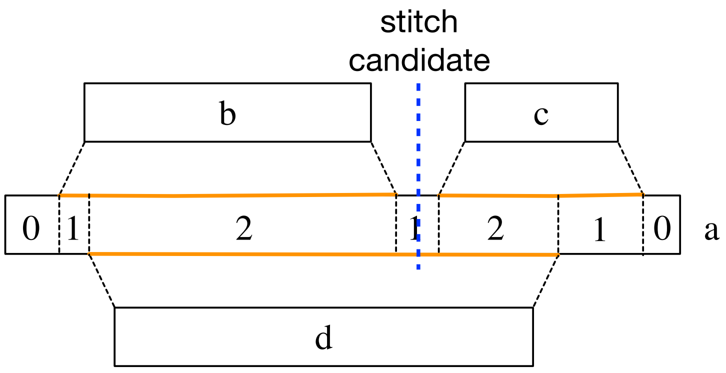


Figure 3.3: Projection results, where the projection sequence is 0121210 and the middle segment whose label is ”1” should be inserted a stitch for TPLD, which is highlighted by blue.

black dash line in Figure 3.3, the feature $a$ is divided into 7 segments. Each segment is labeled by the number of projected conflict features, then we can get its projection sequence: 01212101010. The rules of the projection sequence are different when the number of masks varies. The general rules of the projection sequence for TPLD can be summarized as follows [128]: If 1) the projection sequence contains sub-sequences whose value $xyz$ satisfies $x > y, z > y$; 2) the sub-sequence is not at the beginning or end of the projection sequence with form 01010, then the middle positions of $y$ should insert one stitch candidate. As shown in Figure 3.3, the middle feature $a$ has three conflict features, $b, c, d$. According to the rules stated above, one stitch candidate is inserted into $a$ as shown in the figure. In our implementation, such a stitch candidate generation approach supports any number of masks and therefore can be used for general MPL. However, when the mask number is larger than three, the stitch candidates may be redundant or missed since we haven’t considered special properties for larger mask numbers.

### Stitch Redundancy Removal

Although the current stitch candidate generation algorithm is able to find all possible stitches [131, 57], there are a few stitch candidates that are redundant after further graph simplification. One example is shown in Figure 3.4(a), where the edge $a$-$b$ is redundant, i.e., $a, b$ can be assigned with the same color without additional cost. To clarify the phenomenon, we define $C(u)$ as the cost of node $u$ and compute as:

$$C(u) = \sum_{i \in N^c(u)} c(i,u) + \alpha \sum_{i \in N^s(u)} s(i,u), \tag{3.1a}$$

$$\text{s.t.} \quad c(i,u) = \min\{\sum_{r_j \in p_i} (x_j == x_u), 1\}, \forall p_i \in N^c(u), \tag{3.1b}$$

$$s(i,u) = (x_i \neq x_u), \forall r_i \in N^s(u), \tag{3.1c}$$

$$x_i, x_u \in \{1, \ldots, k\}, \tag{3.1d}$$

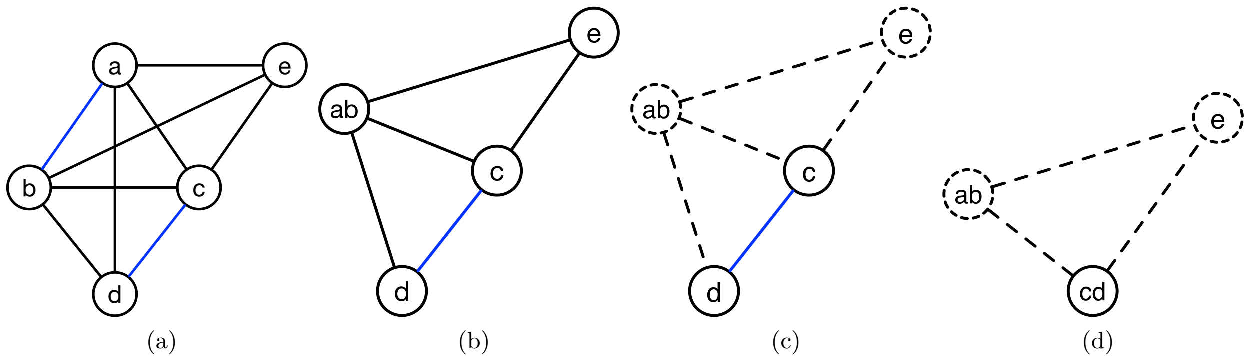


Figure 3.4: An TPLD example of stitch redundancy removal (SRR). The conflict edge is marked with black and the stitch edge is blue. Dotted edges/nodes are removed. (a) The decomposed graph before SRR. (b) Nodes $a$ and $b$ are merged into node $ab$. (c) Node $ab$ and node $e$ are further removed by IVR. (d) Node $c$ and node $d$ are merged into node $cd$.

where $N^c(u)$ is the neighbor feature set of $u$ connected by conflict edges, $N^s(u)$ is the neighbor node set of $u$ connected by stitch edges, the node/feature is defined by $r/p$ respectively and $x_i$ is a variable for the $k$ available colors of the node $r_i$. $x_j == x_u$ represents 1 if $x_i$ equals to $x_u$ and 0 if they are inequivalent. $x_i \neq x_u$ is defined in an opposite way. Take Figure 3.4(a) as an example, for the node $a$, $N^c(a) = \{cd, e\}$, $N^s(a) = \{b\}$, where $cd$ represents the original feature divided by the stitch edge $c$-$d$.

Given a coloring solution $f : V \to \{1, ..., k\}$, where $\{1, ..., k\}$ is the index set of the $k$ colors. $C_f(u)$ is the cost of node $u$ when $u$ is colored by $f$ and computed by:

$$C_f(u) = \sum_{i \in N^c(u)} c_f(i, u) + \alpha \sum_{i \in N^s(u)} s_f(i, u), \tag{3.2}$$

where $c_f(i, u)$ and $s_f(i, u)$ are defined similarly to $c(i, u)$ and $s(i, u)$ in Equation (3.1). The color $x_u$ for node $u$ when calculating $C_f(u)$ is given by $f$, i.e., $x_u = f(u)$. We have the following theorem about stitch redundancy:

**Theorem 1.** *Given a decomposed graph $G$, if there exists a stitch edge $e_s = \{u, v\}$ and the node pair $\{u, v\}$ satisfies three constraints:*

1. $N^c(u) = N^c(v)$;
2. $|N^s(u) \backslash v| \leq 1$;
3. $|N^s(v) \backslash u| \leq 1$,

*then at least one optimal coloring solution will assign the two nodes with the same color.*

*Proof.* The proof can be finished by contradiction. Assume that all of the optimal coloring solutions assign $u, v$ into two different colors. Let $f^*$ be one of the optimal coloring solutions and we have $f^*(u) \neq f^*(v)$. We will show that there is another coloring solution $f'$, which assigns $u, v$ into the same color and has at least the same cost with $f^*$ and thus makes $f'$ be the optimal coloring solution. Without loss of generality, we assume:

$$\sum_{i \in N^c(u)} c_{f^*}(i,u) \leq \sum_{i \in N^c(v)} c_{f^*}(i,v), \tag{3.3}$$

Then $f'$ is defined as follows:

$$f'(i) = \begin{cases} f^*(u), & \text{if } i = v; \\ f^*(i), & \text{otherwise.} \end{cases} \tag{3.4}$$

Since the only difference between $f^*$ and $f'$ is the color of $v$, the cost difference $\triangle$ between $f^*$ and $f'$ on $G$ is given by:

$$\triangle = C_{f^*}(v) - C_{f'}(v). \tag{3.5}$$

By Equation (3.2) and Equation (3.5), $\triangle$ can be further interpreted as:

$$\begin{aligned} \triangle = (&\sum_{i \in N^c(v)} c_{f^*}(i,v) - \sum_{i \in N^c(v)} c_{f'}(i,v)) \\ &+ \alpha(\sum_{i \in N^s(v)} s_{f^*}(i,v) - \sum_{i \in N^s(v)} s_{f'}(i,v)). \end{aligned} \tag{3.6}$$

For the first conflict term, combining the first constraint, Equation (3.3) and Equation (3.4), we have:

$$\sum_{i \in N^c(v)} c_{f^*}(i,v) \geq \sum_{i \in N^c(v)} c_{f'}(i,v). \tag{3.7}$$

For the second stitch term, the third constraint $|N^s(v) \backslash u| \leq 1$ indicates that: $\sum\limits_{i \in N^s(v) \backslash u} s_{f^*}(i,v) \leq 1$ and $\sum\limits_{i \in N^s(v) \backslash u} s_{f'}(i,v) \leq 1$. Moreover, we have $s_{f^*}(u,v) = 1 > s_{f'}(u,v) = 0$ since the colors of $u, v$ by $f^*$ are different. Therefore, we have:

$$\sum_{i \in N^c(v)} s_{f^*}(i,v) \geq 1 \geq \sum_{i \in N^c(v)} s_{f'}(i,v). \tag{3.8}$$

Combining Equation (3.6), Equation (3.7) and Equation (3.8), it is clear to see that $\triangle \geq 0$ always holds, which means that we can color $G$ by $f'$ without additional cost compared with the optimal solution $f^*$ and thus complete the proof. □

According to the theorem, we can conclude that all stitch edges satisfying constraints specified in Theorem 1 are redundant and corresponding node pairs can be merged to further simplify the graph. Motivated by this conclusion, we propose Algorithm 1 to

**Algorithm 1** StitchRedundancyRemoval

```
Input: $S \rightarrow$ Decomposed graph set.
for $DG \in S$ do
    NeedSimplification $\leftarrow$ False;
    for $s_{i,j} \in DG$ do
        if $\{i, j\}$ satisfies constraints in theorem 1 then
            $DG' \leftarrow$ Merge $i, j$ in $DG$;
            NeedSimplification $\leftarrow$ True;
        end if
    end for
    if NeedSimplification then
        $S' \leftarrow$ Simplified sub-graph set by simplifying $DG'$;
        StitchRedundancyRemoval($S'$);
    end if
end for
```

remove redundant stitch candidates. The algorithm is simply described as follows: after the stitch insertion and the graph simplification, the layout is divided and simplified into a decomposed graph set $S$. For each decomposed graph ($DG$) in $S$, the algorithm detects all stitch edges which satisfy the constraints specified in the theorem 1 (line 4) and merges all valid stitch edges (line 5). If $DG$ can be further simplified (line 10) after the removal of redundant stitch edges, the simplified graph set ($S'$) can be processed again (line 11) by Algorithm 1. One simple TPLD example is given in Figure 3.4. As shown in the example, the stitch edge $a$-$b$ is redundant and thus the node pair $\{a, b\}$ is merged (Figure 3.4(b)). After the removal of $a$-$b$, the graph can be further simplified by IVR (Figure 3.4(c)). Then, the stitch edge $c$-$d$ in the simplified graph is also redundant, and thus the node pair $\{c, d\}$ is merged (Figure 3.4(d)).

### 3.1.4 Optimized ILP-based algorithm

In this subsection, we first introduce the previous cost formulation and corresponding ILP-based algorithm proposed in [131], then the non-optimal case of such formulation is provided and discussed, followed by a new cost formulation and the corresponding optimized ILP-based algorithm proposed by us.

#### Original ILP-based Algorithm

Given an input layout specified by features in polygonal shapes, the layout can be translated into an undirected layout graph $G = (V, E)$, where every node $v_i \in V$ corresponds to one feature/sub-feature in the layout and each edge $e_{ij} \in E$ is used to characterize relationships between features. $E$ is composed of both conflict and stitch relationship, denoted by $E = \{CE \cup SE\}$, where $SE$ is the set of stitch edges and $CE$ is the set of conflict edges. One example is shown in Figure 3.1(c), where the stitch edges are orange and the conflict edges are black. Previous work [131] formulates the MPLD problem as below:

$$
\begin{aligned}
\min_{\boldsymbol{x}} \quad & \sum c_{ij} + \alpha \sum s_{ij}, && && \text{(3.9a)}\\
\text{s.t.} \quad & c_{ij} = (x_i == x_j), && \forall e_{ij} \in CE, && \text{(3.9b)}\\
& s_{ij} = (x_i \neq x_j), && \forall e_{ij} \in SE, && \text{(3.9c)}\\
& x_i \in \{0, 1, \ldots, k\}, && \forall x_i \in \boldsymbol{x}, && \text{(3.9d)}
\end{aligned}
$$

where $x_i$ is defined as in Equation (3.1), $c_{ij}$ is a binary variable representing the conflict edge $e_{ij} \in CE$, $s_{ij}$ stands for the stitch edge $e_{ij} \in SE$, $\alpha$, which is a user-defined parameter indicating the relative importance between the conflict cost and the stitch cost and set as 0.1 by default, If two nodes, $v_i$ and $v_j$, within the minimal coloring distance are assigned the same color, i.e., $x_i = x_j$, then $c_{ij} = 1$. On the contrary, $s_{ij} = 1$ when two nodes connected by the stitch edge are assigned different colors, i.e., $x_i \neq x_j$. The objective function is to minimize the weighted sum of the conflict number and the stitch number.

Based on the objective function shown in Equation (3.9), the problem can be solved by ILP [51, 131], where $x_i$ is represented by 1-bit 0-1 variable(s). The ILP model for TPLD can be formulated as in Formula equation 3.10, where the objective function of MPLD in Equation (3.9) can be directly applied in ILP-based formula, as shown in Equation (3.10a), constraints Equation (3.10c)–Equation (3.10g) play the same role as Equation (3.9b), where 0–1 variable $c_{ij}$ is true only if two nodes connected by the conflict edge $e_{ij}$ are assigned the same color.

$$
\begin{aligned}
\min \quad & \sum_{e_{ij} \in \mathrm{CE}} c_{ij} + \alpha \sum_{e_{ij} \in \mathrm{SE}} s_{ij} && && \text{(3.10a)}\\
\text{s.t.} \quad & x_{i1} + x_{i2} \leq 1, && && \text{(3.10b)}\\
& x_{i1} + x_{j1} \leq 1 + c_{ij1}, && \forall e_{ij} \in \mathrm{CE}, && \text{(3.10c)}\\
& (1 - x_{i1}) + (1 - x_{j1}) \leq 1 + c_{ij1}, && \forall e_{ij} \in \mathrm{CE}, && \text{(3.10d)}\\
& x_{i2} + x_{j2} \leq 1 + c_{ij2}, && \forall e_{ij} \in \mathrm{CE}, && \text{(3.10e)}\\
& (1 - x_{i2}) + (1 - x_{j2}) \leq 1 + c_{ij2}, && \forall e_{ij} \in \mathrm{CE}, && \text{(3.10f)}\\
& c_{ij1} + c_{ij2} \leq 1 + c_{ij}, && \forall e_{ij} \in \mathrm{CE}, && \text{(3.10g)}\\
& |x_{j1} - x_{i1}| \leq s_{ij1}, && \forall e_{ij} \in \mathrm{SE}, && \text{(3.10h)}\\
& |x_{j2} - x_{i2}| \leq s_{ij2}, && \forall e_{ij} \in \mathrm{SE}, && \text{(3.10i)}\\
& s_{ij} \geq s_{ij1}, s_{ij} \geq s_{ij2}, && \forall e_{ij} \in \mathrm{SE}, && \text{(3.10j)}\\
& x_{ij} \in \{0, 1\}. && && \text{(3.10k)}
\end{aligned}
$$

In Equation (3.10), $c_{ij}$ is true when both $c_{ij1}$ and $c_{ij2}$ are true by the constraint Equation (3.10g). 0–1 variable $c_{ij1}(c_{ij2})$ demonstrates whether $x_{i1}(x_{i2})$ equals to $x_{j1}(x_{j2})$. Therefore, $c_{ij}$ is true only when $x_i = x_j$, i.e., $v_i$ and $v_j$ are assigned the same color. Similarly, constraints Equation (3.10h) - Equation (3.10j) correspond to Equation (3.9c), where 0–1 variable $s_{ij}$ is true only if $v_i$ and $v_j$ are assigned different colors.

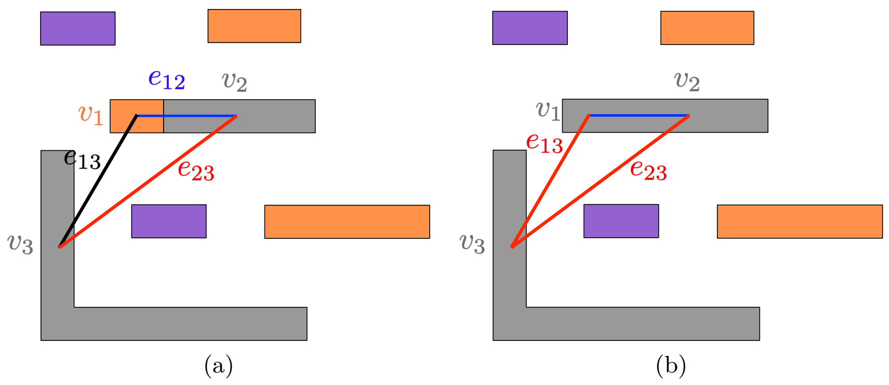


Figure 3.5: An example of the non-optimality of the original ILP-based algorithm. (a) The solution of the original ILP-based algorithm, where one stitch (blue line) happens at the top of the conflict (red line). (b) The solution of our ILP-based algorithm, where no stitch is introduced and can obtain the optimal solution.

### New ILP-based Algorithm

It is no doubt that the cost of the MPLD problem is the weighted sum of the conflict cost and the stitch cost. However, the previous ILP-based algorithm [131] measures the conflict cost by a summation of the binary variables representing conflict edges $e_{ij} \in CE$, i.e, $\sum c_{ij}$. Such a measurement method is not accurate and ignores a simple but important fact: conflict happens between features instead of nodes. In other words, If the stitch candidate divides one feature into two sub-features, which are represented by two nodes $v_1$, $v_2$ in the graph, and both nodes have a conflict edge with the third node $v_3$, i.e., $e_{12}, e_{13} \in CE$, then the previous conflict cost shown in Equation (3.10) will count both $e_{13}$ and $e_{23}$ while they represent the same conflict between features. Figure 3.5 illustrates one example, where the result of the original ILP, as shown in Figure 3.5(a), introduces one more stitch. The reason is: if the stitch edge $e_{12}$ is ignored as shown in Figure 3.5(b), i.e, the two connected nodes, $v_1$ and $v_2$, are assigned the same color and thus $x_1 = x_2$, the original cost function shown in Equation (3.10) will calculate the cost as 2 since both $e_{13}$ and $e_{23}$ are true. Therefore, ILP with the original problem formulation prefers to assign $v_1$ and $v_2$ with different colors, which results in a 1.1 cost value for the original cost function. However, it is easy to see that when this stitch is ignored, the conflict should be 1 instead of 2 since only one conflict between features happens.

Based on this observation, we present a new formulation shown in Equation (3.11). The objective function of the new formulation is the weighted sum of conflict cost ($\sum C_{mn}$) and stitch cost ($\sum s_{ij}$), which exactly matches the objective of the color assignment problem. The modified part is highlighted in blue. $P$ indicates the feature set before stitch insertion, $r_i$ and $r_j$ are the sub-features after stitch insertion and belong to $p_m$ and $p_n$ respectively. For example, $d_1$ and $d_2$ are the sub-features of the original feature $d$ in Figure 3.1.

Given the new formula for MPLD, the problem can also be solved by ILP. The ILP

$$
\begin{aligned}
\min_{\boldsymbol{x}} \quad & \sum C_{mn} + \alpha \sum s_{ij}, && (3.11a)\\
\text{s.t.} \quad & C_{mn} = \min\{ \sum_{\substack{r_i \in p_m,\\ r_j \in p_n,\\ c_{ij} \in \text{CE}}} (x_i == x_j), 1\}, \forall p_m, p_n \in P, && (3.11b)\\
& s_{ij} = (x_i \neq x_j), \quad \forall e_{ij} \in SE, && (3.11c)\\
& x_i \in \{0,1,2\}, \quad \forall x_i \in \boldsymbol{x}. && (3.11d)
\end{aligned}
$$

model for TPLD is formulated in Equation (3.12): here the conflict cost is calculated

$$
\begin{aligned}
\min \quad & \sum_{c_{ij} \in \text{CE}, r_i \in p_m, r_j \in p_n} C_{mn} + \alpha \sum_{e_{ij} \in \text{SE}} s_{ij}, && (3.12a)\\
\text{s.t.} \quad & x_{i1} + x_{i2} \leq 1, && (3.12b)\\
& x_{i1} + x_{j1} \leq 1 + C_{mn1}, \\
& \qquad \forall c_{ij} \in \text{CE}, r_i \in p_m, r_j \in p_n, && (3.12c)\\
& (1 - x_{i1}) + (1 - x_{j1}) \leq 1 + C_{mn1}, \\
& \qquad \forall c_{ij} \in \text{CE}, r_i \in p_m, r_j \in p_n, && (3.12d)\\
& x_{i2} + x_{j2} \leq 1 + C_{mn2}, \\
& \qquad \forall c_{ij} \in \text{CE}, r_i \in p_m, r_j \in p_n, && (3.12e)\\
& (1 - x_{i2}) + (1 - x_{j2}) \leq 1 + C_{mn2}, \\
& \qquad \forall c_{ij} \in \text{CE}, r_i \in p_m, r_j \in p_n, && (3.12f)\\
& C_{mn1} + C_{mn2} \leq 1 + C_{mn}, \\
& \qquad \forall c_{ij} \in \text{CE}, r_i \in p_m, r_j \in p_n. && (3.12g)
\end{aligned}
$$

by $\sum C_{mn}$ between the feature $m$ and $n$ instead of $\sum c_{ij}$ between node $i$ and $j$. In Equation (3.12), $C_{mn}$ is true when both $C_{mn1}$ and $C_{mn2}$ are true by the constraint formulated in Equation (3.12g). 0–1 variable $C_{mn1}(C_{mn2})$ demonstrates whether there exists $r_i \in p_m, r_j \in p_n, s.t.,\ x_{i1}(x_{i2}) = x_{j1}(x_{j2})$. By considering $C_{mn}$, the conflict cost between features instead of nodes, our new ILP-based algorithm is able to capture the conflict cost accurately.

### 3.1.5 Flexible Exact cover-based algorithm

In this subsection, we first introduce the exact cover (EC)-based algorithm proposed by [17], and then some non-optimal examples are discussed. Finally, we propose a flexible EC-based algorithm, which achieves a trade-off between quality and runtime and therefore outperforms the previous algorithm on the quality with a sacrifice in the runtime.

**Algorithm 2** ExactCoverSolver

```
Input: $G_p \leftarrow$ No-stitch graph;
Ourput: Coloring solution;
 1: Convert $G_p$ into exact cover matrix $M$;
 2: Call X* with $G_p$ and $M$;
 3: if X* exits with a solution then
 4:     return the found solution;
 5: else
 6:     Construct the stitch-inserted graph $G'_p$ based on $G_p$;
 7:     Construct the new exact cover matrix $M'$ based on $M$;
 8:     while no solution is found do
 9:         Call X* with $G'_p$ and $M'$;
10:         if X* exits without a solution then
11:             Remove the exact conflict edge in $G'_p$ and $M'$;
12:         end if
13:     end while
14:     return the found solution;
15: end if
```

**Exact Cover (EC)-based Algorithm**

Though our ILP is able to obtain the optimal solution of the objective function, it suffers from runtime for large graphs. EC-based algorithm [17] models the MPLD problem as an exact cover problem, which can be efficiently solved by a customized and augmented combination of dancing links data structure and Algorithm X* (DLX). Generally speaking, the layout is represented by a homogeneous graph. The graph is further translated into a 0-1 matrix and then can be solved as an exact cover problem of the obtained matrix. Each column index in the matrix can be viewed as the element of a universe $U$ to be covered, and each row can be viewed as a subset of the universe. The final solution (a set of rows) of the exact cover problem is then translated back to the solution (coloring results of each node) of the graph coloring problem.

The details of the EC-based algorithm are shown in Algorithm 2. The input of the algorithm is a no-stitch graph $G_p = \{V_p, E_p\}$, which is obtained from the layout features and each feature represents exactly one node in $G_p$. The algorithm first tries to solve the exact cover problem induced by the graph coloring problem on $G_p$, in which no stitch is introduced (lines 1–4). To be more specific, the target graph $G_p$ is translated into a corresponding exact cover matrix $M$ (line 1). Then, algorithm X* is called to solve $M$ (line 2). The details of algorithm X* are illustrated in [17]. If one feasible solution is found by X*, then the solution is returned (lines 3–4). Otherwise, the algorithm is going to solve the exact cover problem induced by the graph coloring problem on the stitch-inserted graph $G'_p$ (lines 5–15). Here, $G'_p = \{V'_p, E'_p\}$ is obtained by splitting the nodes in $G_p$ whose corresponding features have stitch candidates and new edges are added following the distance constraints (line 6). The algorithm then builds up the new matrix $M'$ based on $M$ (line 7) and calls algorithm X* to solve $M'$ (line 9). Furthermore, if graph $G'_p$ is still un-colorable, the detected exact conflict edge will be remarked as the reason for

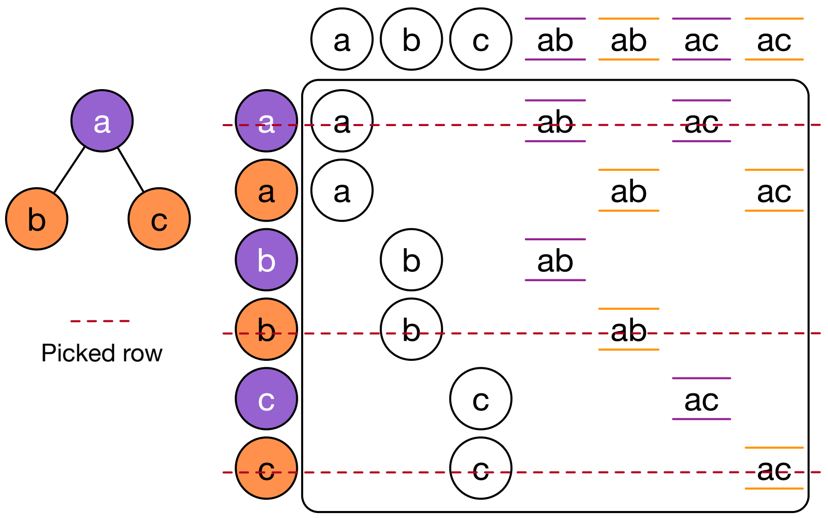


Figure 3.6: Double patterning instance with its exact cover matrix.

un-colorability and removed in $G'_p$ and $M'$ (line 11). Such a procedure is repeated until $G'_p$ is colorable and the final coloring solution is found (lines 8–13).

In the exact cover matrix $M$ translated from $G_p$, all nodes in $V_p$ are inserted into the universe $U$. In addition, for each edge $e \in E_p$, $k$ elements $e_c$, *s.t.* $c \in \{1, ..., k\}$ are inserted into $U$ for the $k$-coloring problem. Therefore, the total size of $U$ (also the column size of $M$) is $\mathcal{O}(|V_p| + k|E_p|)$. For each node $v \in V_p$, $k$ subsets $S^v_c$, *s.t.* $c \in \{1, ..., k\}$ corresponding to $k$ available colors are created, where each subset contains the node element $v \in U$ and $e_c$ for each edge $e = \{u, v\} \in E_p$. $e_c$ is inserted into both $S^v_c$ and $S^u_c$ and thus prevents $u, v$ from being assigned to the same color, which represents the conflict constraint between $u$ and $v$. Therefore, the total size of the subsets (also the row size of $M$) is $\mathcal{O}(k|V_p| + k|E_p|)$. One DPLD example of the translation from $G_p$ to $M$ is shown in Figure 3.6, where row 1,4,6 are selected as the final solution of the exact cover problem so that the corresponding coloring solution is given and shown in Figure 3.6.

In the converted matrix with stitch insertion, $M'$, besides the original rows (subsets) in $M$, additional rows are added below the original rows. Specifically, for each stitch candidate $e_c$, which splits the parent node $v \in V_p$ into two nodes $v'_1, v'_2 \in V'_p$, $k(k-1)$ subsets $S^v_{c_1 c_2}$, *s.t.*, $c_1, c_2 \in \{1, ..., k\}, c_1 \neq c_2$ corresponding to $k(k-1)$ available coloring solutions of $v'_1$ and $v'_2$ are created, where each subset contains the node element $v \in U$ and $e_{c_1}, e_{c_2}$ for each edge $e_{c_1} = \{v'_1, v'\} \in E'_p, e_{c_2} = \{v'_2, v'\} \in E'_p$. $e_{c_1}(e_{c_2})$ inserted in $S^v_{c_1 c_2}$ is to prevent $v'_1(v'_2)$ and $v'$ from being assigned to the same color. Therefore, the total number of newly-added rows is $\mathcal{O}(k^2|E_s|)$, where $|E_s|$ is the number of stitch candidates in all features of $G_p$. Figure 3.7 gives an example of $G'_p$ and $M'$, where the $7_{th}$ row is selected as the part of the final solution so that one stitch candidate is used to avoid the conflict. When graph $G'_p$ is still un-colorable, the exact conflict edge detected by algorithm X* is marked and removed. Such procedure is repeated until $G'_p$ is colorable and the final coloring solution is found.

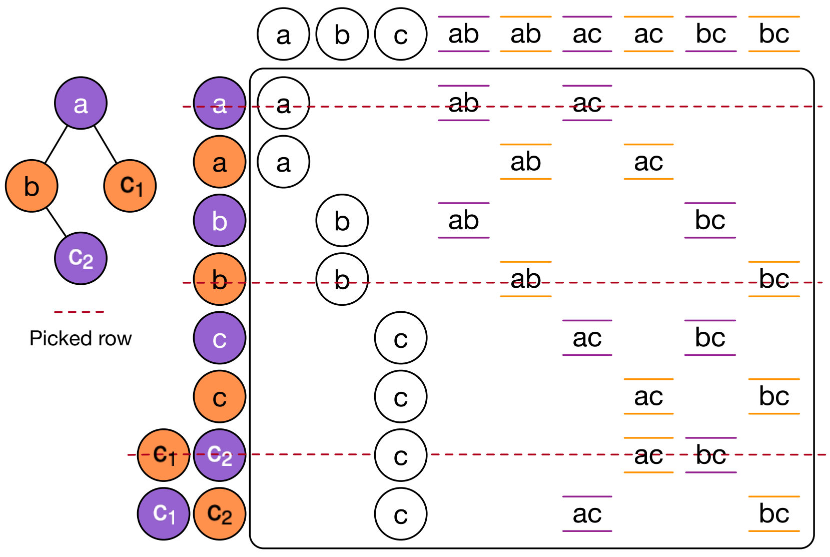
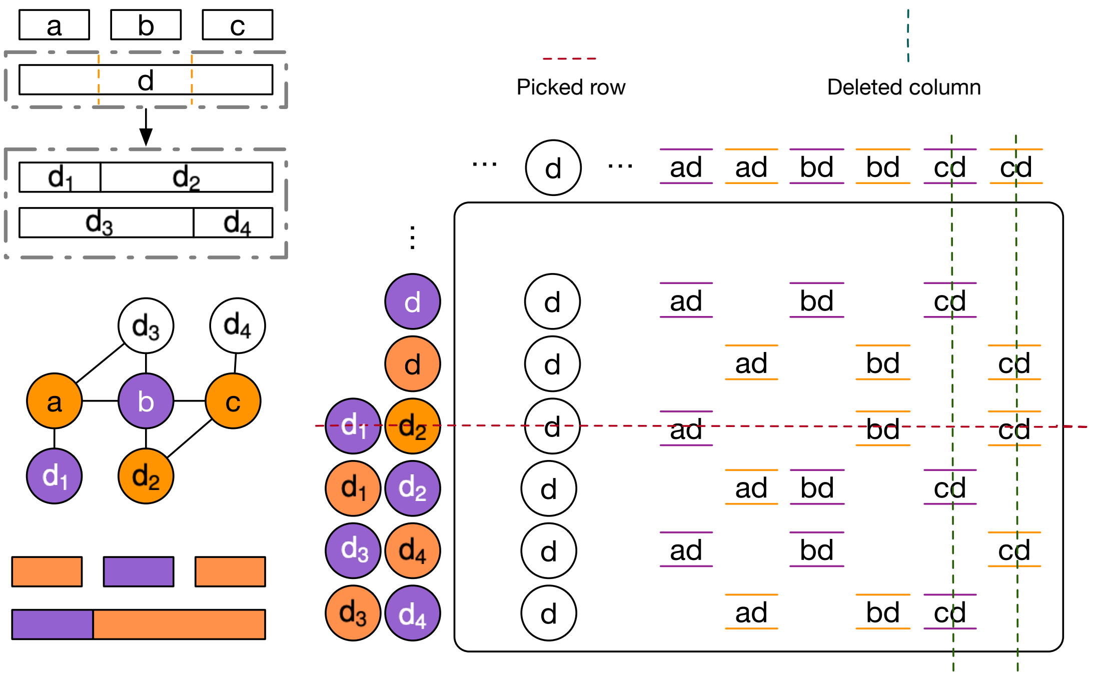


Figure 3.7: Double patterning instance containing stitch edge with its exact cover matrix.

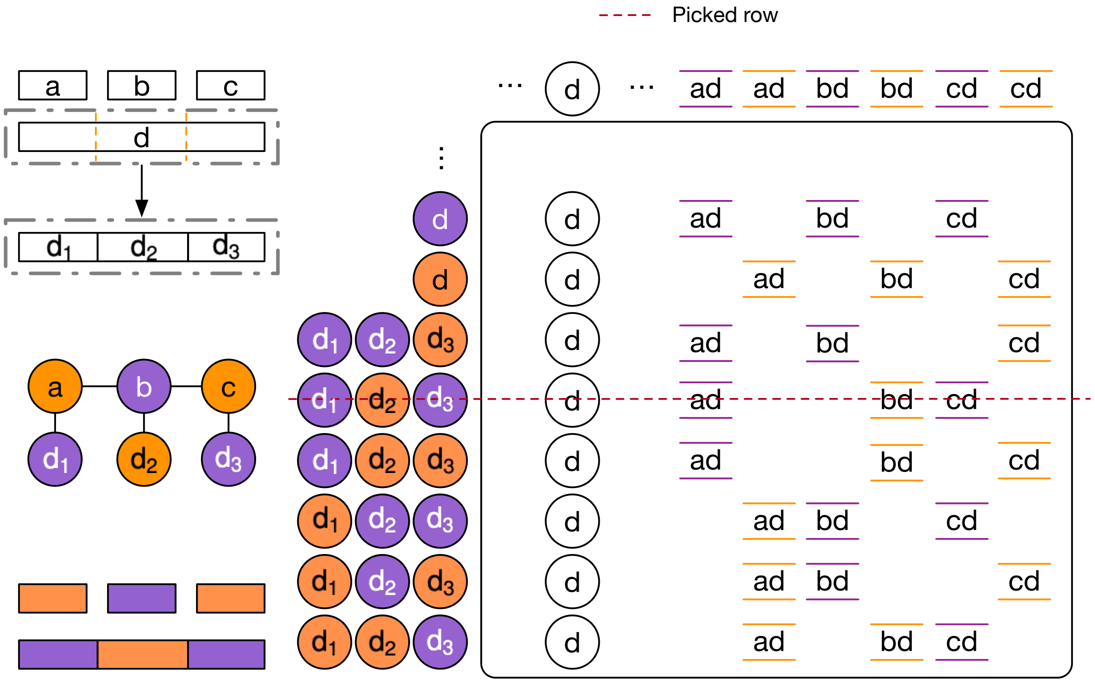
### Flexible Exact Cover-based algorithm

The exact cover-based algorithm shows impressive performance improvement due to the efficient augmenting DLX. However, the algorithm cannot always guarantee the optimality of the results. Here, we propose two techniques to improve the exact cover-based algorithm.

**Flexible Stitch Handling** The first possible reason for the non-optimality is the handling rules for stitch cases. Although the exact cover-based algorithm considers all stitch candidates on features concurrently, the example demonstrated in [17] uses at most one stitch to resolve conflict in a single feature since the rows in $M'$ are added in the unit of stitch candidate. However, there are some features in which multiple stitch candidates are able to resolve multiple conflicts. One example is shown in Figure 3.8, where our algorithm uses two stitches in the feature $d$ and generates a result with cost 0.2, while the original EC-based algorithm only uses one stitch and generates a coloring result with cost 1.1. Although some commercial decomposition tools based on [17] have considered multiple stitch cases to improve the solution quality, related techniques are not detailed in [17]. In OpenMPL, we formalize the flexible stitch handling method by introducing a maximally usable stitch candidate number $n$ and quantifying the complexity of the EC-based algorithm with different $n$.



(a)



(b)

Figure 3.8: (a) The non-optimal case of original EC-based algorithm. (b) The same case by our flexible EC-based algorithm, in which the result is optimal.

The direct reason for the non-optimality in the stitch handling is, the rows are added in the unit of the stitch candidate, which constrains at most one stitch candidate to be selected for each node. To overcome such constraint, i.e., to use an arbitrary number of stitch candidates in one feature, we handle the stitch in the unit of node element. We present our optimal stitch handling method as follows: denote the maximal number of usable stitch candidates as $n$, which is a controllable parameter. For each node element $v \in V_p$, if the corresponding feature of $v$ contains $t_v$ stitch candidates and thus the feature is divided into $t_v + 1$ sub-features, the original stitch handling approach is going to insert $t_v(k^2 - k)$ rows while we insert $C^m_{t_v}(k^{m+1} - k)$ rows, where $m$ is the number of used stitch candidates, which split the feature of $v$ into $m + 1$ sub-features and is calculated by the minimum value between $n$ and $t_v$, i.e., $m = \min\{n, t_v\}$. In our flexible algorithm, each row indicates one possible coloring solution for the divided $m + 1$ sub-features. Clearly, when $n$ equals one, the algorithm is the same as the previous one, i.e., only one stitch candidate is used in each feature and the space complexity is also $\mathcal{O}(t_v k^2)$. When $n$ becomes large enough, i.e., $m = t_v$, the algorithm will use all stitch candidates at the same time, which is more possible to be optimal. However, the large $n$ increases the space complexity to $\mathcal{O}(k^{t_v+1})$ and thus exponentially worsens the runtime. One DPLD example is shown in Figure 3.8, where Figure 3.8(a) is the matrix and corresponding coloring solution following the original stitch handling approach. $(d_1, d_2), (d_3, d_4)$ are the sub-features divided by two stitch candidates respectively and one conflict is introduced. Figure 3.8(b) shows the results for our flexible stitch handling, where $d_1, d_2$, and $d_3$ are the sub-features divided by two stitch candidates at one time, and all conflicts are resolved by stitches. Although the proposed stitch handling approach can obtain optimal results, it suffers from efficiency due to the explosion of the number of newly-added rows, especially when $n$ and $t_v$ are large. To speed up our algorithm without additional quality loss, we further use a heuristic technique. Firstly, the graph follows the original stitch handling approach, i.e., $n = 1$. If all conflicts are resolved by stitches or the graph contains no features whose number of stitch candidates is more than one, then the coloring procedure completes and the optimal stitch handling is not used. Otherwise, the graph is further handled by our flexible algorithm with $n$ as the maximal number of stitch candidates in the features, which is closer to the optimal solution.

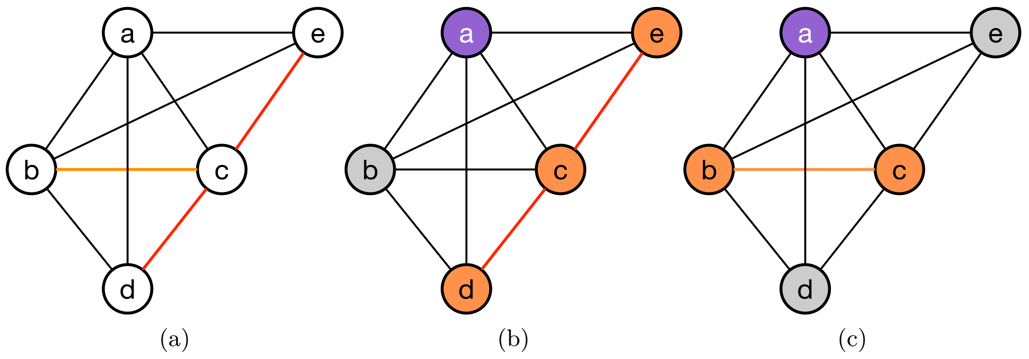


Figure 3.9: The non-optimal case of the original EC-based algorithm due to the traversal order. (a) The exact conflict(s) selected by the original rule (red) and ours (orange). (b) The coloring results by the original rule. (c) The coloring results by our optimized rule.

**Optimized Traversal Order** Another possible reason for the non-optimality is the traversal order of nodes for the conflict-overlapping cases. Here, we formallly define such a case as the **overlapping k-clique**:

**Definition 1.** *For a homogeneous graph $G = \{V, E\}$, where $V = \{V_s, v_1, v_2\}$, $G$ is called an **overlapping k-clique** if $(v_1, v_2) \notin E$ and the subgraphs $G_1 = G \backslash v_1$ and $G_2 = G \backslash v_2$ are both $k$-cliques where $k > 2$.*

One example of the overlapping 4-clique is given in Figure 3.9, where $V_s = \{a, b, c\}$, $v_1 = d$ and $v_2 = e$. With the definition, the following theorem shows the cost of the optimal solution for an overlapping k-clique in the $|V_s|$-coloring problem.

**Theorem 2.** *The optimal solution for an overlapping k-clique in the $|V_s|$-coloring problem has exactly one conflict.*

*Proof.* It is obvious that the optimal solution for a k-clique in the $|V_s|$-coloring problem has exactly one conflict since $|V_s| = k - 1$. Therefore, the optimal solution for both $G_1$ and $G_2$ has exactly one conflict, which results in a lower bound of the conflict number for graph $G$ as one. We then prove that there exists a feasible solution $f : V \to \{1, ..., k-1\}$ which colors $G$ within one conflict. Let's define $f$ as follows: Given any two different nodes in $V_s$, $v_1^s$ and $v_2^s$ , $f$ first assigns color 1 to $v_1^s$ and $v_2^s$ and color 2 to $v_1$ and $v_2$, which generates one conflict. Then $f$ assigns colors $\{3, ..., |V_s|\}$ to the left nodes in $V_s$, i.e., $V_s \backslash \{v_1^s, v_2^s\}$. Since the size of $V_s \backslash \{v_1^s, v_2^s\}$ is $|V_s| - 2$, which is the same as the size of available colors, this coloring procedure is conflict-free. Totally, the number of conflicts is one under this coloring scheme and such completes the proof. □

Although the minimum conflict of an overlapping k-clique is 1 as stated in Theorem 2, the quality of the results by algorithm X* is highly dependent on the traversal order. Original algorithm X* in [17] traverses the node in the BFS order unless one uncovered node has only one possible color. The root of BFS is the node whose corresponding feature has the largest area. If there are multiple available nodes, nodes will be selected following a numerical order in the implementation. However, such BFS-based traversal order may fail to obtain the optimal solution in some overlapping k-cliques as mentioned in [17]. Such a situation can be formally described as:

**Claim 1.** *The solution for an overlapping k-clique in the $|V_s|$-coloring problem by algorithm $X^*$ with the BFS-based traversal order proposed in [17] cannot guarantee optimality.*

*Proof.* The proof can be finished by a simple non-optimal case. Assume that $k > 2$, the corresponding feature of node $v_1$ has the largest area, which makes $v_1$ the root of BFS, and $v_2$ is the node at the end of the numerical order, then the detected exact conflict edge, i.e., the last reported conflict edge, must be the edge between $v_2$ and $v_i^s$, where $v_i^s$ is the node in $V_s$. Therefore, edge $\{v_2, v_i^s\}$ is removed and one conflict happens. However, the sub-graph $G_2$ is still a k-clique and contributes to one conflict in the $|V_s|$-coloring problem besides the edge $\{v_2, v_i^s\}$. Therefore, such a traversal order finally results in at least two conflicts totally, which is not optimal. $\square$

One example of non-optimality is shown in Figure 3.9(b). The edge *c*-*e* is first marked as an exact conflict and then one more conflict *c*-*d* is introduced because the left sub-graph *a*-*b*-*c*-*d* still forms a 4-clique. Considering the non-optimal case of original traversal order, we propose a heuristic traversal order which is nearer to the optimal solution. The differences of our optimized traversal order are organized as follows: (1) The root of BFS is the node with the largest degree; (2) If nodes are in the same depth in the BFS, the node with the smallest degree is selected; (3) If there are multiple uncovered nodes that have only one possible color, the node with the maximal degree is selected. Through these special treatments, the new traversal order is optimal for the k-clique and can be formally described as:

**Theorem 3.** *The solution for an overlapping k-clique in the $|V_s|$-coloring problem by algorithm $X^*$ with the new traversal order guarantees optimality.*

*Proof.* Let $v_i^s$ be the root, $v_i^s \in V_s$ since the root has the largest degree. Because both $v_1$ and and $v_2$ have the smallest degree, which will be selected firstly, the last detected conflict is $\{v_i^s, v_j^s\}$, where $v_j^s \in V_s$. After the edge $\{v_i^s, v_j^s\}$ is removed, both $G_1$ and $G_2$ are not k-cliques and can be colored by algorithm $X^*$ without additional conflict. Therefore, the total number of conflicts by algorithm X* with the new traversal order is one, which is optimal according to the Theorem 2 and completes the proof. $\square$

One example of the new traversal order is shown in Figure 3.9(c), where our new traversal order marks *b*-*c* as the exact conflict and thus achieves optimality.

### 3.1.6 Experimental Results

We implement OpenMPL in C++ and use Boost [1] as the basic graphics library. All of the experiments are tested on an Intel Core 2.9 GHz Linux machine. We conduct experiments

Table 3.1: Effective of stitch redundancy removal (SRR)

| Circuit | Our EC w/o. SRR | | Our EC w. SRR | |
|---|---|---|---|---|
| | time (s) | cost | time (s) | cost |
| `test1_100` | 2.163 | 385.9 | 2.866 | 385.9 |
| `test5_101` | 0.013 | 625.3 | 0.008 | 625.3 |
| `test6_102` | 1.441 | 352.8 | 1.331 | 352.8 |
| `test8_100` | 2.889 | 6238.1 | 2.254 | 6238.1 |
| `test9_100` | 5.064 | 9651.2 | 3.877 | 9651.2 |
| `test10_100` | 11.716 | 11129.3 | 9.783 | 11129.3 |
| average | 3.881 | 4730.433 | 3.353 | 4730.433 |
| ratio | 1.000 | 1.000 | 0.864 | 1.000 |

Table 3.2: Our ILP vs. Original ILP [131] on ISCAS benchmarks

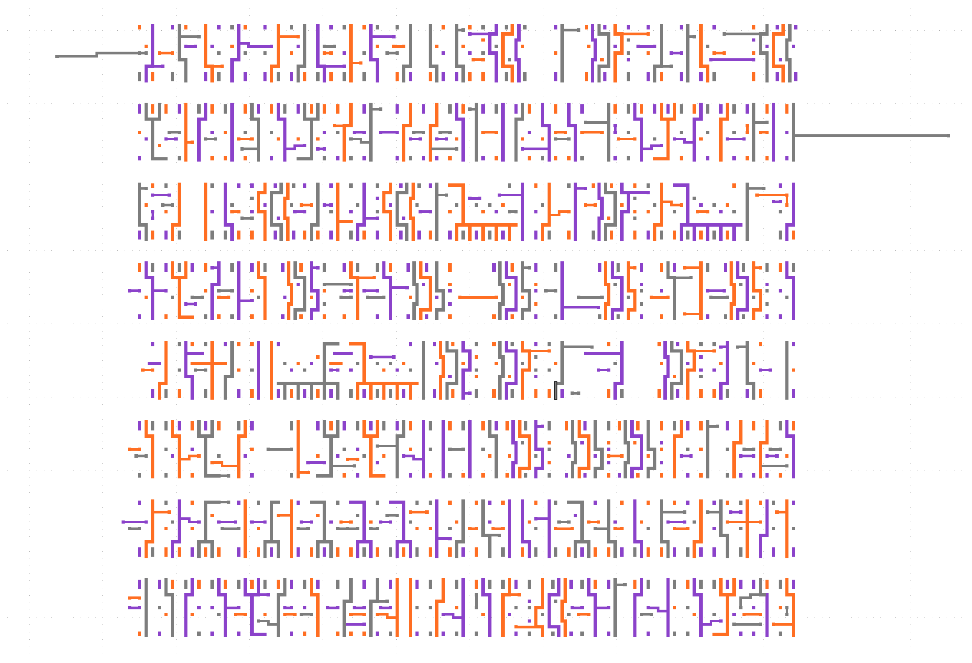
| Circuit | Original ILP[131] | | | | Our ILP | | | |
|---|---|---|---|---|---|---|---|---|
| | time (s) | st# | cn# | cost | time (s) | st# | cn# | cost |
| `C432` | 0.050 | 4 | 0 | 0.4 | 0.054 | 4 | 0 | 0.4 |
| `C499` | 0.029 | 0 | 0 | 0 | 0.017 | 0 | 0 | 0 |
| `C880` | 0.045 | 7 | 0 | 0.7 | 0.039 | 7 | 0 | 0.7 |
| `C1355` | 0.045 | 3 | 0 | 0.3 | 0.038 | 3 | 0 | 0.3 |
| `C1908` | 0.078 | 1 | 0 | 0.1 | 0.063 | 1 | 0 | 0.1 |
| `C2670` | 0.049 | 6 | 0 | 0.6 | 0.055 | 6 | 0 | 0.6 |
| `C3540` | 0.055 | 8 | 1 | 1.8 | 0.059 | 8 | 1 | 1.8 |
| `C5315` | 0.065 | 9 | 0 | 0.9 | 0.060 | 9 | 0 | 0.9 |
| `C6288` | 0.961 | 205 | 1 | 21.5 | 1.075 | 204 | 1 | 21.4 |
| `C7552` | 0.105 | 23 | 0 | 2.3 | 0.146 | 23 | 0 | 2.3 |
| `S1488` | 0.030 | 2 | 0 | 0.2 | 0.031 | 2 | 0 | 0.2 |
| `S38417` | 0.581 | 54 | 19 | 24.4 | 0.635 | 54 | 19 | 24.4 |
| `S35932` | 1.641 | 40 | 44 | 48 | 1.478 | 40 | 44 | 48 |
| `S38584` | 1.540 | 116 | 36 | 47.6 | 1.541 | 116 | 36 | 47.6 |
| `S15850` | 1.604 | 97 | 34 | 43.7 | 1.479 | 97 | 34 | 43.7 |
| average | 0.459 | 38.333 | 9.000 | 12.833 | 0.451 | 38.267 | 9.000 | 12.827 |
| ratio | 1.000 | 1.000 | 1.000 | 1.000 | 0.984 | 0.998 | 1.000 | 0.999 |

under two series of benchmarks. The first smaller benchmarks are the scaled-down and modified versions of ISCAS benchmarks, which are widely used in previous works. The minimum coloring spacing is set to 120 $nm$ for the first ten cases and 100 $nm$ for the last five cases, as in [32, 131, 17]. The second larger benchmarks are the ISPD '19 benchmarks for detailed routing. We use the metal layers in the benchmark obtained by Dr.CU 2.0 [62] and set the minimum coloring spacing as $k \cdot s + (k-1) \cdot w$, where $k$ is the number of colors and set as 3 in our experiments, $s$ is the minimum spacing between two features and $w$ is the standard width of one feature. Here, we only show the results of metal layers which

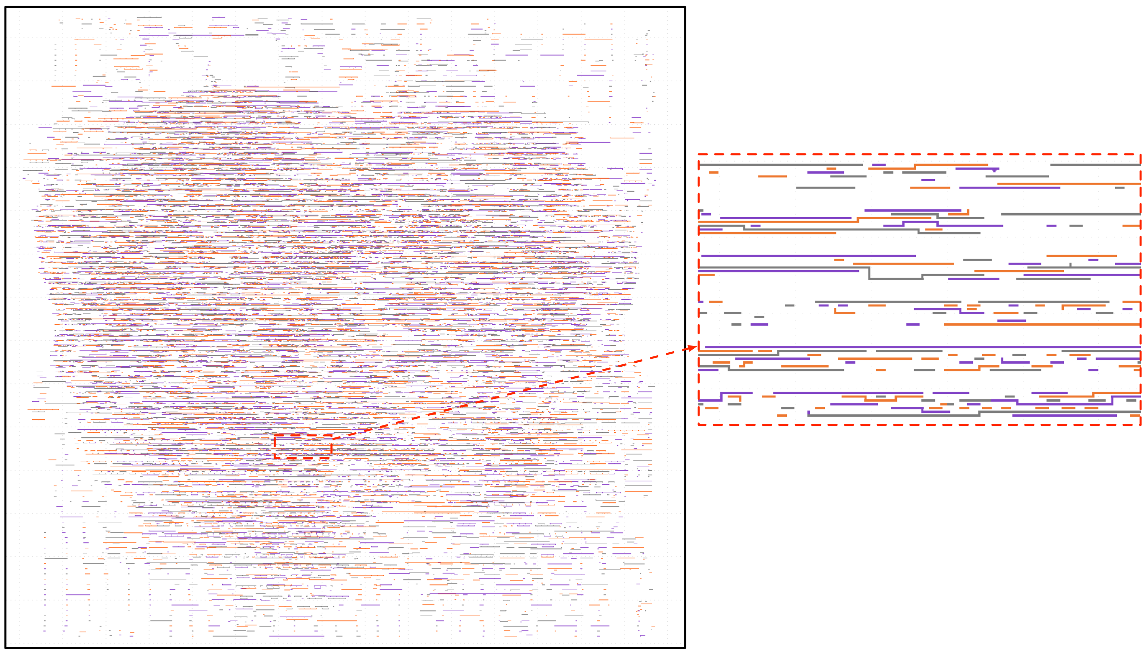
Table 3.3: Our ILP vs. Original ILP [131] on ISPD benchmarks

| Circuit | Original ILP[131] | | | | Our ILP | | | |
|---|---|---|---|---|---|---|---|---|
| | time (s) | st# | cn# | cost | time (s) | st# | cn# | cost |
| `test1_100` | 122.249 | 299 | 243 | 272.9 | 115.075 | 239 | 219 | 242.9 |
| `test5_101` | 118.029 | 232 | 466 | 489.2 | 149.040 | 190 | 433 | 452 |
| `test6_102` | 235.359 | 482 | 115 | 163.2 | 398.106 | 454 | 108 | 153.4 |
| `test8_100` | 19.025 | 4616 | 5683 | 6144.6 | 17.277 | 4389 | 5567 | 6005.9 |
| `test9_100` | 28.365 | 6969 | 8739 | 9435.9 | 25.534 | 6643 | 8559 | 9223.3 |
| `test10_100` | 110.479 | 9594 | 9775 | 10734.4 | 99.529 | 8945 | 9555 | 10449.5 |
| average | 105.584 | 3698.667 | 4170.167 | 4540.033 | 134.094 | 3476.667 | 4073.500 | 4421.167 |
| ratio | 1.000 | 1.000 | 1.000 | 1.000 | 1.270 | 0.940 | 0.977 | 0.974 |

can be decomposed by our ILP within three hours (6 cases in total). Each selected layer with id $n$ on benchmark $m$ is represented by `m_n`. For example, `test1_100` represents the layer with id 100 on the test1 benchmark of ISPD2019. We only focus on the results of different decomposition algorithms on the TPLD problem due to page limit, which is more difficult to obtain optimal results compared with DPLD. More detailed results and discussions can be found in [6]. The stitch weight $\alpha$ is set to 0.1, the thread number is 8 and the graph simplification level is 3 which represents that the framework enables three simplification techniques:`ICC`, `IVR`, and `BCE`. Especially, SDP is set to one thread due to no maintenance of CSDP now. Figure 3.10 shows the decomposition results for the case `C432` of ISCAS benchmarks and the case `test1_100` of ISPD benchmarks.



(a)



(b)

Figure 3.10: An example of the decomposition results. (a) Decomposition result of the circuit `C432` in the ISCAS benchmarks. (b) Decomposition result of the $100_{th}$ layer in the circuit `test1` in the ISPD benchmarks.

Table 3.4: Our EC vs. Original EC [17] on ISCAS benchmarks

| Circuit | Original EC[17] | | | | Our EC | | | |
|---|---|---|---|---|---|---|---|---|
| | time (s) | st# | cn# | cost | time (s) | st# | cn# | cost |
| C432 | 0.005 | 4 | 0 | 0.4 | 0.008 | 4 | 0 | 0.4 |
| C499 | 0.004 | 0 | 0 | 0 | 0.006 | 0 | 0 | 0 |
| C880 | 0.005 | 7 | 0 | 0.7 | 0.007 | 7 | 0 | 0.7 |
| C1355 | 0.007 | 3 | 0 | 0.3 | 0.018 | 3 | 0 | 0.3 |
| C1908 | 0.008 | 1 | 0 | 0.1 | 0.022 | 1 | 0 | 0.1 |
| C2670 | 0.014 | 6 | 0 | 0.6 | 0.021 | 6 | 0 | 0.6 |
| C3540 | 0.029 | 8 | 1 | 1.8 | 0.035 | 8 | 1 | 1.8 |
| C5315 | 0.019 | 9 | 0 | 0.9 | 0.033 | 9 | 0 | 0.9 |
| C6288 | 0.114 | 203 | 8 | 28.3 | 0.142 | 204 | 1 | 21.4 |
| C7552 | 0.028 | 21 | 1 | 3.1 | 0.055 | 21 | 1 | 3.1 |
| S1488 | 0.008 | 2 | 0 | 0.2 | 0.007 | 2 | 0 | 0.2 |
| S38417 | 0.127 | 54 | 19 | 24.4 | 0.175 | 54 | 19 | 24.4 |
| S35932 | 0.286 | 48 | 44 | 48.8 | 0.299 | 40 | 44 | 48 |
| S38584 | 0.291 | 117 | 36 | 47.7 | 0.323 | 117 | 36 | 47.7 |
| S15850 | 0.285 | 100 | 34 | 44 | 0.342 | 100 | 34 | 44 |
| average | 0.082 | 38.867 | 9.533 | 13.42 | 0.1 | 38.4 | 9.067 | 12.907 |
| ratio | 1.000 | 1.000 | 1.000 | 1.000 | 1.220 | 0.988 | 0.951 | 0.962 |

Table 3.5: Our EC vs. Original EC [17] on ISPD benchmarks

| Circuit | Original EC[17] | | | | Our EC | | | |
|---|---|---|---|---|---|---|---|---|
| | time (s) | st# | cn# | cost | time (s) | st# | cn# | cost |
| test1_100 | 1.772 | 236 | 383 | 406.6 | 11.521 | 279 | 358 | 385.9 |
| test5_101 | 3.229 | 282 | 615 | 643.2 | 21.052 | 303 | 595 | 625.3 |
| test6_102 | 7.209 | 560 | 327 | 383 | 57.525 | 558 | 297 | 352.8 |
| test8_100 | 5.585 | 4236 | 5994 | 6417.6 | 10.269 | 4561 | 5782 | 6238.1 |
| test9_100 | 9.042 | 6329 | 9270 | 9902.9 | 17.139 | 6852 | 8966 | 9651.2 |
| test10_100 | 15.14 | 8697 | 10621 | 11490.7 | 67.149 | 9433 | 10186 | 11129.3 |
| average | 6.996 | 3390 | 4535 | 4874 | 30.776 | 3664.333 | 4364 | 4730.433 |
| ratio | 1.000 | 1.000 | 1.000 | 1.000 | 4.399 | 1.081 | 0.962 | 0.971 |

### Effectiveness of Stitch Redundancy Removal

Firstly, we demonstrate the effectiveness of the proposed stitch redundancy removal (SRR) technique. Through stitch redundancy removal, some redundant stitch candidates can be removed and the two connected nodes are merged into one node. Therefore, the graph size for the decomposition is reduced and thus the decomposition runtime is decreased without decomposition quality loss theoretically. We only conducted SRR on the graphs whose sizes are larger than 8. Table 3.1 compares the performance and runtime on the target graphs. Column "time (s)" is the total simplification and decomposition runtime of graphs which have redundant stitches to be removed. The "cost" column is the total decomposition cost of our EC. When the case is sparse, i.e., the case can be easily simplified such that the total number of simplified graphs is huge while the size of each graph is usually small, our SRR may harm the runtime since the number of redundant stitches is not very much while the runtime for scanning all stitches in SRR cannot be avoided. For example, the runtime for graphs on `test1_100` is increased from 2.163 seconds to 2.866 seconds when SRR is used. Despite such a sparse case, which may not be the major bottleneck due to its low complexity, our SRR shows a considerable runtime improvement in most cases. We can see that compared with decomposing the graph by our EC directly, further applying SRR can reduce the average runtime by 13.6% without any performance loss.

**Original ILP Versus Our ILP**

Secondly, we compare our ILP with the original ILP proposed by [131] on both small ISCAS benchmarks and large ISPD benchmarks. The results are shown in Table 3.2 for ISCAS benchmarks and Table 3.3 for ISPD benchmarks. The column "time (s)" is the real time of decomposition in seconds instead of CPU time. Columns "st#" and "cn#" are the stitch number and the conflict number, "cost" is the decomposition cost calculated by Equation (3.12). On the small benchmarks, our ILP shows a slight improvement in both the runtime and the quality. The time is reduced by 1.6% and the stitch number is reduced by 1 on circuit `C6288` while the costs on other circuits are not changed, which indicates that such re-count case in the small benchmarks is not frequent. On the large benchmarks, Our ILP reduces 222 stitches and 96.67 conflicts averagely, i.e., from 3698.667 to 3476.667 and from 4170.167 to 4073.5. Therefore, the average cost is significantly reduced by 118.867 while the runtime is increased by 27%. However, such runtime loss is acceptable considering the unignorable quality improvement.

**Original EC Versus Our EC**

Thirdly, we compare our EC with the original EC proposed by [17] on both small and large benchmarks. The results are listed in Table 3.4 for ISCAS benchmarks and Table 3.5 for ISPD benchmarks. As discussed in Section 3.1.5, the original EC assumes exactly one stitch candidate to be activated for each feature, which reduces the matrix size and thus reduces the time complexity with a potential quality loss. Our EC assumes that at most $n$ stitch candidates are activated for each feature, where $n$ is a dynamic parameter and therefore we can achieve a flexible balance between runtime and quality by changing $n$. The results in both the small and large benchmarks demonstrate our analysis, where $n$ is set to 2 in our EC, i.e., at most two stitch candidates are activated for each feature.

Table 3.6: Non-stitch Decomposition Cost Comparison on ISCAS benchmarks

| Circuit | Our ILP | LP[70] | MIS[32] | SDP[131] | EC[17] | Back. [56] |
|---|---|---|---|---|---|---|
| C432 | 4 | 4 | 4 | 4 | 4 | 4 |
| C499 | 0 | 0 | 0 | 0 | 0 | 0 |
| C880 | 7 | 7 | 7 | 7 | 7 | 7 |
| C1355 | 3 | 3 | 3 | 3 | 3 | 3 |
| C1908 | 1 | 1 | 1 | 1 | 1 | 1 |
| C2670 | 6 | 6 | 6 | 6 | 6 | 6 |
| C3540 | 9 | 9 | 9 | 9 | 9 | 9 |
| C5315 | 9 | 9 | 9 | 9 | 9 | 9 |
| C6288 | 205 | 205 | 205 | 205 | 205 | 205 |
| C7552 | 22 | 22 | 22 | 22 | 22 | 22 |
| S1488 | 2 | 2 | 2 | 2 | 2 | 2 |
| S38417 | 95 | 97 | 95 | 95 | 97 | 95 |
| S35932 | 157 | 166 | 157 | 159 | 163 | 157 |
| S38584 | 230 | 233 | 230 | 231 | 231 | 230 |
| S15850 | 212 | 215 | 212 | 212 | 215 | 212 |
| average | 64.133 | 65.267 | 64.133 | 64.333 | 64.933 | 64.133 |
| ratio | 1.000 | 1.018 | 1.000 | 1.003 | 1.012 | 1.000 |

Table 3.7: Decomposition Cost Comparison on ISCAS benchmarks

| Circuit | Our ILP | | | SDP[131] | | | Our EC | | | Back. [56] | | | MIS [32] | | | LUT [57] | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | st# | cn# | cost | st# | cn# | cost | st# | cn# | cost | st# | cn# | cost | st# | cn# | cost | st# | cn# | cost |
| C432 | 4 | 0 | 0.4 | 4 | 0 | 0.4 | 4 | 0 | 0.4 | 4 | 0 | 0.4 | 6 | 0 | 0.6 | 4 | 0 | 0.4 |
| C499 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C880 | 7 | 0 | 0.7 | 8 | 0 | 0.8 | 7 | 0 | 0.7 | 7 | 0 | 0.7 | 8 | 1 | 1.8 | 7 | 0 | 0.7 |
| C1355 | 3 | 0 | 0.3 | 3 | 0 | 0.3 | 3 | 0 | 0.3 | 3 | 0 | 0.3 | 4 | 1 | 1.4 | 3 | 0 | 0.3 |
| C1908 | 1 | 0 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 0.1 |
| C2670 | 6 | 0 | 0.6 | 6 | 0 | 0.6 | 6 | 0 | 0.6 | 6 | 0 | 0.6 | 11 | 2 | 3.1 | 6 | 0 | 0.6 |
| C3540 | 8 | 1 | 1.8 | 8 | 1 | 1.8 | 8 | 1 | 1.8 | 8 | 1 | 1.8 | 11 | 3 | 4.1 | 8 | 1 | 1.8 |
| C5315 | 9 | 0 | 0.9 | 9 | 0 | 0.9 | 9 | 0 | 0.9 | 9 | 0 | 0.9 | 11 | 3 | 4.1 | 9 | 0 | 0.9 |
| C6288 | 204 | 1 | 21.4 | 203 | 7 | 27.3 | 204 | 1 | 21.4 | 205 | 1 | 21.5 | 243 | 20 | 44.3 | 191 | 14 | 33.1 |
| C7552 | 23 | 0 | 2.3 | 23 | 0 | 2.3 | 21 | 1 | 3.1 | 23 | 0 | 2.3 | 37 | 3 | 6.7 | 22 | 0 | 2.2 |
| S1488 | 2 | 0 | 0.2 | 2 | 0 | 0.2 | 2 | 0 | 0.2 | 2 | 0 | 0.2 | 4 | 0 | 0.4 | 2 | 0 | 0.2 |
| S38417 | 54 | 19 | 24.4 | 46 | 27 | 31.6 | 54 | 19 | 24.4 | 54 | 19 | 24.4 | 82 | 20 | 28.2 | 55 | 19 | 24.5 |
| S35932 | 40 | 44 | 48 | 20 | 64 | 66 | 40 | 44 | 48 | 40 | 44 | 48 | 63 | 46 | 52.3 | 41 | 44 | 48.1 |
| S38584 | 116 | 36 | 47.6 | 105 | 48 | 58.5 | 117 | 36 | 47.7 | 116 | 36 | 47.6 | 176 | 36 | 53.6 | 116 | 36 | 47.6 |
| S15850 | 97 | 34 | 43.7 | 83 | 48 | 56.3 | 100 | 34 | 44 | 97 | 34 | 43.7 | 146 | 36 | 50.6 | 100 | 34 | 44 |
| average | 38.27 | 9.00 | 12.83 | 34.73 | 13 | 16.47 | 38.4 | 9.07 | 12.91 | 38.33 | 9.00 | 12.83 | 53.48 | 11.47 | 16.75 | 37.67 | 9.87 | 13.63 |
| ratio | 1.00 | 1.00 | 1.00 | 0.91 | 1.44 | 1.28 | 1.00 | 1.01 | 1.01 | 1.00 | 1.00 | 1.00 | 1.39 | 1.27 | 1.31 | 0.98 | 1.10 | 1.06 |

$^*$ The results of MIS [32] and LUT [57] are directly quoted from their papers.

Our EC reduces the average cost by 3.8% on the small benchmarks and 2.9% on the large benchmarks. As a tradeoff, the average runtime is increased from 0.082s to 0.1s on the small benchmarks and from 6.996s to 30.776s on the large benchmarks, which is not trivial and demonstrates one of the drawbacks for the EC-based algorithm: the increase

Table 3.8: Decomposition comparison on ISPD benchmarks

| Circuit | Our ILP | | | | SDP [131] | | | | Our EC | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | st# | cn# | cost | time | st# | cn# | cost | time | st# | cn# | cost | time |
| `test1_100` | 239 | 219 | 242.9 | 115.075 | 287 | 269 | 297.7 | 4.929 | 279 | 358 | 385.9 | 11.521 |
| `test5_101` | 190 | 433 | 452 | 149.04 | 228 | 527 | 549.8 | 10.52 | 303 | 595 | 625.3 | 21.052 |
| `test6_102` | 454 | 108 | 153.4 | 398.106 | 477 | 144 | 191.7 | 67.856 | 558 | 297 | 352.8 | 57.525 |
| `test8_100` | 4389 | 5567 | 6005.9 | 17.277 | 4547 | 5750 | 6204.7 | 22.526 | 4561 | 5782 | 6238.1 | 10.269 |
| `test9_100` | 6643 | 8559 | 9223.3 | 25.534 | 6880 | 8842 | 9530 | 35.01 | 6852 | 8966 | 9651.2 | 17.139 |
| `test10_100` | 8945 | 9555 | 10449.5 | 99.529 | 9457 | 9963 | 10908.7 | 76.583 | 9433 | 10186 | 11129.3 | 67.149 |
| average | 3476.667 | 4073.5 | 4421.167 | 134.094 | 3646 | 4249.167 | 4613.767 | 26.237 | 3664.333 | 4364 | 4730.433 | 30.776 |
| ratio | 1.000 | 1.000 | 1.000 | 1.000 | 1.049 | 1.043 | 1.044 | 0.270 | 1.054 | 1.071 | 1.070 | 0.230 |

Table 3.9: Decomposition Runtime (s) Comparison on ISCAS benchmarks

| Circuit | Our ILP | SDP[131] | Our EC | Back. [56] |
|---|---|---|---|---|
| `C432` | 0.054 | 0.027 | 0.008 | 0.003 |
| `C499` | 0.017 | 0.016 | 0.006 | 0.004 |
| `C880` | 0.039 | 0.046 | 0.007 | 0.004 |
| `C1355` | 0.038 | 0.024 | 0.018 | 0.005 |
| `C1908` | 0.063 | 0.028 | 0.022 | 0.01 |
| `C2670` | 0.055 | 0.05 | 0.021 | 0.015 |
| `C3540` | 0.059 | 0.076 | 0.035 | 0.163 |
| `C5315` | 0.06 | 0.085 | 0.033 | 0.022 |
| `C6288` | 1.075 | 0.615 | 0.142 | 1.308 |
| `C7552` | 0.146 | 0.209 | 0.055 | 0.02 |
| `S1488` | 0.031 | 0.031 | 0.007 | 0.007 |
| `S38417` | 0.635 | 1.673 | 0.175 | 0.262 |
| `S35932` | 1.478 | 5.118 | 0.299 | 0.299 |
| `S38584` | 1.541 | 4.99 | 0.323 | 0.281 |
| `S15850` | 1.479 | 4.416 | 0.342 | 0.63 |
| average | 0.451 | 1.16 | 0.1 | 0.202 |
| ratio | 1.000 | 2.572 | 0.222 | 0.448 |

of $n$ results in an exponentially increasing on the runtime.

**Comparison of Different Decomposers**

Fourthly, we compare different decomposers in both stitch-enabled cases and no-stitch cases. The quality results without stitch for ISCAS benchmarks are listed in Table 3.6. Column "Back." is the result of the backtracking algorithm introduced in [56]. As shown in Table 3.6, our ILP, MIS [32], and backtracking [56] in our implementation obtain the optimal solution while other relaxation-based or heuristic methods degrade the result quality.

For the stitch-enabled cases, the quality comparison is shown in Table 3.7 and Table 3.8; The runtime comparison is shown in Table 3.9 and Table 3.8. Especially, backtracking is not shown in Table 3.8, since it cannot be processed within three hours for any layout in ISPD benchmarks. The results of MIS [32] and LUT [57] in Table 3.7 are directly quoted from their papers. The ratio is calculated based on the results of our ILP. For the decomposition cost, our optimized ILP outperforms other algorithms and achieves the best cost performance as expected. On the small benchmarks, SDP is the worst and increases the cost by 28.4% while the cost of our EC and backtracking are close to the ILP. On the large benchmarks, SDP only increases the cost by 4.4% and is better than our EC, which increases the cost by 7%. For the runtime, the original EC is the best due to the efficient augmenting DLX technique. Backtracking shows a good runtime performance on the small benchmarks due to our heuristic algorithm but fails to obtain the results on the large benchmarks within three hours. The runtime of SDP is much worse than our ILP on the small benchmarks, i.e., 2.572× runtime, while much better on the large benchmarks, whose ratio is close to our EC, i.e., 0.27 vs. 0.23.

## 3.2 Adaptive Layout Decomposition with Graph Embedding Neural Networks

The semiconductor industry nowadays is greatly challenged by extreme scaling which imposes severe issues on circuits manufacturing. Among various advanced lithography techniques, multiple patterning lithography (MPL) is one of the most practical solutions to enhance the manufacturability and has been widely adopted in industry [85].

The core problem of multiple patterning lithography is the layout decomposition which assigns features on a layout to separate masks for printability improvement and is also called multiple patterning lithography decomposition (MPLD). If two features located closer than minimum coloring distance are assigned to the same mask, a coloring conflict is introduced. Additionally, stitches can be inserted to assist conflict resolving, at a cost of potential yield loss though. Therefore, the objective of MPLD is to find a mask assignment for features such that the number of conflicts and stitches are minimized.

Due to the $\mathcal{NP}$-hardness of the general layout decomposition problem, a variety of decomposition approaches have been proposed to achieve high quality and efficiency. These approaches can be roughly categorized into three types: mathematical programming, graph-theoretical approaches and heuristic approaches. The mathematical programming approach formulates the problem into integer linear programming (ILP) [120, 51, 133, 131, 128, 130], and its relaxations such as semi-definite programming (SDP) [131], linear programming (LP) [70] and discrete relaxation method [67]. Besides mathematical programming, graph-theoretical approaches resolve the problem with graph theories, e.g., the maximal independent set (MIS) [32], the shortest-path [23, 105], and the fixed-parameter tractable (FPT) [58] algorithms. Some heuristic approaches are also proposed in [57, 32, 131, 49], which are generally efficient but may have low quality. A recent work formulated MPLD into an exact cover problem and achieved high quality and efficiency with algorithm $X$ [49]. Another extremely fast solution is based on graph matching [57], in which a coloring solution library for small graphs is constructed, and then graphs are colored efficiently by graph matching.

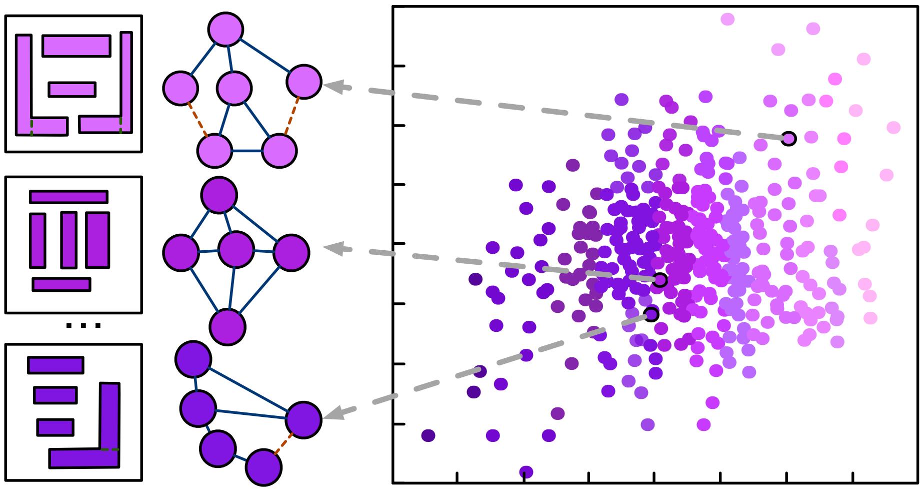


Figure 3.11: An example of graph embeddings of layout graphs, where the graphs are transformed into vector space.

Although many decomposition algorithms have been developed, there is no conclusion that one decomposer is always better than another. ILP-based method ensures the optimality but suffers from runtime overhead for large layouts. Exact-cover (EC) based method demonstrates high efficiency for large layouts at a cost of marginal degradation on the solution quality. The graph matching based method shows good performance in both efficiency and quality for small graphs. But the library size of this method cannot be too large and only non-stitch graphs are supported, which is not applicable to large layouts or layouts with stitches. This observation motivates that it is worth exploring how to adaptively select the most suitable MPLD strategies for a given layout, which is non-trivial and still an open problem so far.

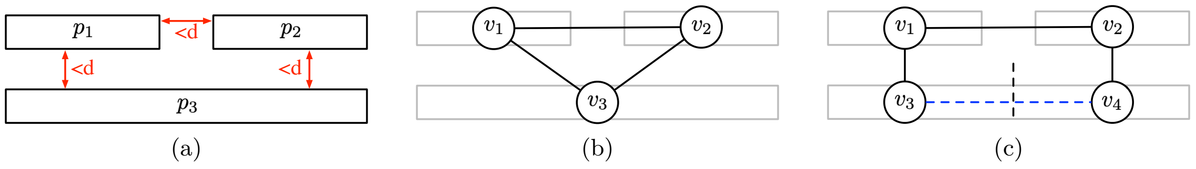


Figure 3.12: An example of the routed layout and its graph representations. (a) The input routed layout; (b) The homogeneous graph representation, where the black line represents the conflict relation; (c) The heterogeneous graph representation considering stitches, where the stitch candidate is marked by the black dotted line, and the stitch edge is highlighted in blue. Here, the relationship between $p$ and $v$ is: $p_1 = \{v_1\}, p_2 = \{v_2\}, p_3 = \{v_3, v_4\}$.

With successful deep learning applications in various fields by learning from historical data, we can naturally cast the problem into a classification task and leverage learning-based approaches. We need to investigate as much information of the graphs as possible and let our framework learn to adaptively utilize proper decomposition algorithms. However, graphs usually vary in terms of scale, making them hard to digest for learning models. Therefore, we need to obtain graph embedding under unified shape to represent the graph as shown in Figure 3.11. Specifically, we use some techniques to generate the graph embedding such that the graph is transformed into a vector space in a lower but unified dimension with maximal representation capability and the powerful graph embedding helps us to adaptively select the best decomposer, where the best refers to the best solution quality at the lowest runtime.

Among different graph embedding methods, graph neural networks (GNN) are widely used for irregular graph representations. In this paper, we develop several GNN variations to obtain graph embeddings for different usages. First, we propose a non-stitch layout decomposer that purely depends on the graph embedding obtained by a specifically-designed GNN. Second, The graph embeddings are used as representations to select ILP-based decomposer (optimal but slow), EC-based decomposer (efficient but may not be optimal), or GNN-based decomposer (efficient and nearly optimal but does not support stitch). Besides decomposer selection, the graph embedding helps us to avoid isomorphic graphs during library construction. After that, it is used for matching graphs efficiently in the library and predict whether the stitch edges in the layout graph are needed or not.

The main contributions are summarized as follows:

- We point the redundancy of stitch candidates in the layout graph, and develop a stitch redundancy prediction method based on graph embeddings.
- We design a non-stitch layout decomposer that purely depends on message passing GNN.
- We design a graph library construction algorithm based on graph embeddings for small graphs excluding isomorphic ones.
- We propose an adaptive workflow for efficient decomposer selection and graph matching using graph embeddings.
- We conduct experiments on widely used benchmarks and experimental results demonstrate that our framework can reduce the runtime by 97.5% while still preserving the optimality compared with optimal but slow ILP-based decomposer.

The rest of this section is organized as follows. Section 3.2.1 lists basic terminologies related to this work and covers. Section 3.2.2 introduces existing state-of-the-art decomposers and proposes a pure GNN-based decomposer, which is specifically designed for the non-stitch layout graphs. Section 3.2.3 shows details of the GNN-based framework, including graph library construction and GNN model construction. Section 3.2.4 covers experimental results.

### 3.2.1 Preliminaries

**Conflict and stitch** A *conflict* happens when two features whose relative distance is less than $d$ are assigned different masks. Sometimes, the conflict can be resolved by dividing the feature using two masks, i.e., assigning two different masks. Such a division by different masks is called a *stitch*. The polygonal feature $p$ is split by stitch(es) into sub-features, i.e., $p = \{..., r_i, ...\}$. To find effective stitches, many works [131, 17] generate a series of *stitch candidates* in the features before decomposition. These stitch candidates indicate possible locations of stitches to prevent the occurrence of conflicts. Previous works have shown that current stitch candidates are able to cover all possible stitches [131, 51].

**Graph format of MPLD** The problem formulation of MPLD can be found in Section 3.1.2. Moreover, MPLD problem can be modeled as a variation of a pure graph-based problem, since the input layout can be translated into an undirected graph $G = (V, E)$ without any information loss. When we consider the stitch candidates, i.e., pre-define possible stitch locations, $G$ is a *heterogeneous* graph where the node $v_i$ corresponds to the subfeature $r_i$, and the the edge set $E$ is composed of two subsets: the conflict edge set $CE$ and the stitch edge set $SE$. Otherwise, $G$ is a simple *homogeneous* graph, where the node corresponds to one polygonal feature, i.e., $v_i \to p_i$, and the edge only represents the conflict relation. One example of the two representations are shown in Figure 3.12. In our paper, we focus on the heterogeneous layout graph that is more complex but practical.

In the heterogeneous layout graph, if one feature is split into multiple sub-features by stitch candidate(s), it will be translated into multiple nodes in the graph. To be more specific, one node is either 1) one polygonal feature if there is no stitch candidates in the feature or 2) one sub-feature split by the stitch candiate(s) in the polygonal feature. The edge set $E = \{CE, SE\}$ models the relations between nodes. Two nodes are connected by the conflict edge $e \in CE$ if their relative distance is less than $d$ and they do not belong to the same feature; Two nodes are connected by the stitch edge $e \in SE$ if they belong to the same feature and split by one stitch candidate.

The objective of MPLD problem is to assign masks to (sub)features so that the weighted sum of conflicts and stitches are minimized. From the perspective of a graph coloring problem, the objective is to assign colors to each node so that the weighed sum of conflict cost and stitch cost is minimized. Let $f : v \to \{1, ..., k\}$ be the coloring(decomposition) function and $f(v)$ be the color assigned to $v$ by $f$. Given two features, $p_m, p_n$, the conflict cost adds to one if at least one corresponding node pair connected by the conflict edge is assigned different colors, i.e., $\exists r_i \in p_m, r_j \in p_n : \{v_i, v_j \in CE, f(v_i) = f(v_j)\}$. The stitch cost adds to one if the two nodes connected by one stitch edge is assigned the same color. Formally, the objective can be formulated as shown in Equation (3.13),

$$
\begin{aligned}
\min_{f} \quad & \sum_{p_m, p_n \in P; m \neq n} C_{mn} + \alpha \sum_{\{v_i, v_j\} \in SE} s_{ij}, && \text{(3.13a)} \\
\text{s.t.} \quad & C_{mn} = \min\{ \sum_{\substack{r_i \in p_m, \\ r_j \in p_n, \\ \{v_i, v_j\} \in \text{CE}}} c_{ij}, 1\} && \text{(3.13b)} \\
& s_{ij} = \begin{cases} 1, & \text{if } f(v_i) \neq f(v_j); \\ 0, & \text{otherwise.} \end{cases} && \text{(3.13c)} \\
& c_{ij} = \begin{cases} 1, & \text{if } f(v_i) = f(v_j); \\ 0, & \text{otherwise.} \end{cases} && \text{(3.13d)}
\end{aligned}
$$

where $\alpha$ is a parameter indicating the relative importance between the conflict cost $C$ and the stitch cost $s$, which is usually set as 0.1.

**Graph Isomorphism and Graph Matching** Intuitively, graph isomorphism problem is to decide whether two ordered graphs are identical after they are un-ordered. Graph matching is not only to decide whether they are identical or not, but also to give an order map of nodes if they are identical.

The formal definition of graph isomorphism and graph matching is stated as follows [12]: Given two graphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ with $|V_1| = |V_2|$, where $V_1, V_2$ and $E_1, E_2$ are corresponding node sets and edge sets, respectively. The object of graph matching is to find a node-to-node mapping $f : V_1 \to V_2$ such that $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$. This is called an isomorphism if such a mapping $f$ exists, and $G_1$ is said to be isomorphic to $G_2$.

In the graph library construction, graph isomorphism is one of the most critical factors because $n! - 1$ isomorphic graphs of any valid graph will be re-collected in the library if no isomorphism-free techniques are used, where $n$ is the number of nodes. Also, graph matching is inevitable when extracting the corresponding node coloring results stored in the graph library for the matched graph.

**Graph Neural Networks (GNN)** With the development and further study of Neural Network, GNN, as a branch of Neural Network, has shown promising results in many domains such as the graph embedding.

GNN takes the graph as input and returns the node embeddings or graph embedding. Nowadays, most widely-used GNNs adopt an iterative manner that is composed of two repeated steps: aggregation and combination, which exploit the neighborhood information and ego-information respectively. Specifically, for each node $u$ in graph $G$, the aggregation step aggregates neighbor $v$'s representations $\boldsymbol{h}_v$ and obtain an intermediate representation $\hat{\boldsymbol{h}}_u$ such that the final graph embedding is able to contain graph structure information. During the combination, GNN combine the aggregated representation $\hat{\boldsymbol{h}}_u$

with the ego-feature, and the result feature becomes the input of next layer. GNN can be also explained in a message-passing way where the intermediate representations can be viewed as messages. The aggregation is the actual message-passing phase and each node passes its message to its neighbors along the edge. The combination is served as the integration phase, in which each node integrates received the message and reduces it into its new message. Each message-pass and integration phase formulate one GNN layer. A general GNN layer can be described as follows:

$$\boldsymbol{h_v^{(i)}} = \mathrm{COM}^{(i)}(\boldsymbol{h_v^{(i-1)}}, \mathrm{AGG}^{(i)}(\{\boldsymbol{h_u^{(i-1)}} : u \in \mathcal{N}(v)\})), \tag{3.14}$$

where $\boldsymbol{h}_v^i$ is the feature of node $v$ after $i_{th}$ layer, COM is the combination function, and The representation after the final layer is called the node embedding of each node and the graph embedding by GNN is usually obtained by some node invariant operations on node embeddings such as summation or mean.

**Problem Formulation** Given a set of layout graphs and two state-of-the-art decomposers, ILP-based decomposer and EC-based decomposer, our objective is to train several GNN variations to obtain the graph embeddings such that 1) the embedding can help to directly color the homogeneous layout graph, i.e., layout that does not contain any stitch candidate; 2) the embedding can be used to build a graph library for small graphs, recording the coloring solutions; 3) any new graph can find the best decomposer using its embedding; 4) any new small graph can find the coloring solution directly through graph matching with graphs in the library.

### 3.2.2 Layout Decomposition Algorithms

#### State-of-the-art Decomposers

Among the past few years, lots of decomposers are developed to solve the MPLD problem. We compare all state-of-the-art decomposers in terms of four perspectives: 1) Result quality; 2) Efficiency; 3) Flexibility on multi-thread, GPU-acceleration, larger layout, and more masks; 4) Whether the method supports the stitch insertion. A general comparison is shown in Table 3.10. In the following paragraphs, we simply introduce these existing decomposers and discuss the performance on the four listed perspectives.

**Integer Linear Programming(ILP)** Given the objective function in the form of graph as shown in Equation (3.13), the problem can be naturally solved by ILP [51, 131], where the node color $f(v_i)$ can be represented by 1-bit 0-1 variable(s). The ILP model for TPLD is decribed in Formula equation 3.12.

ILP-based method gives the optimal solution and supports the stich scheme with only a trivial modification of the modeling process. However, the poor efficiency impedes its employment on large layout, which becomes more and more important with the development of semiconductor industry.

**Semi-definite Programming (SDP)** Solving Equation (3.12) using ILP is $\mathcal{NP}$-hard, as an alternative solver, SDP can approximately solve Equation (3.12) in linear time. The basic idea is to program the colors by vectors so that the inner product between two vectors give different values based on whether the two vectors are the same or not. For example, in the TPLD problem [131, 79, 129], the three colors are assigned to three 2-dimension vectors, $(1, 0), (-1/2, \sqrt{3}/2)$, and $(-1/2, -\sqrt{3}/2)$ respectively. Then, given any two vectors $\boldsymbol{v}_i, \boldsymbol{v}_j$, which represents the colors of node $i$ and node $j$, we have the following properties:

$$\boldsymbol{v}_i \cdot \boldsymbol{v}_j = \begin{cases} 1, & f(v_i) = f(v_j) \\ -1/2, & f(v_i) \neq f(v_j) \end{cases} \tag{3.15}$$

Therefore, the MPLD problem can be solved by semidefinite programming in polynomial time if we relax the discrete values of $\boldsymbol{v}$ to a continuous one. Given the solutions of SDP, a fast heuristic mapping process is used to map the solutions to coloring results.

SDP based method makes a good balance on the efficiency and performance, and can be applied to the stitch case by simply adjusting the cost function. Nevertheless, the vector programming process for the node color in [131] is specifically designed for the TPLD problem, when extended to the quadruple patterning problem or even more masks, the dimension of the vector will also increase, which harms the efficiency.

**Exact cover (EC) based method** EC-based method [17] transforms the MPLD problem to an exact cover problem, and solves it by a customized and augmented combination of dancing links data structure and Algorithm X$^*$ (DLX). Here, the routed layout is translated into a 0-1 matrix. In the matrix, each row index represents one possible coloring solution of a single feature $p$. If there is no stitch candidate in $p$, there will be $k$ rows, representing $k$ different color assignments of $p$. Otherwise, there will be more than $k$ rows to represent different color combinations of subfeatures split by stitch candidates. The column index models the conflict relation to make sure that two nodes connected by the conflict edge are not assigned the same color. Finally, the EC-based method returns a set of rows, which can be translated back to the decomposition result of MPLD problem.

The EC-based method demonstrates excellent efficiency, also, it is applicable for the stitch scheme and multi-thread. Moreover, it show an relatively fast execution time for very large case. However, for some cases, it cannot be optimal, and the result quality may vary largely depending on the structure of the layout graph.

**Graph matching based method** The basic idea of graph matching is to build a graph library which contains graphs and corresponding solutions. Each time when we try to decompose the layout, the system will match the target layout graph to graphs in the library. If a match is found, the corresponding decomposition solution is returned. Generally, since the library can be constructed offline, i.e., before any decomposition, the decomposition runtime only depends on the efficiency of the matching algorithm and the size of the library. However, the graph library size explodes when increasing the graph

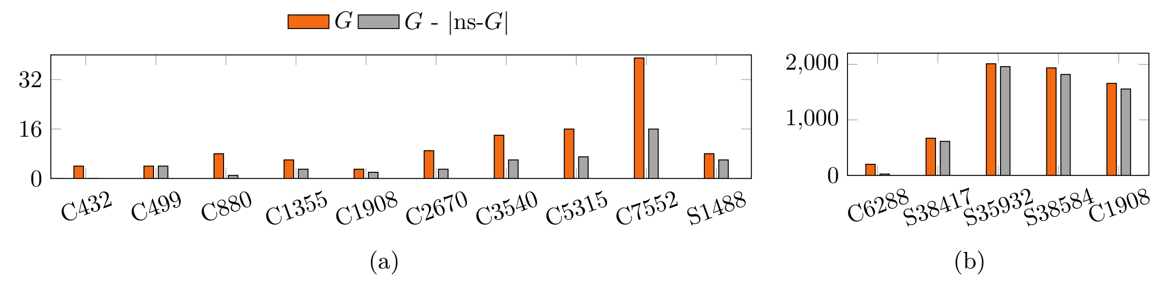


Figure 3.13: The histogram of the number of graphs (orange) and graphs that need not stitches (gray) in (a) small layouts and (b) large layouts.

size or considering the stitch. In [57], the graph library does not support stitch scheme and only contains graphs with size less than six. Therefore, the flexibility is poor compared with other decomposers.

### Non-stitch Layout Decomposer by GNNs

Given the overwhelming success of GNNs, we may attempt to solve the MPLD problem by GNNs directly. However, as a heterogeneous variation of the pure coloring problem, the MPLD problem which contain both stitch edge and conflict edge is more difficult. On the other hand, there exists lots of redundant stitch edges which are not working in the final decomposition results, i.e., no stitches are introduced. The histogram shown in Figure 3.13 empirically states the phenomenon: Over more than 80% layout graphs do not have stitches in the final results while most of them contain stitch edges. Although it is not easy to decompose the layout graph by GNNs directly, the stitch redundancy provides a vision of GNNs applying to the no-stitch layout decomposition. In the following section, we will first introduce how graph embedding is used for the stitch redundancy prediction, and then propose a pure GNN-based method for the decomposition of non-stitch graphs.

**Stitch Redundancy Prediction** Despite the fact that the state-of-the-art stitch candidate generation algorithm is able to enumerate all stitches, there are a huge number of stitch candidates that do not influence the final coloring results, i.e., the two nodes split by the stitch candidates are assigned the same color in any optimal solutions. One example of such a redundancy is given in Figure 3.14, where each stitch candidate splits the corresponding feature into two subfeatures and generate two nodes connected by the stitch edge. Since both nodes are assigned the same color in the optimal solution, these redundant stitch candidates have no influence to the coloring quality.

However, useless candidates will increase the problem complexity largely and result in significant drops in efficiency performance. The layout statistics in Figure 3.13 demonstrates that there exists a large portion of layout graphs that totally need not stitches. To avoid the waste of computation resources and further improve the efficiency of our

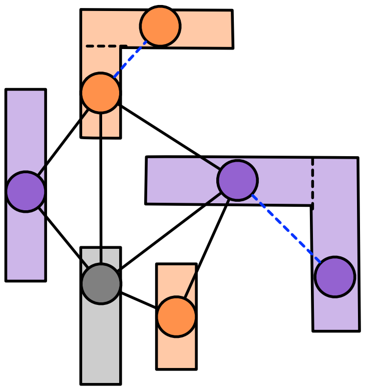


Figure 3.14: The example of redundant stitch candidates. In this layout graph, both stitch candidates (highlighted in blue) finally do not generate any stitch.

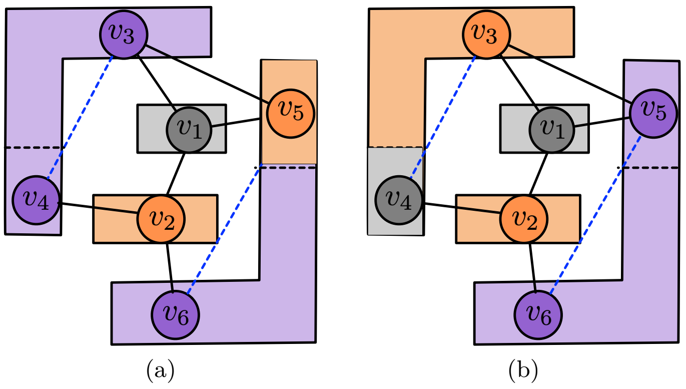


Figure 3.15: The example of alterative solutions for stitches. The activated stitch is $\{v_5, v_6\}$ in (a) and $\{v_3, v_4\}$ in (b) respectively.

decomposition framework, we propose a graph embedding based method to remove these redundant and useless stitch candidates. Since the graph embedding is already obtained for the graph matching and the merge operation can be finished in a constant time, the additional time cost can be ignored in light of the huge benefit from removing the redundant stitches.

Although the successful prediction can bring about a free efficiency improvement, it is not easy to accurately predict which stitch candidate can be removed. The optimal solution is usually not unique: one stitch candidate can be redundant in one optimal solution while not in another one. The stitch candidate $\{v_3, v_4\}$ shown in Figure 3.15 is a representative example caused by the partial symmetry between $d - e$ and $f - g$. In Figure 3.15(a), edge $\{v_3, v_4\}$ is redundant since node $v_3$ and $v_4$ are assigned the same color. On the contrary, the two nodes have different color in another optimal solution shown in Figure 3.15(b), indicating that the stitch edge $\{v_3, v_4\}$ cannot be removed.

Considering the non-uniqueness of stitches as illustrated in Figure 3.15, we regard

**Algorithm 3** Pure GNN-based Layout Decomposer

```
Input: $\mathcal{A} \leftarrow$ GNN trained for predict whether stitch candidates are needed;
Input: $\mathcal{A}' \leftarrow$ GNN trained for graph coloring;
Input: $g \leftarrow$ Target graph;
Input: $iter \leftarrow$ Number of repetitive executions;
Ourput: $\boldsymbol{x} \rightarrow$ The coloring results for each node in $g$;
 1: $\boldsymbol{h}_s \leftarrow \mathcal{A}(g)$;
 2: confidence $\leftarrow MLP(\boldsymbol{h}_s)$;
 3: if confidence $\leq b$ then
 4:     Decompose $g$ by other decomposer;
 5:     return the decomposition solution;
 6: else
 7:     $g \leftarrow$ Remove all stitch edges in $g$ and merge related nodes;
 8: end if
 9: for $i \in \{1, ..., iter\}$ do
10:     $\boldsymbol{x} \leftarrow$ Randomly initialized probability distribution of colors for each node;
11:     $\boldsymbol{h} \leftarrow \mathcal{A}(g, \boldsymbol{x})$;
12: end for
13: return the best solution in $\{\boldsymbol{h_1}, ..., \boldsymbol{h_{iter}}\}$;
```

the problem as a graph classification problem rather than a edge classification problem. That is, the algorithm predicts *whether the graph contains at least one redundant stitch candidate* instead of *whether the stitch edges in the graph are redundant.* Therefore, the redundancy prediction can be implemented as a 2-class classifier on graph-level and simply modeled by a multi-layer perceptron (MLP) that uses the graph embedding as input. A detailed illustration can be found on Algorithm 3. After obtaining corresponding graph embedding $\boldsymbol{h}_s$ (line 1), $\boldsymbol{h}_s$ is fed into the implemented MLP and predicts a confidence value (line 2). After prediction, if the confidence of the graph is larger than a specific bar, say $b$ (lines 6–7), the graph will merge all stitch candidates in the graph, which results in a non-stitch graph.

**Non-stitch Layout Decomposer by GNNs** We use the message passing GNN as a backbone, and give a prior that the node embedding represents the probability (belief) of color assignments. The detail is described as follows: Given a non-stitch graph $G = \{V, E\}$, where $E = \{CE\}$, we first randomly assign each node $v \in V$ a discriminative attribute $\boldsymbol{x}_v \in \mathbb{R}^k$ that represents the probability distribution of $k$ masks.

In the aggregation step, we simply sum up the features from all neighbors. Formally, let $\boldsymbol{m}_v^{(i)} \in \mathbb{R}^k$ be the result returned by AGG$^{(i)}$ for the node $v$ in the $i$-th layer, the aggregation layer can be represented by: $\boldsymbol{m}_v^{(i)} = \sum_{u \in \mathcal{N}'(v)} \boldsymbol{h}_u^{(i-1)}$, where $\mathcal{N}(v)$ is defined as the subset of $\mathcal{N}(v)$.

In the combination function, we define the COM$^{(i)}$ as a simple trainable weighted summation between ego-featre and features from neighbors:

$$\boldsymbol{h}_v^{(i)} = \boldsymbol{h}_v^{(i-1)} \lambda_C^{(i)} + \boldsymbol{m}_v^{(i)} \lambda_A^{(i)}. \tag{3.16}$$

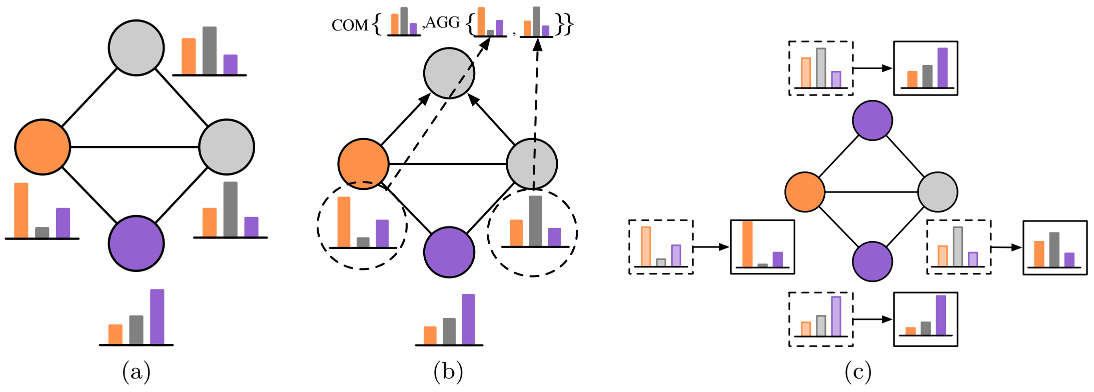


Figure 3.16: A toy example on how AC-GNN gives the coloring results directly. (a) The randomly initialized color distribution; (b) The message passing procedure finished by trainable AC-GNN; (c) Final results (color distribution) of AC-GNN.

Table 3.10: Comparsion among different decomposers

| Methods | Quality | Efficiency | Flexibility | Stitch |
|---|---|---|---|---|
| ILP | Optimal | Poor | Medium | Yes |
| SDP | Near optimal | Medium | Medium | Yes |
| EC | Near optimal | Fast | Strong | Yes |
| Graph Matching | Optimal | Fast | Poor | No |
| Our GNN decomposer | Near optimal | Very fast | Strong | No |

Here, both $\lambda_C^{(i)}$ and $\lambda_A^{(i)}$ are trainable variables. After obtaining the node embedding $\boldsymbol{h}_i$, the color of each node is assigned based on the $\boldsymbol{h}_i$ that represents the color belief. A figure illustration of the whole process is shown in Figure 3.16. In our implementation, we iteratively execute the GNN multiple times (lines 9–13 in Algorithm 3) by setting different initializations. Finally, we select the best solution among all iterations.

### 3.2.3 Adaptive Decomposition Framework

In this section, we first briefly present the workflow of our proposed framework. Then we describe the GNN used for graph embedding, and how the graph embedding is used for graph library construction, graph matching, and decomposer selection.

#### Overview

Combining with the pure GNN-based decomposer described in Section 3.2.2, we further propose an decomposition framework that selects the decomposition methods adaptively. Our framework can be divided into two modules by whether the operation is needed in the decomposition (online) or not (offline). The offline module is prepared before any decomposition, including graph library construction and GNN model training.

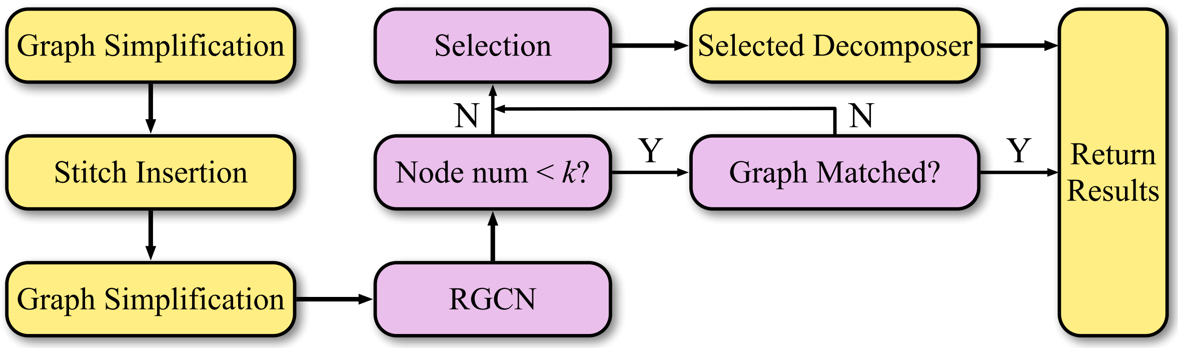


Figure 3.17: The workflow of our framework. Purple blocks are executed in our framework while the yellow blocks are directly executed in OpenMPL [64].

Firstly, we train the RGCN model, then we use graph embeddings obtained by the trained RGCN model to build the isomorphism-free graph library. When the above offline steps are finished, we can execute layout decomposition following the workflow shown in Figure 3.17. The input is transformed into a graph first and is simplified by several simplification techniques such as Independent Component Computation (ICC) [131], Hide Small Degrees [57, 131], Biconnected Component Analysis [51, 133]. Next, stitch candidates are inserted by pattern projection [131]. After stitch insertion, the simplified homogeneous graphs are transformed into heterogeneous graphs which contain both conflict and stitch edges and then these simplified heterogeneous graphs are fed into the RGCN model to obtain the graph embeddings. For a graph whose graph size is under the size constraint $max\_size$, the corresponding graph embedding is first used to determine whether there is an isomorphism between the target graph and graphs in the library. If the isomorphic graph is found in the library, the corresponding node embeddings of two graphs are used to get the node-to-node mapping and directly return the final coloring result by the mapping in the library. If no isomorphic graph is found or the graph size is larger than $max\_size$, the graph embedding is followed by a fully connected layer for decomposer selection and the graph is then decomposed by the selected decomposer. Especially, another graph embedding is obtained and used for the stitch redundancy prediction, if it is predicted as redundant, all stitches are merged and the simplified graph is decomposed by our proposed GNN decomposer. After all graphs are decomposed, a color recovery process is executed to get the final layout decomposition results.

### Graph Embedding Neural Network

Graph embedding neural network is one of the most critical parts in our framework since the graph embedding obtained by the neural network is the basis for every module, e.g. graph matching, algorithm selection, and stitch redundancy prediction. Considering that the simplified graph is heterogeneous, which contains both conflict and stitch edges, we applied Relational Graph Convolutional Networks (RGCN) similar to [96] to obtain the graph embedding. The process for graph embedding is shown in Figure 3.18. The original layout is transformed into multiple heterogeneous graphs by graph simplification and stitch insertion. Those simplified graphs are the input of the model and the model is composed of two neural network layers. For each node $v_i$ in a graph $G = \{V, E\}, E = \{E_c, E_s\}$, the node representation $\boldsymbol{h}_i^{(l+1)} \in \mathbb{R}^{D^{(l+1)}}$ at the $(l+1)_{th}$ layer of the neural network can be calculated by the following formula:

$$\boldsymbol{h}_i^{(l+1)} = ReLU\left(\sum_{e \in E} \sum_{j \in N_i^e} \boldsymbol{W}_e^{(l)} \boldsymbol{h}_j^{(l)} + \boldsymbol{h}_i^{(l)}\right), \tag{3.17}$$

where $D^{(l)}$ is the dimension of node representation at the $l_{th}$ layer, $\boldsymbol{W}_e^{(l)} \in \mathbb{R}^{D^{(l+1)} \times D^{(l)}}$ is a learnable weight matrix of edge type $e \in E$ and $N_i^e$ denotes the set of neighbor nodes of node $v_i$ connected by $e$. Intuitively, RGCN specified in Equation (3.17) works like the classical GCN, as both neural network layers contain two phases, aggregation and encoding. The difference is that edges in GCN share the same learnable weight in each layer on the encoding phase while only edges in the same edge type share the weight matrix for RGCN, which means that the message integration for different kinds of edges is independent. One central issue resulted from the different weight matrixes strategy is that the number of parameters rapidly grows with the number of the edge categories in the graph. Also, this kind of strategy can easily lead to overfitting due to a large number of parameters. The issue is solved by regularization of weight and we adopt a basis decomposition [96], in which each weight matrix $\boldsymbol{W}_e^{(l)}$ is a linear combination of basis transformations $\boldsymbol{V}^{(l)}$ and defined by:

$$\boldsymbol{W}_e^{(l)} = \sum_{b=1}^{B} \delta_{rb}^{(l)} \boldsymbol{V}_b^{(l)}, \tag{3.18}$$

where $\boldsymbol{V}_b^{(l)} \in \mathbb{R}^{D^{(l+1)} \times D^{(l)}}$ is one of the multiple basis transformations and $\delta_{rb}^{(l)}$ is the learnable coefficient.

In our implementation, we also adopt widely-used ReLU as the activation function, the input feature of node $v_i$ is defined as:

$$\boldsymbol{h}_i^{(0)} = \sum_{j \in N_i} I_{\{e_{i,j} \in E_c\}} + \alpha I_{\{e_{i,j} \in E_s\}}, \tag{3.19}$$

where $I_{\{\cdot\}}$ is an indicator function and $\alpha = -0.1$ is a user-defined parameter following the general stitch cost. After obtaining the node embeddings by RGCN model, for the algorithm selection, we calculate the graph embedding by the summation of the node embeddings considering that graph size influences the decomposition quality of EC-based decomposer. Formally, we have $\boldsymbol{h} = \sum_{i \in V} \boldsymbol{h}_i^{(out)}$, where $\boldsymbol{h}_i^{(out)}$ is the node embedding of node $v_i$. As for the stitch redundancy prediction, we use a max-pooling because it is some subgraph structures that determine whether there exists redundant stitch edges or not.

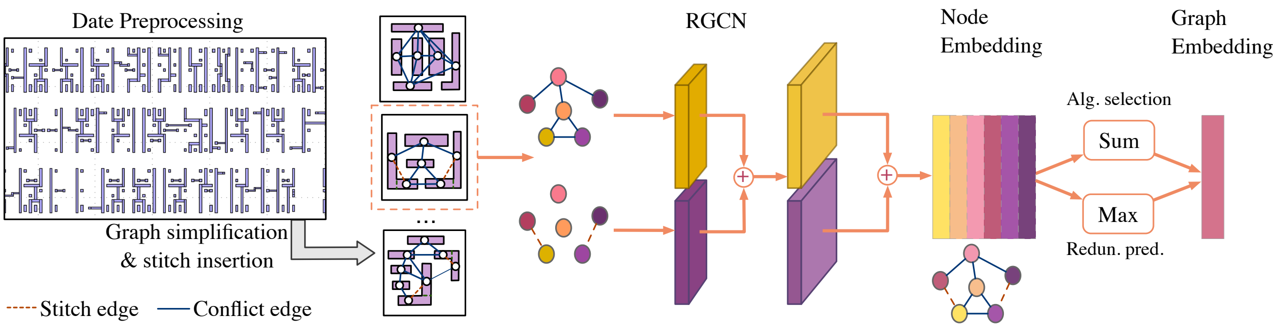


Figure 3.18: Overview of the process for graph embedding

### Graph Library Construction

Generally speaking, it is possible to enumerate all the valid graphs under the size constraint such that we can build up a graph library to accelerate decomposition by simply matching the graph with graphs in the library and collect the coloring information stored in the library.

Previous work [57] constructed a graph library that contains all homogeneous graphs (23 in total) with node number less than seven following the algorithm described in [101, 81]. However, the graph in the previous library does not contain stitch edge, which means that one heuristic stitch insertion and coloring method should be used if the no-stitch graph is not colorable. Therefore, we propose an isomorphism-free heterogeneous graph library construction algorithm that contains all the possible graphs with both stitch edges and conflict edges.

We first define the target heterogeneous graph as $G = \{V, E\}$, where $V, E$ are the node set and the edge set respectively. Furthermore, we define a corresponding parent graph $G^p = \{V^p, E^p\}$, which is the no-stitch form of $G$ by merging nodes connected by stitch edges. Different from the general 2-connected graph described in [101], the graph transformed by circuit layout has some specific rules, especially after stitch insertion. The rules are stated as follows:

- $G^p$ is a 3-connected graph instead of 2-connected.
- The degree of each node in $G$ is at least two.
- One node pair $\{u, v\}$ cannot be connected if $u, v$ are in the stitch relation. Stitch relation of two nodes means that there is a path connecting them and only go through stitch edges with length larger than one.

**Algorithm 4** Graph Library Construction

```
Input: $max\_size$ → Maximal graph size.
Ourput: $L$ → The isomorphism-free library of valid graphs;
 1: $L \leftarrow \{\}$;
 2: $S_p \leftarrow$ Generate graphs following method in [101];
 3: $S_p \leftarrow$ Remove invalid Graphs in $S_p$;
 4: $S \leftarrow$ Enumerate graphs containing stitches from graphs in $S_p$;
 5: for $G \in S$ do
 6:     if $G$ satisfies layout graph rules then
 7:         $\boldsymbol{h} \leftarrow$ normalize(RGCN($G$));
 8:         $\boldsymbol{\mathcal{L}}_h \leftarrow$ Extract graph embeddings stored in the library;
 9:         if max($\boldsymbol{\mathcal{L}}_h \times \boldsymbol{h}$) $< 1$ then
10:             Decompose $G$ with ILP-based decomposer;
11:             Insert $G$ into $L$;
12:         end if
13:     end if
14: end for
```

- The neighbors connected by conflict edge cannot be totally the same for two nodes in stitch relation.

The pseudocode of our library construction algorithm is illustrated in Algorithm 4. Firstly, we enumerate $G^p$ by the method in [101] (line 2), which generates isomorphism-free 2-connected graph set and removes all graphs which are not 3-connected (line 3). Then for each $G^p$, we enumerate valid $G$ which satisfies the size constraint and all the rules above by splitting nodes in $G^p$ and insert stitch edges (lines 4–6). Note that there may be multiple isomorphic graphs in the enumeration of $G$ such that we use graph embedding to avoid isomorphism. Specifically, every time when the enumerated $G$ is going to put into the library, $G$ will be fed into the RGCN model (line 7) and obtain a corresponding normalized graph embedding $\boldsymbol{h} \in \mathbb{R}^D$, where $D$ is the dimension of the graph embedding. Then the normalized graph embeddings $\boldsymbol{\mathcal{L}}_h \in \mathbb{R}^{k \times D}$ stored in the library are extracted (line 8) and a vector-matrix multiplication is performed i.e., $\boldsymbol{m} \in \mathbb{R}^k = \boldsymbol{\mathcal{L}}_h \times \boldsymbol{h}$, where $k$ is the number of graphs stored in the library temporarily. Then whether there is an isomorphic graph in the library or not is determined by checking the maximum element in $\boldsymbol{m}$ (line 9) because two unit vectors are equal if and only if their product is 1. The idea is based on the fact that a GCN-based model is insensitive to the node order, which means that the graph embeddings of all isomorphic graphs by a GCN-based model are totally the same. After isomorphism determination, $G$ won't be inserted into the library if there is an isomorphic graph. Otherwise, $G$ will be decomposed by ILP-based decomposer for optimal solution (line 10), then graph $G$ with its optimal coloring result, corresponding graph embedding and node embeddings will be stored in the library (line 11).

### Graph Matching and Decomposer Selection

**Graph Matching** When the graph embedding is obtained by our model and the graph size is under the limitation, we directly match the graph with graphs in the library. We use the obtained graph embedding to find isomorphic graphs in the library, then we use the corresponding node embeddings to find the node-to-node mapping and return the solution directly.

To illustrate the process clearly, we provide a simple example and explain the details step by step. The graph library $\boldsymbol{\mathcal{L}}$ in this example is composed of three graphs, in which each graph has four nodes and the dimension of graph embedding is two. The library stores all information of graphs needed by our framework including its node embeddings $\boldsymbol{\mathcal{L}}_u \in \mathbb{R}^{3\times4\times2}$, graph embeddings $\boldsymbol{\mathcal{L}}_h \in \mathbb{R}^{3\times2}$ and optimal solutions $\boldsymbol{\mathcal{L}}_s \in \mathbb{R}^{4\times3}$:

$$\boldsymbol{\mathcal{L}}_u = \begin{bmatrix}0.4 & 0.4\\ 0.3 & -1.0\\ 0.1 & 0.6\\ -0.2 & 0.8\end{bmatrix} \qquad \boldsymbol{\mathcal{L}}_h = \begin{bmatrix}0.6 & 0.8\\ 0.6 & -0.8\\ 1 & 0\end{bmatrix} \qquad \boldsymbol{\mathcal{L}}_s = \begin{bmatrix}1 & 1 & 0\\ 2 & 1 & 1\\ 0 & 2 & 0\\ 1 & 0 & 2\end{bmatrix}$$

Different colors represent different graphs in the library. Take a target graph $G$ with four nodes for example, we use RGCN model to obtain the corresponding node embedding $\boldsymbol{u} \in \mathbb{R}^{4\times2}$ and graph embedding $\boldsymbol{h} \in \mathbb{R}^2$, where $\boldsymbol{h} = \sum_i \boldsymbol{u}_i$:

$$\boldsymbol{u} = \begin{bmatrix}0.3 & -1.0\\ -0.2 & 0.8\\ 0.4 & 0.4\\ 0.1 & 0.6\end{bmatrix}, \boldsymbol{h} = \begin{bmatrix}0.6\\ 0.8\end{bmatrix}. \tag{3.20}$$

We first multiply the graph embedding $\boldsymbol{h}$ with graph embeddings $\boldsymbol{\mathcal{L}}_h$ in the library, i.e., $\boldsymbol{m} \in \mathbb{R}^3 = \boldsymbol{\mathcal{L}}_h \times \boldsymbol{h}$:

$$\boldsymbol{m} = \underset{\boldsymbol{\mathcal{L}}_h}{\begin{bmatrix}0.6 & 0.8\\ 0.6 & -0.8\\ 1 & 0\end{bmatrix}} \times \underset{\boldsymbol{h}}{\begin{bmatrix}0.6\\ 0.8\end{bmatrix}} = \begin{bmatrix}1\\ -0.28\\ 0.6\end{bmatrix}$$

Then the matched graph index $i$ in the library is defined by:

$$i = \begin{cases}\arg\max(\boldsymbol{m}), & \text{if} \max(\boldsymbol{m}) = 1;\\ -1, & \text{otherwise},\end{cases} \tag{3.21}$$

where $-1$ means there is no isomorphic graph matched in the library such that the graph matching process is terminated and redirected to decomposer selection, otherwise, the $i_{th}$ node embedding in the library is extracted and compared with the target graph's node embedding to get the final node-to-node mapping. This comparison method is also based on the node order insensitivity of the GCN-based model, if the input feature doesn't contain any information related to the node order such as a one-hot vector of the node order, the final graph embedding is then order-invariant because the message passing process is only related to the neighbors instead of the node order. In this example, $\boldsymbol{m}[0] = 1$ such that $i = 0$, which means that the first node embedding $\boldsymbol{\mathcal{L}}_u[0]$ is used to compare with $\boldsymbol{u}$.

The node-to-node mapping $f$ is executed by comparing two node embeddings and formulated by:

$$f(j) = k, \text{ if } \boldsymbol{u}[j] = \boldsymbol{\mathcal{L}}_u[i][k] \text{ for } j, k \text{ in } \{0, \ldots, |G| - 1\}, \tag{3.22}$$

where $|G|$ means the number of nodes in the graph. In this example, $|G|$ is exactly 4 and $f$ is then defined by: $f(\{0, 1, 2, 3\}) = \{1, 3, 0, 2\}$.

After $f$ is found, the solution $\boldsymbol{s}$ can be matched quickly by:

$$\boldsymbol{s}[j] = \boldsymbol{\mathcal{L}}_s[f(j)][i], \text{ for } j \text{ in } \{0, \ldots, |G| - 1\}, \tag{3.23}$$

so the final solution of $G$ in this example is mapped as $[2, 1, 1, 0]$.

**MPL Decomposer selection** When the size is larger than the size limitation or no mapping is found in the library, the graph embedding is used to select the decomposer. Therefore, the decomposer selector can be regarded as a 2-class classifier and simply modeled by a summation of one trainable weight matrix $\boldsymbol{W}_s \in \mathbb{R}^{2\times D}$ and a bias vector $\boldsymbol{b}_s \in \mathbb{R}^2$ combined with arg max function, which can be formulated as:

$$y = \arg\max(\boldsymbol{W}_s\boldsymbol{h} + \boldsymbol{b}_s), \tag{3.24}$$

where $\boldsymbol{h} \in \mathbb{R}^D$ is the graph embedding obtained by RGCN model with dimension $D$. The final decomposition result is then generated by the selected decomposer.

### 3.2.4 Experimental Results

The experiments are performed on the scaled-down and modified ISCAS benchmarks, which are widely used in previous works [49, 131, 57]. The framework is mainly implemented in Python with PyTorch [86] and DGL [108] and integrated into the open-source layout decomposition framework OpenMPL [64]. Figure 3.17 specifies the detailed task execution platform of the workflow. It should be noted that our graph embedding as well as the whole framework is very general that they can be naturally extended to other decomposition tasks under different lithography constraints. We follow the same settings in [131, 49, 57] on the minimum color space, where the first ten cases are set to 120 nm and the last five cases are set to 100nm. The cost of stitch is set to 0.1 such that the decomposition cost is calculated by $cn\# + 0.1st\#$, mask number is set to 3 and the graph simplification level in OpenMPL is 3. In the training phase of our model, we concatenate the graph embedding network with the following MPL decomposer selector such that the cross-entropy loss function can be adopted. In the algorithm selection, the label of each simplified graph for training is set as 0 (ILP) if the cost by ILP-based decomposer is smaller than EC-based decomposer and 1 (EC) for other cases. In the stitch redundancy prediction, each layout graph is labeled as positive if there exists at least one stitch in the optimal solution. Generally, we prepare and train three independent GNN models for 1) graph matching and decomposer selection (RGCN); 2) stitch redundancy prediction (RGCN), and 3) non-stitch GNN decomposer (Equation (3.16)). The RGCN model contains two layers whose output dimensions are 32, 64 respectively such that the dimension of graph embedding is 64. The training strategy follows the idea of K-fold cross-validation, specifically, each time two of the 15 layouts in the benchmark are used as the test/validation set separately, and the other 13 layouts are put together to form a training set. Therefore, there are 15 trained models for 15 layouts following the same model configurations. The layout is first preprocessed by graph simplification and stitch insertion such that the dataset is composed of multiple graphs. Considering that our dataset is significantly unbalanced since EC-based decomposer is optimal and also the fastest in most cases, we set the training epoch to 1 and use a weighted random sampling strategy with weight ratio 300:1 to avoid overfitting. In all GNN-related operations such as the non-stitch GNN solver, algorithm selection, and stitch redundancy prediction, the simplified graphs of the target layout are batched together for efficient inference. All the experiments are conducted on an Intel Core 2.9 GHz Linux machine with one NVIDIA TITAN Xp GPU.

#### Effectiveness of model selection

In the first experiment, we compare the effectiveness of our proposed RGCN model with conventional GCN model. The classical GCN model only supports homogeneous graphs while there are two kinds of edges in this task. Therefore, we slightly modify the message passing function by multiplying the edge weight $\alpha_e$ for different edge types:

$$\boldsymbol{u}_i^{(l+1)} = ReLU\left(\sum_{e\in E}\sum_{j\in N_i^e} \alpha_e \boldsymbol{W}^{(l)}\boldsymbol{u}_j^{(l)} + \boldsymbol{u}_i^{(l)}\right), \tag{3.25}$$

where $\alpha_e$ is 1 for conflict edge and -0.1 for stitch edge following the weighted cost setting, the negative sign is due to the fact the stitch edge and conflict edge play different roles in decomposition: nodes connected by a conflict edge are assigned different colors while stitch edge indicates same color. The result is illustrated by the confusion matrix shown in Table 3.11, where each row contains the number of graphs selected to be decomposed by the corresponding decomposer while each column contains the number of graphs labeled by the corresponding decomposer. For example, the element (0,0) in the confusion matrix

Table 3.11: F1 score comparison of (a) proposed RGCN and (b) conventional GCN.

(a)

| | | Label ILP | Label EC |
|---|---|---|---|
| Predicted | ILP | 13 | 682 |
| | EC | 0 | 5900 |
| Recall | | **100.0%** | |
| F1-score | | **0.0367** | |

(b)

| | | Label ILP | Label EC |
|---|---|---|---|
| Predicted | ILP | 2 | 244 |
| | EC | 11 | 6338 |
| Recall | | 15.4% | |
| F1-score | | 0.0154 | |

Table 3.12: Decomposition Cost Comparison

| Circuit | ILP | | | SDP | | | EC | | | [66] | | | [66] + GNN decomposer | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | st# | cn# | cost | st# | cn# | cost | st# | cn# | cost | st# | cn# | cost | st# | cn# | cost |
| C432 | 4 | 0 | 0.4 | 4 | 0 | 0.4 | 4 | 0 | 0.4 | 4 | 0 | 0.4 | 4 | 0 | 0.4 |
| C499 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C880 | 7 | 0 | 0.7 | 7 | 0 | 0.7 | 7 | 0 | 0.7 | 7 | 0 | 0.7 | 7 | 0 | 0.7 |
| C1355 | 3 | 0 | 0.3 | 3 | 0 | 0.3 | 3 | 0 | 0.3 | 3 | 0 | 0.3 | 3 | 0 | 0.3 |
| C1908 | 1 | 0 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 0.1 | 1 | 0 | 0.1 |
| C2670 | 6 | 0 | 0.6 | 6 | 0 | 0.6 | 6 | 0 | 0.6 | 6 | 0 | 0.6 | 6 | 0 | 0.6 |
| C3540 | 8 | 1 | 1.8 | 8 | 1 | 1.8 | 8 | 1 | 1.8 | 8 | 1 | 1.8 | 8 | 1 | 1.8 |
| C5315 | 9 | 0 | 0.9 | 9 | 0 | 0.9 | 9 | 0 | 0.9 | 9 | 0 | 0.9 | 9 | 0 | 0.9 |
| C6288 | 205 | 1 | 21.5 | 203 | 4 | 24.3 | 203 | 5 | 25.3 | 205 | 1 | 21.5 | 205 | 1 | 21.5 |
| C7552 | 21 | 1 | 3.1 | 21 | 1 | 3.1 | 21 | 1 | 3.1 | 21 | 1 | 3.1 | 21 | 1 | 3.1 |
| S1488 | 2 | 0 | 0.2 | 2 | 0 | 0.2 | 2 | 0 | 0.2 | 2 | 0 | 0.2 | 2 | 0 | 0.2 |
| S38417 | 54 | 19 | 24.4 | 48 | 25 | 29.8 | 54 | 19 | 24.4 | 54 | 19 | 24.4 | 54 | 19 | 24.4 |
| S35932 | 40 | 44 | 48 | 24 | 60 | 62.4 | 46 | 44 | 48.6 | 40 | 44 | 48 | 40 | 44 | 48 |
| S38584 | 117 | 36 | 47.7 | 108 | 46 | 56.8 | 116 | 37 | 48.6 | 117 | 36 | 47.7 | 117 | 36 | 47.7 |
| S15850 | 97 | 34 | 43.7 | 85 | 46 | 54.5 | 100 | 34 | 44 | 97 | 34 | 43.7 | 97 | 34 | 43.7 |
| average | | | 12.893 | | | 15.727 | | | 13.267 | | | 12.893 | | | 12.893 |
| ratio | | | **1.000** | | | 1.220 | | | 1.029 | | | **1.000** | | | **1.000** |

indicates the number of graphs which is labeled as positive (ILP) and also selected to be decomposed by ILP-based decomposer. In the experiment, we use two more metrics, recall and F1 score. Recall is used to measure the proportion of ILP-labeled graphs that are correctly identified and influences the decomposition quality directly. F1-score is a general metric for the model's accuracy. According to Table 3.11, we can see that the F1-score of our model is more than 2× of that in conventional GCN, which demonstrates the powerful representation capability of our model compared with conventional GCN. Another important point is that our model classifies all the graphs labeled as positive correctly such that our recall achieves 100% while conventional GCN only classifies 15.4% correctly.

### Comparison with other state-of-the-art methods

In the second experiment, we compare our results with state-of-the-art decomposers. All the decomposers are implemented and measured in OpenMPL under one thread such that

Table 3.13: Decomposition Runtime Comparison

| Circuit | ILP | SDP | EC | [66] | [66] + GNN decomposer |
|---|---|---|---|---|---|
| C432 | 0.486 | 0.016 | 0.005 | 0.007 | 0.024 |
| C499 | 0.063 | 0.018 | 0.011 | 0.015 | 0.023 |
| C880 | 0.135 | 0.021 | 0.010 | 0.014 | 0.032 |
| C1355 | 0.121 | 0.024 | 0.011 | 0.015 | 0.025 |
| C1908 | 0.129 | 0.024 | 0.017 | 0.031 | 0.023 |
| C2670 | 0.158 | 0.044 | 0.035 | 0.046 | 0.040 |
| C3540 | 0.248 | 0.086 | 0.032 | 0.038 | 0.043 |
| C5315 | 0.226 | 0.106 | 0.039 | 0.049 | 0.027 |
| C6288 | 5.569 | 0.648 | 0.151 | 0.154 | 0.775 |
| C7552 | 0.872 | 0.157 | 0.071 | 0.111 | 0.108 |
| S1488 | 0.147 | 0.031 | 0.013 | 0.016 | 0.023 |
| S38417 | 7.883 | 1.686 | 0.329 | 0.729 | 0.140 |
| S35932 | 13.692 | 5.130 | 0.868 | 1.856 | 0.373 |
| S38584 | 13.494 | 4.804 | 0.923 | 1.840 | 0.310 |
| S15850 | 11.380 | 4.320 | 0.864 | 1.792 | 0.328 |
| average | 3.640 | 1.141 | 0.225 | 0.448 | 0.153 |
| ratio | 1.000 | 0.313 | 0.062 | 0.123 | 0.042 |

Table 3.14: F1 score of stitch redundancy prediction. The results include (a) all instances (b) instances whose prediction confidence are above the bar.

(a)

| | | Label | |
|---|---|---|---|
| | | No Need | Need |
| Pred. | No Need | 5962 | 46 |
| | Need | 55 | 498 |
| Recall | | 99.23% | |
| F1-score | | 0.9916 | |

(b)

| | | Label | |
|---|---|---|---|
| | | No Need | Need |
| Pred. | No Need | 5730 | 0 |
| | Need | 2 | 185 |
| Recall | | **100.0%** | |
| F1-score | | **0.9998** | |

we can keep the preprocess procedure the same and compare the results without potential bias due to different simplification method or stitch insertion techniques. Table 3.12 lists the decomposition cost of all decomposers. Table 3.13 lists all the decomposition runtime regardless of graph simplification and stitch insertion for better comparison. As expected, there is no one existing decomposer which can dominate others among existing decomposers. EC-based decomposer outperforms others on runtime while causing some additional costs. ILP-based decomposer obtains the optimal results while the runtime is significantly worse than others. SDP-based decomposer shows a runtime improvement compared with ILP-based decomposer but cannot compete with EC-based decomposer on both runtime and quality. If our proposed non-stitch GNN decomposer is not integrated, our framework [66] still obtains the optimal results in all cases since the selector selects all graphs labeled as ILP correctly and such avoid optimality loss. The average runtime is reduced to 12.3% compared with ILP-based decomposer because of the efficient graph matching technique and EC-based decomposer which is selected as the decomposer in most cases. Moreover, when we integrate the GNN decomposer into our framework, the runtime can be further reduced to 4.2% with the results being optimal. The main reasons for the large improvement is because of 1) the existence of considerable graphs that need not stitches, 2) the efficiency of our proposed purely GNN decomposer under GPU acceleration.

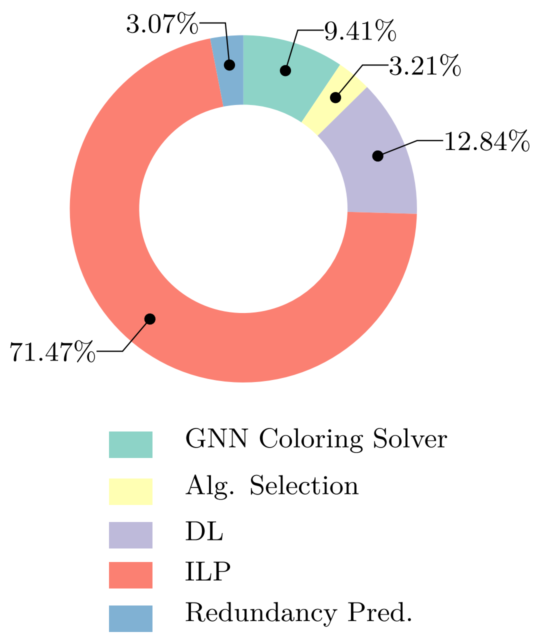


Figure 3.19: Runtime breakdown of our framework.

### Runtime analysis

In the third experiment, we compare the runtime distribution in our framework. The decomposition runtime of our framework is mainly composed of five parts: decomposition runtime by our GNN decomposer, decomposition runtime by ILP-based decomposer, decomposition runtime by DL-based decomposer, algorithm selection time, and the runtime for the stitch redundancy prediction. The runtime for graph matching and graph embedding are counted in the decomposer selection since the 2-class classifier is integrated into the graph embedding network for fast inference. Figure 3.19 shows the result, where the metric is the total decomposition runtime of 15 layouts as before. From the figure, we can clearly see that the decomposition runtime by the selected decomposer (ILP and DL) is the major bottleneck and occupies 84.31% of the total runtime. The result indicates that the RGCN inference and graph matching runtime of our framework are actually triv-

Table 3.15: Layout statistics and results by GNN decomposer

| Circuit | $\|G\|$ | $\|nsc\text{-}G\|$ | $\|ns\text{-}G\|$ | $\|pred.\ ns\text{-}G\|$ | ILP cost | GNN cost | ILP time | GNN time |
|---|---|---|---|---|---|---|---|---|
| C432 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0.0135 |
| C499 | 4 | 0 | 4 | 1 | 0 | 0 | 0.0041 | 0.0134 |
| C880 | 8 | 0 | 1 | 0 | 0 | 0 | 0 | 0.0135 |
| C1355 | 6 | 0 | 3 | 2 | 0 | 0 | 0.0045 | 0.0136 |
| C1908 | 3 | 0 | 2 | 0 | 0 | 0 | 0 | 0.0135 |
| C2670 | 9 | 0 | 3 | 0 | 0 | 0 | 0 | 0.0136 |
| C3540 | 14 | 0 | 6 | 0 | 0 | 0 | 0 | 0.0140 |
| C5315 | 16 | 0 | 7 | 2 | 0 | 0 | 0.0619 | 0.0135 |
| C6288 | 200 | 0 | 25 | 17 | 0 | 0 | 0.0051 | 0.0143 |
| C7552 | 39 | 1 | 16 | 3 | 0 | 0 | 0.0045 | 0.0135 |
| S1488 | 8 | 0 | 6 | 6 | 0 | 0 | 0.0339 | 0.0135 |
| S38417 | 670 | 3 | 613 | 584 | 16 | 16 | 2.8480 | 0.0169 |
| S35932 | 2010 | 12 | 1958 | 1869 | 22 | 22 | 8.6960 | 0.0164 |
| S38584 | 1936 | 3 | 1817 | 1735 | 22 | 22 | 8.1550 | 0.0171 |
| S15850 | 1657 | 4 | 1556 | 1510 | 22 | 22 | 6.8680 | 0.0153 |
| average | 440.27 | 1.533 | 401.13 | 381.93 | 5.467 | 5.467 | 1.778 | 0.0152 |
| ratio | 1.000 | 0.003 | 0.911 | 0.867 | **1.000** | **1.000** | 1.000 | **0.008** |

ial such that our method has strong scalability and can be applied to select other more efficient decomposers in the future. The inference runtime of algorithm selection and redundancy prediction are very close because both of them use RGCN as the backbone with the same parameters.

#### Effectiveness of Redundant Stitch Prediction

In the fourth section, we demonstrate the effectiveness of our GNN-based stitch redundancy predictor empirically. The results are presented in Table 3.14 in the form of confusion matrix. Table 3.14 (a) counts all instances, and (b) only counts instances whose prediction score is larger than 0.99 are selected. According to the results, we can see that our GNN-based predictor successful predicts most redundancy, which larges improves the efficiency. More importantly, benefit from the bar constraint, the prediction avoids any false prediction that predicts a graph with a "need-stitch" label as "no-need stitch". A more detailed result for each circuit is shown in Table 3.15, where $|pred.\ ns\text{-}G|$ is the number of successful predictions among all instances.

#### Effectiveness of Non-Stitch Decomposer by GNN

In the final experiment, we separately study the effectiveness of our proposed GNN-based decomposer which is specifically for non-stitch graphs. The results are shown in Table 3.15, where $G$ is the graph set after simplification and stitch insertion. *nsc*-$G$ (no stitch candidate graph) is a subset of $G$ in which graphs contain no stitch edges. *ns*-$G$

(no stitch graph) is a subset of $G$ in which the optimal decomposition results contain no stitches. *pred. ns*-$G$ (predicted no stitch graph) is a subset of $G$ in which our proposed stitch redundancy predictor predicts that these graphs do not need stitch edges. ILP(GNN) cost represents the total cost decomposed by ILP method (Our proposed GNN decomposer) for graphs in *pred. ns*-$G$, ILP time is the total decomposition time by ILP method for graphs in *pred. ns*-$G$. GNN time is the total execution time by our GNN decomposer. Since we implement the decomposer in a batch-process manner, we use the GNN decomposer to decompose all graphs even before the stitch redundancy prediction rather than waiting the prediction result of each case (note that the additional runtime is trivial for the fast inference). Therefore, in some layouts, such as `C432`, our GNN decomposer still decomposes some graphs even there is no redundant graphs according to the prediction, i.e., $|pred.\ ns\text{-}G| = 0$.

According to Table 3.15, we can observe some statistical properties in the layout dataset. First, existing stitch candidate generation algorithm will insert stitch candidates into most graphs. Among over 6,000 graphs, only 23 graphs are free of stitch edges. However, we observe that most of these inserted stitches edges are not useful: in the final optimal results, 91.1% graphs contain no stitches, meaning that considerable generated stitch candidates are redundant. Our predictor can predict the redundancy with a high accuracy (381.93 over 401.13). Then, for graphs whose stitch edges are predicted as redundant, we can employ our GNN based decomposer, which is specifically for the homogeneous graphs, i.e., graphs only containing conflict edges. As shown in the table, our GNN decomposer achieves the same result quality with the optimal ILP solver, with a large improvement on the efficiency (reduce to 0.8%).

□ **End of chapter.**

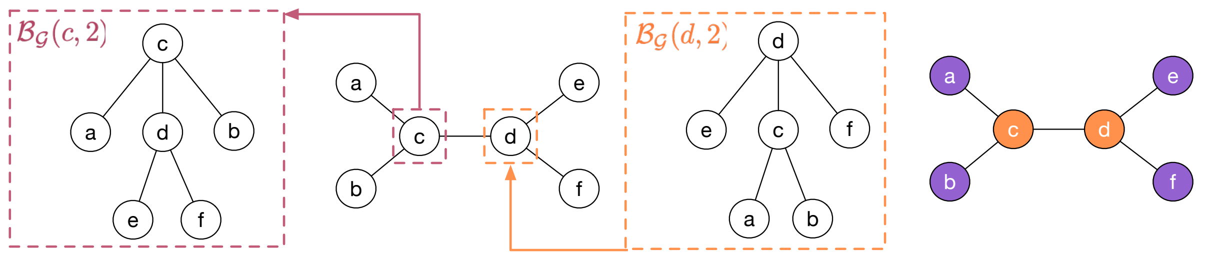
# Chapter 4

# Graph Coloring

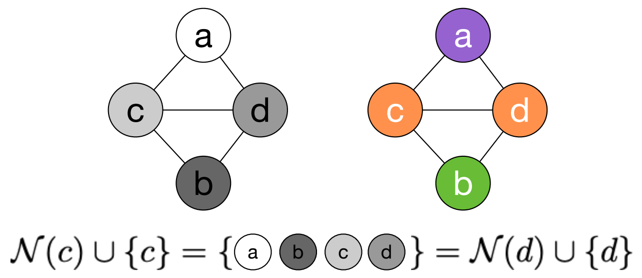
Graph neural networks (GNNs) have shown overwhelming success in various fields, such as molecules, social networks, and web pages[42]. The main idea behind GNNs is a neighborhood aggregation scheme (or called message passing), where each node aggregates feature vectors from its neighbors and combines them with its own feature vector to produce a new one. GNNs following such a scheme are also called aggregation-combination GNNs (AC-GNNs) [11]. After finite iterations of such aggregation and combination, the corresponding feature vector of each node is called node embedding to represent the node.

The development of GNNs stimulated the interest in their applications to the NP-hard problems (MIS [69], graph coloring [48], graph matching [72]), while most works apply GNNs for NP-hard problems by empirical intuition and experimental trials. In this work, we focus on solving the graph coloring problem using GNNs by investigating the power of GNNs *for the graph coloring problem*. Some recent works [73, 11, 117, 35, 74, 29] study the power of a GNN by analyzing when a GNN maps two nodes to the same node embedding. In their study, a maximally powerful GNN with depth $L$ should map two $L$-local equivalent nodes to the same node embedding [117, 73]. However, when applied in the graph coloring problem, such a definition raises some problems. First, the coloring task is not under homophily but *heterophily*. Graphs under heterophily are the ones where connected nodes are assigned different labels/features/colors instead of a similar one (homophily). Therefore, two local equivalent nodes are not necessarily assigned the same node embedding, which contradicts with previous study. One example can be found in Figure 4.1(a), where the node pair $\{c, d\}$ is local equivalent but should assigned different colors to avoid the conflict. Second, every AC-GNN is bounded by its depth $L$. Therefore, the maximally powerful GNN is identified by $L$-local equivalence instead of a *global* equivalence. This constraint makes an AC-GNN possible to be a *local method*, which has been demonstrated to be non-optimal in many NP-hard problems such as MIS [90, 36].

Motivated by these limitations, we define the power of AC-GNNs in the coloring problem as its ability to assign nodes different colors. We then observe and theoretically prove a set of conditions that make AC-GNNs powerful in the graph coloring problem. Based on these observations, we develop a series of rules to design powerful AC-GNNs *specifically for the graph coloring problem.* Besides the study on the power of GNNs, we aslo explore the color equivariance of GNNs to satisfy the requirement for pre-fixing colors of some nodes. We make the following contributions: (1): We discuss the limitations of


$\mathcal{B}_{\mathcal{G}}(c,2)$
$\mathcal{B}_{\mathcal{G}}(d,2)$
a
b
c
d
e
f

(a)


a
b
c
d
$\mathcal{N}(c) \cup \{c\} = \{$ a b c d $\} = \mathcal{N}(d) \cup \{d\}$

(b)

Figure 4.1: Examples of graph structures in which some AC-GNNs fail to discriminate the equivalent node pair $\{c, d\}$. (a) left: the input graph with the same node attribute; right: the coloring results by the most powerful AC-GNN. (b) left: the input graph with different node attributes (represented by the gray scale); right: the coloring results by the most powerful *integrated* AC-GNN. The aggregation for any *integrated* AC-GNN in both $c$ and $d$ are the same since $\mathcal{N}(c) \cup \{c\} = \mathcal{N}(d) \cup \{d\}$. Here, the most powerful refers to the discrimination power,i.e., the ability to assign nodes different colors.

main-stream AC-GNNs for the graph coloring problem from two perspectives: the coloring task is under heterophily, resulting in a contradiction in the definition of the power of GNNs; AC-GNNs are shown to be local methods, and the further proof demonstrates that any local methods (including AC-GNNs) cannot be optimal in the coloring problem. We also prove the positive correlation between model depth and its power in the coloring problem. (2) We give a necessary and sufficient condition for a simple AC-GNN to be color equivariant. (3) We summarize and theoretically prove a series of rules that make a GNN color equivariant and powerful in the coloring problem. Combining these rules, we develop a simple GNN-based approach by un-supervised learning, which proves to enhance the discrimination power and retain the color equivariance. (4) We validate our findings by extensive empirical evaluation including five datasets from different subjects. Our method shows substantially superior performance compared with other existing AC-GNN variations and even outperforms state-of-the-art heuristic algorithms with a significant efficiency improvement.

## 4.1 Preliminaries

**Local graph teminology.** Here, we introduce terms used in local graph analysis and follow the same definition in [36]. We leave other graph terminologies in Appendix A. For every positive integer $r$ and every node $u \in \mathcal{V}$, we define $\mathcal{B}_{\mathcal{G}}(u, r)$ as the subgraph of $\mathcal{G}$ induced by node $u$ with distance at most $r$ from $u$. One example is given in Figure 4.1(a). Consider two nodes $u, v \in \mathcal{V}$, we say $\{u, v\}$ is $r$*-local equivalent* if it is $r$-local topologically equivalent by $\pi_r$ and $\boldsymbol{x}_w = \boldsymbol{x}_{\pi_r(w)}$ holds for every $w \in \mathcal{B}_{\mathcal{G}}(u, r)$.

**Graph coloring.** Let $k$ be the number of available colors, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the input graph and each vertex $v \in \mathcal{V}$ be associated with an attribute $\boldsymbol{x}_v$, a coloring function $f_k : (v, \mathcal{G}, \boldsymbol{x}_v) \to \{1, ..., k\}$ returns a color of $v$ indexed by $col_v \in \{1, ..., k\}$. In the following pages, $k$ follows the same definition if not specified and $f(\mathcal{G})$ represents the coloring solution on $\mathcal{G}$ by $f$ for simplification. Given a graph $\mathcal{G}$ colored by $f_k$, a *conflict function* $c : (u, v, f_k) \to \{0, 1\}$ is used to measure the performance of $f_k$ on $\mathcal{G}$. Specifically, $c(u, v, f_k) = 1$ when $u$ and $v$ are connected and assigned the same color:

$$c(u, v, f_k) = \begin{cases} 1, & \text{if } \ f_k(v, \mathcal{G}, \boldsymbol{x}_v) = f_k(u, \mathcal{G}, \boldsymbol{x}_u) \\ & \ \text{ and } \{u, v\} \in \mathcal{E}; \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

The edge $e = \{u, v\}$ is called a *conflict* if $c(u, v, f) = 1$. The objective of the graph coloring problem is widely formulated in two ways: 1) $k$*-coloring problem*: Given $k$, minimize the number of conflicts as in Equation (4.2); 2) Given a conflict constraint $c_{max}$, minimize the number of used colors as in Equation (4.3).

$$\min \sum_{\{u,v\} \in \mathcal{E}} c(u, v, f_k). \tag{4.2}$$

$$\min k, \ \text{ s.t. } \sum_{\{u,v\} \in \mathcal{E}} c(u, v, f_k) \le c_{max}. \tag{4.3}$$

When we set $c_{max}$ as 0, i.e., no conflict is introduced by $f_k$, we call the obtained minimum color number as the *chromatic number* of $\mathcal{G}$, $\mathcal{X}$. In the following pages, we say a coloring function is *optimal* if it colors the graph without conflict in the $\mathcal{X}$-coloring problem.

**Graph neural networks (GNNs).** GNNs are to learn the embedding of a node or the entire graph based on the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and node features $\{\boldsymbol{x}_v : v \in \mathcal{V}\}$. We follow the same notations in a previous work [11] to formally define the basics for GNNs. Let $\{\text{AGG}^{(i)}\}_{i=1}^{L}$ and $\{\text{COM}^{(i)}\}_{i=1}^{L}$ be two sets of *aggregation* and *combination* functions. An *aggregation-combine GNN* (AC-GNN) computes the feature vectors $\boldsymbol{h}_v^{(i)}$ for every node

$v \in \mathcal{V}$ by:

$$\boldsymbol{h}_v^{(i)} = \mathrm{COM}^{(i)}(\boldsymbol{h}_v^{(i-1)}, \mathrm{AGG}^{(i)}(\{\boldsymbol{h}_u^{(i-1)} : u \in \mathcal{N}(v)\})), \tag{4.4}$$

where $\mathcal{N}(v)$ denotes the neighborhood of $v$, i.e., $\mathcal{N}(v) = \{u : \{u, v\} \in \mathcal{E}\}$ and $\boldsymbol{h}_v^{(0)}$ is the node attribute $\boldsymbol{x}_v$. Finally, each node $v$ is classified by a node classification $CLS(\cdot)$ applied to the node embedding $\boldsymbol{h}_v^{(L)}$. When the AC-GNN is used for the graph coloring problem, $CLS(\cdot)$ returns a $col_v \in \{1, ..., k\}$. Then, an AC-GNN $\mathcal{A}$ with $L$ layers is also called $L$-AC-GNN and defined as $\mathcal{A} = (\{\mathrm{AGG}^{(i)}\}_{i=1}^{L}, \{\mathrm{COM}^{(i)}\}_{i=1}^{L}, CLS(\cdot))$. Here, we define $\mathcal{A}(v, \mathcal{G}, \boldsymbol{x}_v)$ as the color of $v$ assigned by $\mathcal{A}$.

*simple AC-GNN*: The properties of aggregation, combination and classification functions are widely studied in recent years [11, 117, 63, 82, 55, 115] and many variations of these functions are proposed. Among various function architectures, we say an AC-GNN is *simple* as in [11] if the aggregation and combination functions are defined as follows:

$$\mathrm{AGG}^{(i)}(\mathbb{X}) = \sum_{\boldsymbol{x} \in \mathbb{X}} \boldsymbol{x}, \tag{4.5}$$

$$\mathrm{COM}^{(i)}(\boldsymbol{x}, \boldsymbol{y}) = \sigma(\boldsymbol{x}\boldsymbol{C}^{(i)} + \boldsymbol{y}\boldsymbol{A}^{(i)} + \boldsymbol{b}^{(i)}), \tag{4.6}$$

where $\boldsymbol{C}^{(i)}$, $\boldsymbol{A}^{(i)}$, and $\boldsymbol{b}^{(i)}$ are trainable parameters, $\sigma$ is an activation function.

*integrated AC-GNN*: The aggregation and combination functions can also be integrated such as networks explored in [55, 73]. We say that such AC-GNN is *integrated* when aggregation and combination functions are integrated as follows:

$$\boldsymbol{h}_u^{(i)} = \mathrm{COM}^{(i)}(\mathrm{AGG}^{(i)}(\{\boldsymbol{h}_w^{(i-1)} : w \in \mathcal{N}(u) \cup \{u\}\})). \tag{4.7}$$

In integrated AC-GNNs, aggregation functions aggregate features from neighborhood and the node itself simultaneously, which means they treat the neighborhood information and ego-information (information from the node itself) equally.

## 4.2 Powerful GNNs for Graph Coloring

In this section, we focus on the question: *What kinds of designs make a GNN more/less powerful in the graph coloring problem?* Although GNNs demonstrate their power in various tasks, most of them even cannot beat the simplest heuristic algorithms in the coloring problem. One motivating experiment is shown in Table 4.1, where the solved ratio is one minus the ratio between the number of conflicts and the number of edges. For example, given a graph with 100 edges, if a coloring function colors the graph with 10 conflicts, then the solved ratio is calculated by $1 - 10/100 = 0.9$. The Greedy represents the greedy algorithm that colors nodes in the order of node IDs. We observe that all tested GNNs do not work at all in the coloring problem. We analyze the reasons from two perspectives.

First, the graph coloring problem is under heterophily instead of homophily. That is, linked nodes are more likely from different color classes rather than a same one. However, previous studies define the power of GNNs as the capability to maps two equivalent nodes to the same embedding. Under the heterophily, it is critical to rethink the definition of a GNN's power *specifically for the coloring problem.* After the power is formalized, the next question is: *What factors may enhance or harm the power?*

Table 4.1: Solved ratio of existing GNNs and the simplest greedy algorithm on layout dataset over three runs. The node attributes are set to all-one vectors. $d$: depth.

| | $d = 2$ | $d = 10$ |
|---|---|---|
| GCN [55] | 0.55 $\pm$0.01 | 0 $\pm$0 |
| SAGE [41] | 0 $\pm$0 | 0 $\pm$0 |
| GIN [117] | 0.59 $\pm$0.01 | 0.58 $\pm$0.01 |
| GAT [106] | 0 $\pm$0 | 0 $\pm$0 |
| Greedy | **0.962** | |

Besides the concern on heterophily, every AC-GNN is bounded by its depth $L$. This constraint makes an AC-GNN possible to be a *local method*, which has been demonstrated to be non-optimal in many NP-hard problems such as MIS [90, 36]. Then, we try to figure out *whether AC-GNNs are local methods* and, if they are, *whether a local method can be optimal for the graph coloring problem*?

Here, we discuss the power of GNNs for graph coloring by answering the questions raised above. We leave all proofs in Appendix B due to the page limit.

### 4.2.1 Discrimination power under heterophily

**Q:** *How to determine whether a GNN is powerful in coloring problem?*

In the graph coloring problem, connected nodes are assigned to different colors. Therefore, a powerful GNN should map the two connected nodes to node embeddings as differently as possible. Intuitively, we can study the power of a GNN in the coloring problem by analyzing ***its ability to assign nodes different colors.*** Here, we refer the power as the *discrimination power* of GNNs to differ with previous expressive power under homophily. Formally, we define that a coloring method $f$ discriminates a node pair $(u, v)$ as follows:

**Definition 2** (discriminate). *A coloring method $f$ discriminates a node pair $(u, v)$ if $f$ assigns $u$ and $v$ different colors, i.e., $f(u, \mathcal{G}, \boldsymbol{x}_u) \neq f(v, \mathcal{G}, \boldsymbol{x}_v)$.*

**Definition 3** (distinct node pair). *A node pair $(u, v)$ is a distinct node pair if the colors of $u$ and $v$ are different in any optimal solution.*

Following the definitions, we can answer the question above: *more powerful a GNN is, more distinct node pairs it will discriminate.* Ideally, an optimal GNN should be able to discriminate all distinct node pairs.

Given the definitions above, one may try to build an optimal AC-GNN which colors all graphs without conflict through discriminating all distinct node pairs:

**Q:** *Can we design an AC-GNN which discriminates any distinct node pair?*

The following Proposition 1 refutes the existence of such a “perfect” AC-GNN:

**Property 1.** *All AC-GNNs cannot discriminate any equivalent node pair.*

The equivalent (and also distinct) node pair $\{c, d\}$ in Figure 4.1(a) is one example of such node pair. Since the equivalent node pair have the the same subgraph structure and the same node attribute distributions, the AC-GNN always return the same results in each layer. Hence, AC-GNNs are not optimal for any graph that contains these node pairs, i.e., connected and also equivalent pairs.

To avoid such a non-optimal case, we can break the equivalence between two nodes by assigning different node attributes such as random features [94] or one-hot vectors [11]. The solution also aligns with the conclusion in [73], which proves that with different attributes GNNs become significantly more powerful. Indeed, making nodes different purposely strengthens the AC-GNN by eliminating the equivalent node pairs (although the topological equivalence is preserved). However, the superficial methods on the node attributes cannot influence and solve the underlying defects of some specific AC-GNNs, for example, integrated AC-GNN. Considering that an integrated AC-GNN treats the node feature and features of its neighbors equally, i.e., all of them are aggregated into the same multi-set, it is more difficult for integrated AC-GNNs to discriminate the node with its neighbors. Property 2 points out that an integrated AC-GNN cannot be optimal since there always exists a set of graphs which at least one distinct node pair is not discriminated by the integrated AC-GNN:

**Property 2.** *If nodes $u$ and $v$ in a graph $\mathcal{G}$ is connected and share the same neighborhood except each other, i.e., $\mathcal{N}(u)\backslash\{v\} = \mathcal{N}(v)\backslash\{u\}$, then an integrated AC-GNN cannot discriminate $\{u, v\}$.*

The property points the deficiency of integrated AC-GNN even if nodes are differentiable by their attributes. One example is given in Figure 4.1(b), where the distinct node pair $\{c, d\}$ cannot be discriminated by any integrated AC-GNN even the node attributes are different.

### 4.2.2 Locality

Local methods are widely used in the combinatorial optimization problems such as maximum independent set (MIS) and graph coloring. The formal definition of local method for the coloring problem is described as follows, which is a direct rephrasing in [36]:

**Definition 4** (local method [36])**.** *A coloring method $f$ is $r$-local if it fails to discriminate any $r$-local equivalent node pair. A coloring method $f$ is local if $f$ is $r$-local for at least one positive integer $r$.*

Along with the study of the local methods, the upper bound of a local method for the MIS problem is investigated. David et al. [36] gives an upper bound $1/2 + 1/(2\sqrt{2})$ of the size of an MIS produced by any local method in the random $d$-regular graph as $d \to \infty$

and [90] strengthens the bound to $1/2$. A random $d$-regular graph is a graph with $n$ nodes and the nodes in each node pair are connected with a probability $d/n$. Starting from the upper bound of any local method for the MIS problem, we may try to figure out:

**Q:** *Whether a local method is also non-optimal in the graph coloring problem?*

The answer is yes. we finish the proof by making use of the upper bound studied in the MIS problem and bridging the connection between a local method for MIS problem and coloring problem. Corollary 1 states the non-optimality of a local method for the graph coloring problem:

**Corollary 1.** *A local coloring method is non-optimal in the random $d$-regular tree as $d \to \infty$.*

Due to the localized nature of the aggregation function in GNNs, an AC-GNN with a fixed number of layers, say $L$ layers, cannot detect the structure or information of nodes at a distance further than $L$. Considering the non-optimality of a local coloring method stated in Corollary 1 and the localized nature of GNNs, we can reduce our analysis of whether there exists an optimal AC-GNN for graph coloring to whether an AC-GNN is a local coloring method? Corollary 2 answers the question as yes:

**Corollary 2.** *$L$-AC-GNN is a $L$-local coloring method and thus a local coloring method.*

Corollary 1 and Corollary 2 directly lead to the following theorem:

**Theorem 4.** *AC-GNN is not optimal, specifically for the random $d$-regular tree as $d \to \infty$.*

Based on the analysis above, we can see that the locality of AC-GNN makes it infeasible to to be an optimal coloring function. To solve the problems raised by locality of AC-GNNs, which inhibits AC-GNNs from detecting the global graph structure, many efforts have been made to devise a global scheme such as global readout functions [11], randomness [126, 27] and deeper networks [61, 21, 118]. Among all global techniques, a deep architecture is believed to be global as long as it covers the full graph. Given a graph with diameter $R$, a $R$-AC-GNN is indeed able to detect the information from the whole graph. However, it is impossible to find an AC-GNN which is able to cover all graphs: it is always bounded by its depth. Then, if we cannot develop an optimal AC-GNN by simply stacking layers, does this method contribute to the discrimination power? Formally:

**Q:** *Is deeper AC-GNN more powerful in the coloring problem?*

We answer the question as yes, and gives a more specific statement:

**Property 3.** *Let $\{u, v\}$ be a node pair in any graph $\mathcal{G}$, and $L$ be any positive integer. If a $L$-AC-GNN discriminates $\{u, v\}$, a $L^+$-AC-GNN also discriminates it.*

A $L^+$-AC-GNN is an AC-GNN by stacking injective layers after $L$-AC-GNN (before $CLS(\cdot)$). An injective layer includes a pair of injective aggregation function and injective combination function.

Besides deeper networks, we also review and discuss other mainstream global techniques, and proves whether they are truly global following Definition 4 [36]. All details are leaved in Appendix E.

## 4.3 Color Equivariance

In the coloring problem, the node attribute and the final features can be set as the probability distribution of colors, i.e., color beliefs, as in [15, 60, 138]. For example, the node attribute (probability distribution) of $u$: $\boldsymbol{u}$ =[0.5 (red), 0.2 (blue), 0.3 (green)] means that the node $u$ initially has 50% probability to be colored as red, 20% as blue, and 30% as green. Under this assumption, not only should we consider the order equivariance, but also the color equivariance.

Equivariance [134, 88, 78, 117, 35] is an important property for a function if it is defined on the input elements that are *equivariant* to the permutation of the elements. Color equivariance is not relevant to the discrimination power, whereas its importance emerges in practical applications such as the layout decomposition problem [50], where each color represents a mask and some metal features (nodes) are pre-assigned to some specific masks (colors). Let's continue with the example $u$: $\boldsymbol{u}$ =[0.5 (red), 0.2 (blue), 0.3 (green)]. If we are required to pre-color node $u$ to be red color and one solution for an AC-GNN is to modify the node attribute of $u$ to $\boldsymbol{u}$ =[1.0, 0, 0], i.e., predefine the possibility of red color as 100%. In this case, if the AC-GNN is not color equivariant, the final feature of $u$ obtained by AC-GNN may not set red as $u$'s color. That is, only color equivariant AC-GNN knows the differences of colors.

To investigate conditions of functions to be color equivariant, we first formalize the definition of an equivariant function:

**Definition 5** (equivariance [78, 134])**.** *A function $f : \mathbb{R}^k \to \mathbb{R}^k$ is equivariant if $f(\boldsymbol{h})\boldsymbol{P} = f(\boldsymbol{hP})$ for any permutation matrix $\boldsymbol{P} \in \mathbb{R}^{k\times k}$ and feature vector $\boldsymbol{h} \in \mathbb{R}^k$.*

Similarly, *color equivariant* follows the definition above, where $\boldsymbol{h} \in \mathbb{R}^k$ is the color belief. A simple AC-GNN $\mathcal{A}$ with $L$ layers is color equivariant if and only if all functions in $\{\text{COM}^{(i)} = \sigma(\boldsymbol{x}\boldsymbol{C}^{(i)} + \boldsymbol{y}\boldsymbol{A}^{(i)} + \boldsymbol{b}^{(i)}) : i \in 1, ..., L\}$ are color equivariant. Then, the following theorem states the sufficient and necessary conditions for $\mathcal{A}$ to be color equivariant:

**Theorem 5.** *Let $\mathcal{A}$ be a simple AC-GNN and both input and output be the probability distribution of $k$ colors, $\mathcal{A}$ is color equivariant if and only if the following conditions hold:*

- *For any layer $i$, all the off-diagonal elements of $\boldsymbol{C}^{(i)}$ are tied together and all the diagonal elements are equal as well. That is,*

$$\begin{aligned} &\boldsymbol{C}^{(i)} = \lambda_C^{(i)}\boldsymbol{I} + \gamma_C^{(i)}(\boldsymbol{1}\boldsymbol{1}^\top) \\ &\lambda_C^{(i)}, \gamma_C^{(i)} \in \mathbb{R} \quad ; \boldsymbol{1} = [1, ..., 1]^\top \in \mathbb{R}^k. \end{aligned} \tag{4.8}$$

- *For any layer $i$, all the off-diagonal elements of $\boldsymbol{A}^{(i)}$ are also tied together and all the diagonal elements are equal as well. That is,*

$$\begin{aligned} &\boldsymbol{A}^{(i)} = \lambda_A^{(i)}\boldsymbol{I} + \gamma_A^{(i)}(\boldsymbol{1}\boldsymbol{1}^\top) \\ &\lambda_A^{(i)}, \gamma_A^{(i)} \in \mathbb{R} \quad \boldsymbol{1} = [1, ..., 1]^\top \in \mathbb{R}^k. \end{aligned} \tag{4.9}$$

- *For any layer $i$, all elements in $\boldsymbol{b}^{(i)}$ are equal. That is,*

$$\boldsymbol{b}^{(i)} = \beta^{(i)}\boldsymbol{1} \quad \beta^{(i)} \in \mathbb{R} \quad \boldsymbol{1} = [1, ..., 1]^\top \in \mathbb{R}^k. \tag{4.10}$$

The theorem above is actually an extension of Lemma 3 in [134] from a standard neural network layer $f = \epsilon(\Theta\boldsymbol{x})$ to a simple AC-GNN. Based on Theorem 5, a simple AC-GNN is color equivariant when the trainable matrices/vectors $\boldsymbol{C}, \boldsymbol{A}, \boldsymbol{b}$ in Eq. equation 4.6 are calculated by several scalars, i.e., $\lambda, \gamma, \beta$.

## 4.4 Our Method

Based on the discussions above, we summarize a series of rules that make a GNN $\mathcal{A}$ color equivariant and enhance its discrimination power as follows: (1) The input graph contains no equivalent node pair (Property 1); (2) $\mathcal{A}$ does not integrate the aggregation and combination function (Property 2); (3) $\mathcal{A}$ should be as deep as possible (Property 3); (4) Layers in $\mathcal{A}$ should be injective (Property 3) (5) If $\mathcal{A}$ follows the form of a simple AC-GNN, it should satisfy the conditions in Theorem 5 to make it color equivariant.

With the guidance of the rules above, we propose a very simple architecture, *Graph Discrimination Network (TreeNet)* based on simple AC-GNN, Note that there are not solely one architecture that satisfies all the rules above. We select TreeNet as an example considering the balance between efficiency and performance. The discussion and experimental results of other models that satisfy the rules above are given in the Appendix. We describe TreeNet for the graph coloring problem as follows:

**Forward Computation** For a $k$-coloring problem, the node attribute is the centered probability distribution of $k$ colors and is initialized randomly to eliminate the equivalent node pairs (rule 1). Formally, the node attribute $\boldsymbol{x} \in \mathbb{R}^k$ is calculated by:

$$\boldsymbol{x} = \boldsymbol{x}' - \frac{1}{k} \tag{4.11}$$

where $\boldsymbol{x}'$ is the random probability distribution.

The aggregation function is the same with Equation 4.5 (rule 2,4). Let $\boldsymbol{m}_v^{(i)} \in \mathbb{R}^k$ be the result returned by $\mathrm{AGG}^{(i)}$ for the node $v$ in the $i$-th layer, the aggregation layer is organized as follows:

$$\boldsymbol{m}_v^{(i)} = \sum_{u \in \mathcal{N}(v)} \boldsymbol{h}_u^{(i-1)} \tag{4.12}$$

In the combination function, we define the $\mathrm{COM}^{(i)}$ following Theorem 5 to make TreeNet color equivariant (rule 3,4,5):

$$\boldsymbol{h}_v^{(i)} = \boldsymbol{h}_v^{(i-1)}\lambda_C^{(i)} + \boldsymbol{h}_v^{(i-1)}\gamma_C^{(i)}(\mathbf{1}\mathbf{1}^\top) + \boldsymbol{m}_v^{(i)}\lambda_A^{(i)} + \boldsymbol{m}_v^{(i)}\gamma_A^{(i)}(\mathbf{1}\mathbf{1}^\top) + \beta^{(i)}\mathbf{1}$$

where $\lambda, \gamma, \beta$ are trainable scalars. Finally, the classification function $CLS(\cdot))$ in TreeNet is defined as an argmax function, since the final node embedding is still a probability distribution of colors.

**Loss Function** Considering that an optimal color solution keeps optimal regardless of the permutation of colors, it is not an easy job to develop a supervised training scheme since there is not only a single optimal solution. Li et al. [69] uses supervised learning followed by a heuristic tree search to alleviate the multi-solution issue in the MIS problem. However, the complexity of the tree search explodes when the number of colors $k$ and the number of nodes $n$ become large. Here, we use a un-supervised margin loss, motivated by the fact that final node embeddings of connected nodes should be as different as possible, and formulated by:

$$\min \sum_{\{u,v\}\in\mathcal{E}} \max\{m - d(\boldsymbol{h}_u, \boldsymbol{h}_v), 0\}. \tag{4.13}$$

where $\boldsymbol{h}_u \in \mathbb{R}^k$ is the probability distribution obtained by $\mathcal{A}$. $d$ is the Euclidean distance between the node pair. $m$ is the pre-defined margin.

**Preprocess & Postprocess** Our method also contains preprocess and postprocess procedures, which are widely used in other coloring methods [137, 66]. In the preprocess part, the node with degree less than $k$ is removed iteratively. In the postprocess part, we iteratively detect 1) whether a color change in a single node will decrease the cost or 2) whether a swap of colors between connected nodes will decrease the cost. We implement the two additional steps by tensor operations, which significantly boost the efficiency. The experiments on the two steps and detailed algorithms are shown in the Appendix D.

**Why training?** In our proposed GDN, only these scalars $(\lambda, \gamma, \beta)$ need to be trained. Indeed, it is viable to design a training-free version, i.e., pre-define these scalars. For example, by following the intuition that the feature of each node should be as different as its neighbor, we can directly set $\lambda, \gamma$ as positive and negative values respectively *without* training. However, it is not easy to find a "best" value for $\lambda, \gamma, \beta$ by theoretical analysis or by intuition. Therefore, we prefer a learning-based method, which learns these relationships through training. At the same time, the training scheme has a strong interpretability. For example, the ratio between $\lambda_A^1$ and $\lambda_C^1$ indicates the relative importance between the features of each node and its neighbors, and the ratio between scalars of different layers ($\lambda_A^1$ and $\lambda_A^2$) is the relative importance between neighbors of different depths. A detailed comparison and introduction of previous coloring works including non-training one and training one are covered in Appendix C. The experiments about different selection schemes of $\lambda, \gamma, \beta$ are covered in the Appendix.

## 4.5 Experiments

### 4.5.1 Experimental setup

Detailed settings, more experiments and analysis are shown in the Appendix. We evaluate our models and baselines on four datasets here, the basic information on these datasets are shown in Table 4.2, where the column $\mathcal{X}(k)$ is the chromatic number except layout dataset, which is set to 3 in real-world circuit design. The chromatic number in random dataset is not static due to its random nature. More information about dataset is shown in the supplement.

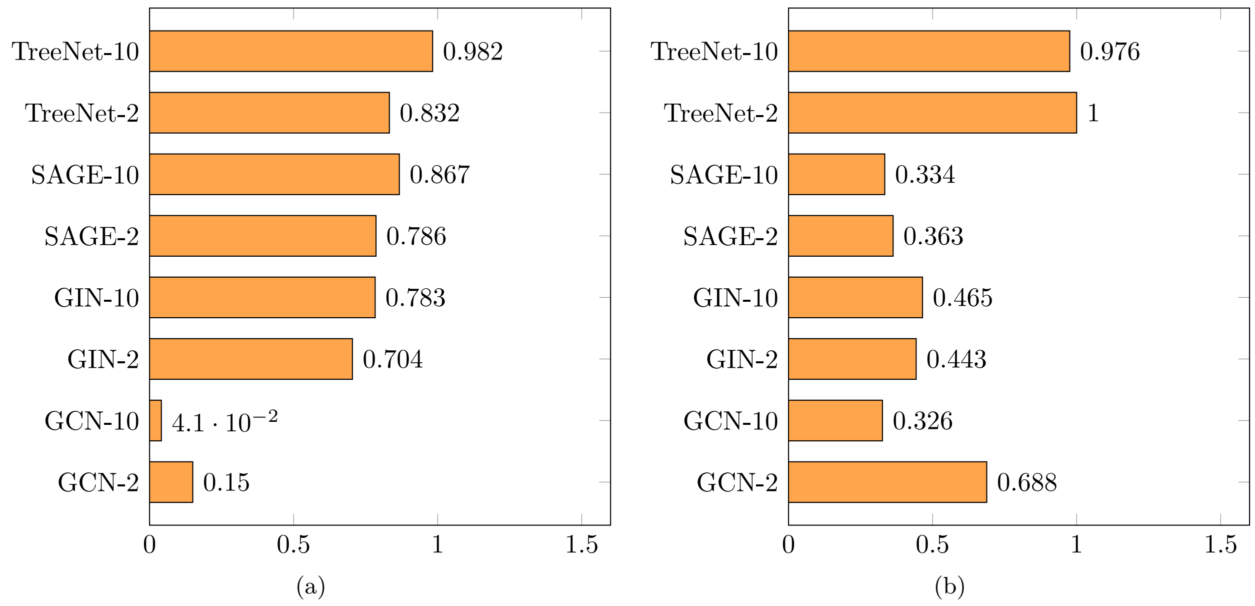


Figure 4.2: (a) Solved ratio by different AC-GNN variations; (b) Fixed color ratio by different AC-GNN variations.

We mainly compare our models with two previous works which focus on the graph coloring problem: (1) GNN-GCP [60], combing GNN, RNN, and MLP to obtain the node embedding and using a k-means method to color the node. We obtain models from the author and directly obtain the results. (2) Tabucol [44], a well-known heuristic algorithm using Tabu search. We follow the original setting with iteration limit of 1000 (or the time limit of 24 hours) and the number of uncolored node pairs is returned if the algorithm fails to find a perfect coloring assignment within the limit. We also compare different variants of AC-GNN in previous works: GCN [55], GIN [117] and GraphSAGE [41]. All AC-GNN variations are only tested in layout dataset since AC-GNN variations require a fixed number of colors to make output shape keep the same, More details on the datasets and baseline model configurations can be found in Appendix.

### 4.5.2 Comparison with other AC-GNN variations

The comparison with other trainable AC-GNN variations is conducted on the layout dataset. The results are shown in Figure 4.2(a). "GDN-$k$" represents TreeNet with a depth of $k$. According to the results, we observe the following: (1) GCN, the most representative integrated AC-GNN, is much worse than other AC-GNNs, which demonstrates our rule 1. (2) Most AC-GNNs benefit from a deeper network, which aligns with our rule 3. (3) Although other non-integrated AC-GNN, such as GIN and GraphSAGE, achieve an acceptable solved ratio, our method is far more better than other AC-GNNs.

We also validate the color equivariance of models by simulating the pre-color constraint in the layout decomposition problem. For each instance, we randomly select one node and set its color by changing the node attribute, known as the color distribution. We measure the color equivariant capability by checking the fixed color ratio, defined as the ratio between the number of successfully fixed graph and the number of total graphs. A successfully fixed graph is the graph whose selected node (metal feature) is colored as expected with the pre-assigned one by $\mathcal{A}$. From the results shown in Figure 4.2(b), we can see that the fixed color ratio of our TreeNet is much higher than other variations, matching with our analysis before,

### 4.5.3 Comparison with other graph coloring methods

The comparison with other graph coloring methods is conducted on all collected datasets. The results are shown in Table 4.2, where $k$ is the number of available colors and cost is the number of conflicts in the coloring result. GNN-GCP gives "-" if it fails to find a chromatic number prediction, while Tabucol gives "-" if it fails to color the graph within 24 hours. According to the results, we observe the following: (1) Our model is much more powerful than GNN-GCP, also with a better efficiency. (2) Our model outperforms the state-of-the-art heuristic algorithm with a slightly better result quality and 500$\times$ speedup.

Table 4.2: Graph datasets information and results by different coloring methods.

| **Dataset** | **Graph** | $\lvert\mathcal{V}\rvert$ | $\lvert\mathcal{E}\rvert$ | **d%** | $\mathcal{X}(k)$ | **GNN-GCP** |  | **Tabucol** |  | **Ours** |  |
|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  | Cost | Time | Cost | Time | Cost | Time |
| Layout | 35158 | 641202 | 787242 | - | 3 | 386009 | 3896 | 2392 | 82301 | **1557**±45 | 5.84 ±0.23 |
| ratio |  |  |  |  |  | 247.9 | 667.1 | 1.54 | 14092 | **1.0** | **1.0** |
|  | Cora | 2708 | 5429 | 0.15 | 5 | 1291 | 3.90 | 31 | 15410 | **0** ±0 | 0.81 ±0.08 |
| Citation | Citeseer | 3327 | 4732 | 0.09 | 6 | 1733 | 2.74 | 6 | 44700 | **0** ±0 | 1.42 ±0.15 |
|  | Pubmed | 19717 | 44338 | 0.03 | 8 | 4393 | 4.50 | - | >24h | **21** ±4 | 1.41 ±0.12 |
| ratio |  |  |  |  |  | 353.2 | 3.06 | - | - | **1.0** | **1.0** |
|  | Power Law Tree | 7956 | 7756 | 2.8 | - | 4519 | 10.47 | 33 | 4417 | **0** ±0 | 6.95 ±0.12 |
| Random | Small World | 7956 | 29716 | 10.7 | - | 5563 | 7.56 | 64 | 2021 | **29** ±2 | 7.73 ±0.12 |
|  | Holme and Kim | 7956 | 15712 | 5.7 | - | 8443 | 7.99 | **87** | 5317 | 456 ±26 | 5.64 ±0.12 |
| ratio |  |  |  |  |  | 38.20 | 1.28 | **0.38** | 578.5 | 1.0 | **1.0** |
|  | jean | 80 | 254 | 8 | 10 | 76 | 0.06 | **0** | 0.95 | **0** ±1 | 0.13 ±0.02 |
|  | anna | 138 | 493 | 5 | 11 | 87 | 0.08 | **0** | 3.23 | 0 ±0 | 0.17 ±0.03 |
|  | huck | 74 | 301 | 11 | 11 | 117 | 0.05 | **0** | 0.15 | **0** ±0 | 0.13 ±0.04 |
|  | david | 87 | 406 | 11 | 11 | - | - | **0** | 4.83 | 1 ±0 | 0.19 ±0.01 |
|  | homer | 561 | 1628 | 1 | 13 | 1628 | 1.09 | **0** | 274 | 1 ±0 | 0.29 ±0.02 |
|  | myciel5 | 47 | 236 | 22 | 6 | 35 | 0.04 | **0** | 0.20 | **0** ±0 | 0.12 ±0.01 |
|  | myciel6 | 95 | 755 | 17 | 7 | 94 | 4.33 | **0** | 0.79 | **0** ±0 | 0.21 ±0.02 |
|  | games120 | 120 | 638 | 9 | 9 | 301 | 0.07 | **0** | 0.93 | **0** ±1 | 0.08 ±0.01 |
|  | Mug88_1 | 88 | 146 | 4 | 3 | 146 | 0.33 | **0** | 0.12 | **0** ±0 | 0.01 ±0 |
| COLOR | 1-Insertions_4 | 67 | 232 | 10 | 2 | 42 | 0.05 | **0** | 0.16 | **0** ±0 | 0.07 ±0 |
|  | 2-Insertions_4 | 212 | 1621 | 7 | 4 | 360 | 0.09 | **1** | 255 | 2 ±0 | 0.08 ±0.01 |
|  | Queen5_5 | 25 | 160 | 53 | 5 | 37 | 0.03 | **0** | 0.13 | **0** ±0 | 0.05 ±0.01 |
|  | Queen6_6 | 36 | 290 | 46 | 6 | 290 | 0.38 | **0** | 4.93 | 4 ±0 | 0.05 ±0 |
|  | Queen7_7 | 49 | 476 | 40 | 7 | 126 | 0.04 | **10** | 36.9 | 11 ±1 | 0.06 ±0 |
|  | Queen8_8 | 64 | 728 | 36 | 8 | 188 | 0.05 | 8 | 61.3 | **7** ±2 | 0.06 ±0.01 |
|  | Queen9_9 | 81 | 1056 | 33 | 9 | 296 | 0.07 | **5** | 97.8 | 10 ±1 | 0.09 ±0.01 |
|  | Queen8_12 | 96 | 1368 | 30 | 12 | 260 | 0.10 | 10 | 139 | **7** ±0 | 0.09 ±0 |
|  | Queen11_11 | 121 | 3960 | 55 | 11 | 396 | 0.10 | 33 | 213 | **24** ±3 | 0.07 ±0.01 |
|  | Queen13_13 | 169 | 6656 | 47 | 13 | 728 | 0.20 | **42** | 401 | **42** ±4 | 0.08 ±0.01 |
| ratio |  |  |  |  |  | - | - | **1.0** | 736.17 | **1.0** | **1.0** |

□ **End of chapter.**

Chapter 5

# Routing Tree Construction

## 5.1 Introduction

In VLSI routing, wirelength (WL) and pathlength (PL) are the two fundamental metrics for the routing tree construction. Here, WL is directly related to power consumption, routing resource usage, cell delay and wire delay. Meanwhile, a long PL from the root (i.e., the source pin) implies high wire delay. However, optimizing either one of them does not necessarily benefit the other one.

The minimization of WL and PL has been investigated for a long history. Various approaches have been proposed to construct the routing tree by optimizing both WL and PL. These approaches can be roughly categorized into two types. The first type starts the construction from a tree that consists only of the source pin and iteratively adds the node into the tree with a newly-added edge. The two most influential and representative approaches of this type are the Prim-Dijkstra (PD) [10] construction and its improved version PD-II [9]. The second type [31, 95, 19] starts the construction from an initial topology with small WL such as FLUTE [24], and iteratively finds and reroutes the node whose PL is out of the bound. Among all the approaches above, PD-II [9] and SALT [19] are the two most prominent ones which demonstrate a superior trade-off between WL and PL compared with other state-of-the-art approaches. However, neither PD-II nor SALT always dominates the other one in terms of both WL and PL for all nets. Specifically, given a maximal WL constraint, although the PL of SALT is better than PD-II in most cases, there is still a considerable proportion on which PD-II is better, especially when the size of the net is large. The statistics shown in Table 5.2 demonstrate such a phenomenon, where PD-II outperforms SALT in 10.4% of huge nets when the WL degradation constraint is 0. Here the PL is measured by the normalized PL. Besides the uncertainty of deciding the best approach for any single case, there is another concern about the best parameter in PD-II and SALT. Both PD-II and SALT use a parameter to help decide whether one update should happen or not. For example, $\alpha$ in PD-II is included in the cost function to balance WL and PL. Correspondingly, $\epsilon$ in SALT is used to decide whether one node should be rerouted or not. Given any single case, i.e., a set of pins, the estimation of the

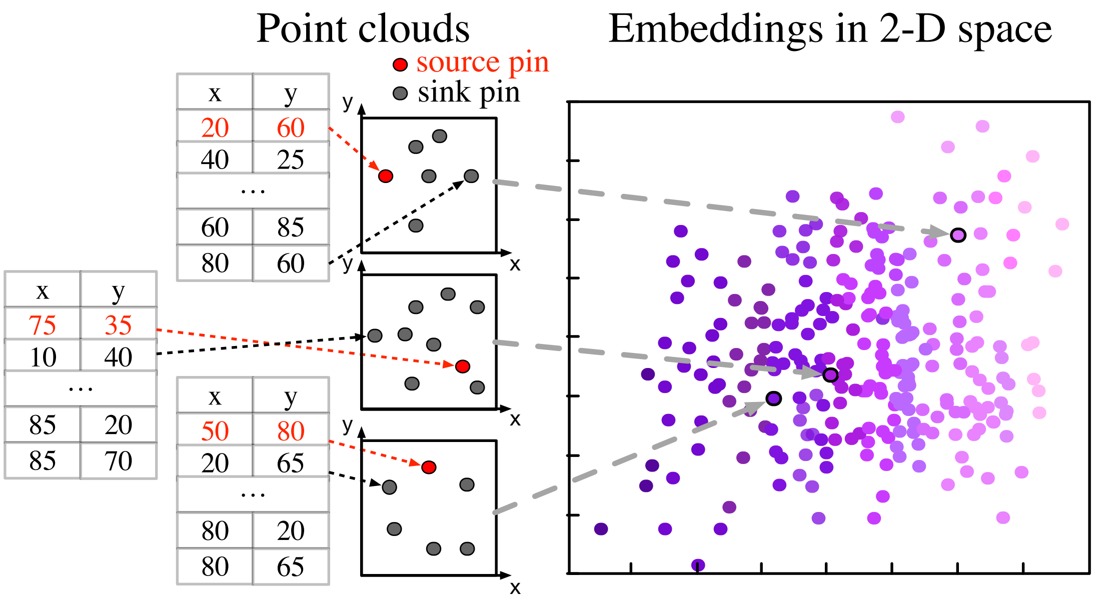


Figure 5.1: Cloud embeddings for tree construction, where point clouds are transformed into unified 2-D Euclidean space.

best parameter is still non-trivial and also an open problem to achieve the best PL given one WL constraint.

The recent overwhelming success of deep learning applications in various fields suggests that we can naturally cast the problems into the classification task (approach selection) and the regression task (parameter prediction). However, the set of 2-D points, which is the original input of the tree construction and also called the point cloud, usually varies in terms of the number of points, making it hard to be fed into a learning model. Therefore, we first need to transform a point cloud with unfixed size into a vector with a fixed size, where the vector is also called cloud embedding. The cloud embedding should be in a unified vector space with maximal representation capability such that the cloud embedding can help us to determine the best routing tree construction approach and predict the best parameter. One example of the point clouds for the routing tree construction and corresponding cloud embeddings are shown in Figure 5.1.

Although many works [88, 89, 111, 68] adapt the powerful deep learning-based methods for cloud embedding, none of them handles the point cloud specifically for the tree construction quite well due to some special properties in the tree construction. Through comprehensive analysis and consideration of these properties, we propose a deep learning-based model, TreeNet, to obtain the cloud embedding specifically for the tree construction. The obtained cloud embedding is then used as a representation to help select the best routing tree construction approach and predict the best parameter for the selected approach. Finally, we use the selected approach and corresponding parameter to construct

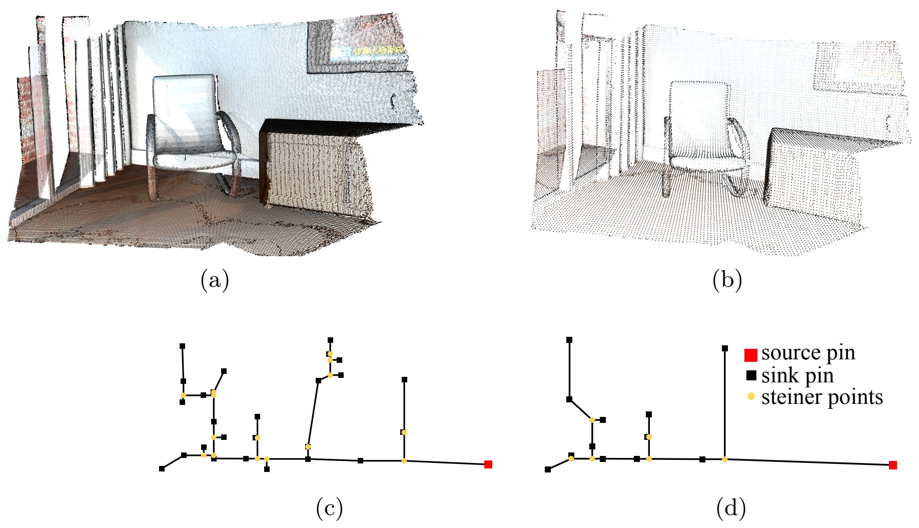


Figure 5.2: Examples of the down-sampling: (a) The general point cloud without the down-sampling; (b) The general point cloud with the down-sampling; (c) The constructed tree without the down-sampling; (d) The constructed tree with the down-sampling.

the routing tree. The main contributions are summarized as follows: 1) We formalize special properties of the point cloud for the routing tree construction; 2) We design TreeNet, a novel deep net architecture to obtain the cloud embedding for the tree construction; 3) We propose an adaptive flow for the routing tree construction, which uses the cloud embedding obtained by TreeNet to select the best approach and predict the best parameter; 4) Experiments on widely used benchmarks and demonstrate the effectiveness of our embedding representation, compared with all other deep learning models; 5) Experimental results show that our methods outperform other state-of-the-art routing tree construction methods in terms of both quality and runtime.

## 5.2 Preliminaries

### 5.2.1 Routing Tree

The routing tree is constructed by a set of terminals. Assume a input net $\boldsymbol{V} = \{v_0, \boldsymbol{V}_s\}$, $v_0$ is the source and $\boldsymbol{V}_s$ is the set of sinks. Let $G = \{\boldsymbol{V}, \boldsymbol{E}\}$ be the connected weighted routing graph. The edge weight of $G$ is the distance between vertices. A routing tree $T = \{\boldsymbol{V}', \boldsymbol{E}'\}$ is a spanning/Steiner tree that is constructed from $\boldsymbol{V}$ with $v_0$ as the root. A Steiner tree inserts new points from $\boldsymbol{V}$, i.e., $\boldsymbol{V}' \supseteq \boldsymbol{V}$, where the newly inserted points are called steiner points. The objective of the routing tree is to minimize both WL and PL. The WL metric is called the lightness or normalized WL, which is computed by the WL ratio with that of minimum spanning tree (MST), i.e., $lightness = \frac{w(T)}{w(MST(G))}$, where $w(\cdot)$ is the total weight. The PL metric is controversial and there are two widely used metrics. The first one is called the shallowness [18], which is computed by the maximal PL ratio with the shortest-path tree (SPT) among all vertices, i.e., $shallowness = \max\{\frac{d_T(v_0, v)}{d_G(v_0, v)} | v \in \boldsymbol{V}_s\}$. The second one is called the normalized path length [9], which is computed by the total PL normalized by the total shortest-path distance, i.e., $normPL = \frac{\sum_{v \in \boldsymbol{V}} d_T(r, v)}{\sum_{v \in \boldsymbol{V}} d_G(r, v)}$. Note that $d(\cdot)$ mentioned above denotes as the Manhattan distance.

### 5.2.2 Point Cloud

Point Cloud is defined as a set of scattered points in a 2D plane or 3D space. Therefore, the input of the routing tree construction, i.e., a set of 2-D points, can be modeled as a point cloud. Typical deep learning-based methods obtain the embedding of a general point cloud by a similar philosophy of the convolution layer. The convolution-like operation is usually composed of three procedures: `Sampling`, `Grouping` and `Encoding`. `Sampling` selects a set of centroids from the original point cloud. `Grouping` selects a set of neighbors for each centroid, which is like the local region constrained by a convolution kernel in the original convolution. `Encoding` is to encode the new centroid feature using the original one and the local feature aggregated from the neighbors of the centroid.

### 5.2.3 Problem Formulation

Given a set of 2-D pins and two routing tree construction algorithms, SALT [18] and PD-II [9], our objective is to obtain the embedding of the given point cloud by TreeNet such that 1) the embedding can be used to select the best algorithms for the given point cloud; 2) the embedding can be used to estimate the best parameter $\epsilon$ of SALT for the given point cloud; 3) the embedding can be used to estimate the best parameter $\alpha$ of PD-II for the given point cloud.

## 5.3 The Proposed Approaches

In this section, we first formalize a set of special properties of the point cloud specifically for the routing tree construction. Then we propose TreeNet for cloud embedding by considering these properties. Finally, an adaptive workflow for the routing tree construction based on the cloud embedding is introduced.

### 5.3.1 Property Analysis

Given the input net $\boldsymbol{V} = \{v_0, \boldsymbol{V}_s\}$. Let $f : \boldsymbol{V} \rightarrow T$ be the function for routing tree construction and $T$ is the target routing tree. Here, we say $T = T'$ if and only if $T$

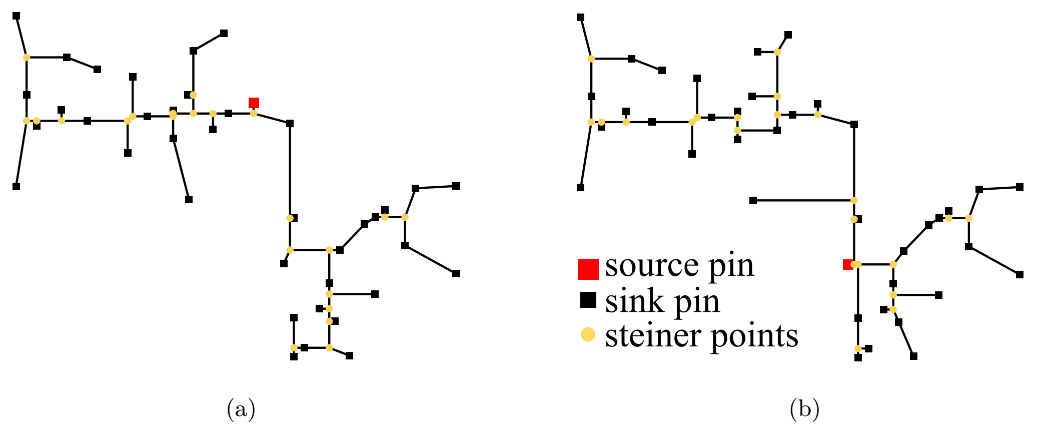


Figure 5.3: Examples of the routing trees with the same node distribution but different root (highlighted by red).

and $T'$ have the same node coordinates and the same topology. Ideally, a powerful neural network maps nets with different (same) routing trees to embeddings which are as different (similar) as possible. Therefore, we may design the cloud embedding method by learning the behavior of $f$.

**Property 1.** *Let $d : \boldsymbol{V} \rightarrow \boldsymbol{V'}$ be a function for down-sampling, where $\boldsymbol{V'}$ is a proper subset of $\boldsymbol{V}$. $f(\boldsymbol{V}) \neq f(d(\boldsymbol{V}))$ holds if there exists $v \in \boldsymbol{V} - d(\boldsymbol{V})$ so that $v$ is not the steiner point in $f(d(\boldsymbol{V}))$ .*

Property 1 points the deficiency of down-sampling. Actually, the inequality holds for most cases even without the condition. One down-sampling example is shown in Figure 5.2. With down-sampling, the skeleton of the general point cloud is still easy to classify as shown in Figure 5.2(b). However, given the point cloud for the routing tree construction, the routing tree with down-sampling (Figure 5.2(d)) is totally different from the one without down-sampling (Figure 5.2(c)). Here,

**Property 2.** *Let $\boldsymbol{V}_s^p$ be the permutation of the sink set $\boldsymbol{V}_s$. $f(\{v_0, \boldsymbol{V}_s^p\}) = f(\{v_0, \boldsymbol{V}_s\})$ holds for any $\boldsymbol{V} = \{v_0, \boldsymbol{V}_s\}$.*

**Property 3.** *Let $\boldsymbol{V}^p$ be the permutation of the input net $\boldsymbol{V}$. $f(\boldsymbol{V}^p) \neq f(\boldsymbol{V})$ holds if the source in $\boldsymbol{V}^p$ is different from the source in $\boldsymbol{V}$.*

Property 2 shows the permutation invariance of the point cloud and Property 3 states the sensitivity of the root. One example is shown in Figure 5.3, where the space distributions of point locations are totally the same while only the root is assigned differently. A typical method may regard the two point clouds as a similar pair while they actually represent completely different trees.

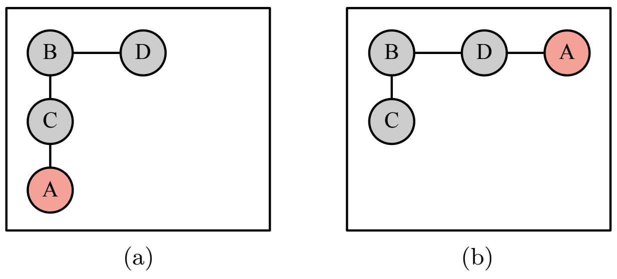


Figure 5.4: Examples of the node with the same coordinates and local neighbors but different parent-child relationships. Here root is highlighted in red.

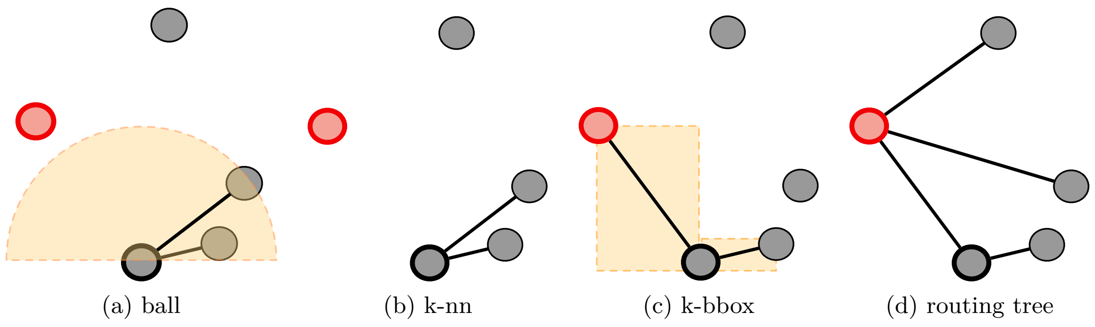


Figure 5.5: Comparison among ball query (a) k-nn (b) and k-bbox (c) grouping methods ($k = 2$ in this example). The orange regions represent the query ball in (a) and bounding boxes in (c). The centroid is highlighted by black and the root is by red.

**Property 4.** *For any sink set* $\boldsymbol{V}_s$ *with* $|\boldsymbol{V}_s| > 1$*, there exists two different pins,* $v_0$ *and* $v_0'$ *in the 2-D plane so that* $f(\{v_0, \boldsymbol{V}_s\}) \neq f(\{v_0', \boldsymbol{V}_s\})$*. Moreover, the inequality holds when we only consider the topology.*

As an extension of Property 3, Property 4 states the possible inequality even when the sink set $\boldsymbol{V}_s$ is not changed. It not only demonstrates the sensitivity of the root, but also shows the deficiency of only considering local information, i.e, information stored in $\boldsymbol{V}_s$. One example is shown in Figure 5.4, where the node $B$ in both Figure 5.4(a) and Figure 5.4(b) have the same coordinates and local neighbors $(C, D)$ but the parent-child relationships $B - C$ and $B - D$ are clearly different.

**Property 5.** *Let* $G_{ball}$*,* $G_{knn}$ *and* $G_{bbox}$ *be the graph constructed from* $\boldsymbol{V}$ *by ball query,* $k$ *nearest neighbor and bounding box respectively. The minimum spanning tree,* $T$ *may not be the subgraph of* $G_{ball}$ *or* $G_{nn}$*, but always the subgraph of* $G_{bbox}$*.*

Property 5 states that $G_{bbox}$ [9] is more likely to capture the structure of the routing tree, compared to $G_{ball}$ [89] and $G_{knn}$ [111, 68]. $G_{bbox}$ is the graph whose nodes are connected with their *bbox-neighbors*. We call the node $u_j$ as the *bbox-neighbor* of $u_i$ if there is no other node in the smallest bounding box containing $u_i$ and $u_j$. One example of the comparison is shown in Figure 5.5, $G_{ball}$ and $G_{knn}$ fail to find the correct neighbors of the selected centroid.

### 5.3.2 Cloud embedding by TreeNet

Considering all these properties discussed above, we propose a specialized model, TreeNet, to obtain the embedding of the point cloud for the routing tree construction. Basically, TreeNet is a hierarchical model and composed of a number of convolution-like operations. We refer to this operation as TreeConv. The comparison between TreeConv and other methods [88, 89, 68, 111] are summarized in Table 5.1.

Our TreeConv (see Figure 5.6) leverages the local correlation information and the root information with two key procedures: `Grouping` and `Encoding`. Different from some typical works, TreeConv omits the `Sampling` phase considering Property 1. Therefore, each node is selected as the centroid. Given a point cloud $\boldsymbol{H} \in \mathbb{R}^{N\times D}$, where $N$ is the number of points and $D$ is the dimension of each point, our `Grouping` selects $k$ neighbors for each centroid $u_i$ to represent the local point cloud structure based on $G_{bbox}$ [9]. One example is shown in Figure 5.5(c). We first select $k$ nearest *bbox-neighbors* of $u_i$ as the neighbors. If the number of *bbox-neighbors* for $u_i$ is less than $k$, we then select other nearest nodes to fill up $k$ neighbors. Therefore, `Grouping` returns a list of neighbors $\boldsymbol{E}_i \in \mathbb{R}^k$ for each centroid $u_i$. After `Grouping`, our `Encoding` outputs a new feature $\boldsymbol{v}'_i \in \mathbb{R}^{D'}$ for each node $u_i$ such that the new point cloud $H' = \{\boldsymbol{v}'_0, ..., \boldsymbol{v}'_{N-1}\} \in \mathbb{R}^{N\times D'}$. For each element $v'_{ic}$ in $\boldsymbol{v}'_i$, our `Encoding` leverages the global position information [88], the ”local” neighborhood information [111, 68], and the root information considering Property 4. The computation can be formulated as:

$$v'_{ic} = \max_{j\in E_i} \sigma(\boldsymbol{\theta}_c \cdot \texttt{CONCAT}(\boldsymbol{v}_i - \boldsymbol{v}_j, \boldsymbol{v}_i - \boldsymbol{v}_r, \boldsymbol{v}_i)), \tag{5.1}$$

where $\boldsymbol{v}_r$ is the input feature of the root, $\sigma$ is the `LeakyReLU` activation function and $\boldsymbol{\theta}_c \in \mathbb{R}^{3D}$ is the trainable weight of $c_{th}$ filter. Finally, the new feature $\boldsymbol{v}'_i$ is processed by a Squeeze-and-Excitation (SE) block [46] for exploiting channel dependencies.

The network architecture used for the cloud embedding is shown in Figure 5.7. The input of the first layer is the point cloud $\boldsymbol{H}_0 \in \mathbb{R}^{N\times 2}$, in which each node has a 2-D normalized coordinate feature. The normalization of each node $u_i$ is based on the root $u_r$ (Property 3, Property 4) and can be formulated as:

$$\tilde{\boldsymbol{v}}_i = \frac{\boldsymbol{v}_i - \boldsymbol{v}_r}{d_{max}}, \tag{5.2}$$

where $\boldsymbol{v}_i$ is the original 2-D coordinate feature of node $u_i$ and $\boldsymbol{v}_r$ is the feature of the root. $d_{max}$ is the maximal distance between the root and any other nodes.

We stack four TreeConvs and include shortcut connections from each layer to the output to extract multi-scale features which are finally concatenated. Then, two permutation-invariant operations (Property 2), max pooling and average pooling, are used to get the

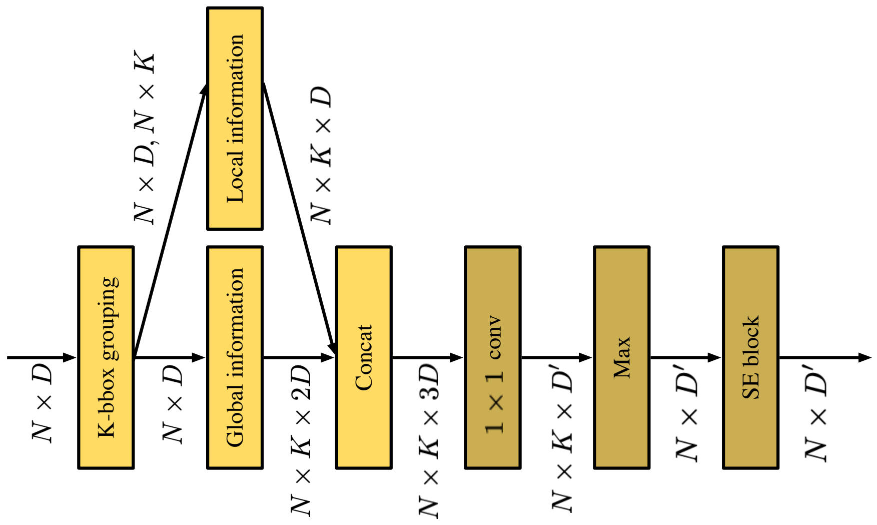


Figure 5.6: Illustration of TreeConv. Brighter blocks indicate `Grouping` and darker blocks indicate `Encoding`.

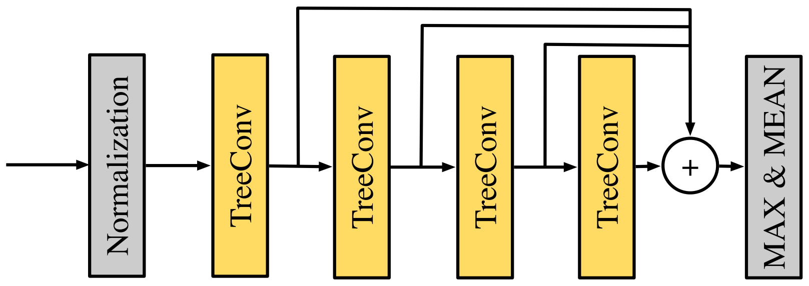


Figure 5.7: Illustration of TreeNet Architecture for the cloud embedding.

cloud embedding, which can be formulated as

$$\begin{aligned}\boldsymbol{H}_c = \texttt{CONCAT}(&\max(\texttt{CONCAT}(\tilde{\boldsymbol{H}}_1, \tilde{\boldsymbol{H}}_2, \tilde{\boldsymbol{H}}_3, \tilde{\boldsymbol{H}}_4)),\\ &\mathrm{mean}(\texttt{CONCAT}(\tilde{\boldsymbol{H}}_1, \tilde{\boldsymbol{H}}_2, \tilde{\boldsymbol{H}}_3, \tilde{\boldsymbol{H}}_4))),\end{aligned} \tag{5.3}$$

where $\boldsymbol{H}_c \in \mathbb{R}^{2\times(D_1+D_2+D_3+D_4)}$ is the final cloud embedding and $\tilde{\boldsymbol{H}}_i \in \mathbb{R}^{N\times D_i}$ is the scaled output of $i_{th}$ TreeConv.

### 5.3.3 Routing Tree Construction based on Point Cloud Embedding

Given the cloud embedding $\boldsymbol{H}_c \in \mathbb{R}^D$ obtained by TreeNet, we can cast the algorithm selection and the parameter prediction problem into classification and regression problem, respectively. The workflow of our framework is shown in Figure 5.8.

Firstly, we use the obtained cloud embedding to determine whether the SALT algorithm is at least as good as the PD-II algorithm, where ”good” means that the best PL

Table 5.1: Comparison to existing methods.

| | `Sampling` | `Grouping` | `Encoding` |
|---|---|---|---|
| PointNet [88] | - | - | $v'_{ic} = \sigma(\boldsymbol{\theta}_c \boldsymbol{v}_i)$ |
| PointNet++ [89] | Fathest Point Sampling (FPS) | ball query’ s local neighborhood | $v'_{ic} = \max_{j \in E_i} \sigma(\boldsymbol{\theta}_c \boldsymbol{v}_j)$ |
| PointCNN [68] | Random/FPS | k nearest neighbor | $\boldsymbol{v}'_i = \texttt{Conv}(X \times \boldsymbol{\theta}(\boldsymbol{v}_i - \boldsymbol{v}_j))$ |
| DGCNN [111] | - | k nearest neighbor | $v'_{ic} = \max_{j \in E_i} \sigma(\boldsymbol{\theta}_c \cdot \texttt{CONCAT}(\boldsymbol{v}_i - \boldsymbol{v}_j, \boldsymbol{v}_i))$, |
| Our work | - | k bounding box neighbor | $v'_{ic} = \max_{j \in E_i} \sigma(\boldsymbol{\theta}_c \cdot \texttt{CONCAT}(\boldsymbol{v}_i - \boldsymbol{v}_j, \boldsymbol{v}_i - \boldsymbol{v}_r, \boldsymbol{v}_i))$, |

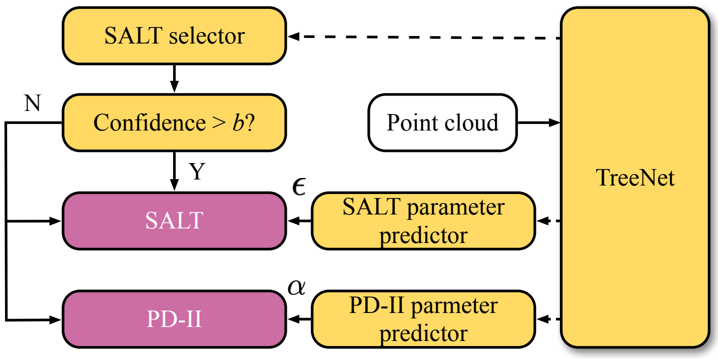


Figure 5.8: The workflow of our framework. Dotted arrows represent that TreeNet generates cloud embeddings and use them to select the algorithm or to predict parameters. The yellow blocks are executed in our framework while the purple blocks are executed by the selected algorithms.

of SALT is at least the same as that of PD-II under the given WL constraint and the PL metric. Therefore, the problem can be regarded as a 2-class classification problem, where one class indicates that the result of SALT is at least as good as PD-II while another one indicates its opposite. Given the cloud embedding $\boldsymbol{H}_c \in \mathbb{R}^D$, the 2-class classifier is implemented by three fully connected layers followed by a softmax layer and can be formulated as:

$$\boldsymbol{y} = \texttt{softmax}(\boldsymbol{W}_3 \sigma(\boldsymbol{W}_2 \sigma(\boldsymbol{W}_1 \boldsymbol{H}_c + \boldsymbol{b}_1) + \boldsymbol{b}_2)), \tag{5.4}$$

where $\sigma$ is the `LeakyReLU` activation function, $\boldsymbol{W}_i$ and $\boldsymbol{b}_i$ are the weight matrix and bias vector, respectively. $\boldsymbol{y} \in \mathbb{R}^2$ is the final classification confidence. Since each wrong selection directly harms the quality of the result, we raise the bar to select SALT algorithm: We directly use SALT algorithm to construct the routing tree only when the confidence of ”SALT is at least as good as PD-II” is larger than a specified confidence bar $b$. Otherwise, we will use both SALT and PD-II to construct the routing tree and use the better result.

After algorithm selection, corresponding parameter is predicted by the obtained cloud embedding and guides the parameter selection. The regression target of the parameter prediction is the ”best” parameter, where ”best” means the result using such parameter achieves the best PL under the WL degradation constraint. We follow a similar approach used in the age prediction [97]. Take the parameter prediction of SALT for example. We set 20 valid parameter $\epsilon_i, i \in \{1, ..., 20\}$ candidates for SALT and each valid parameter is

**Algorithm 5** `ParameterGuidanceConstruction`

```
Input: $\epsilon \to$ Predicted parameter;
Input: $f \to$ Routing tree construction algorithm;
1: $results \leftarrow$ Run $f$ using the parameters in $(\epsilon - \sigma, \epsilon + \sigma) \cup S$;
2: if no result in $results$ satisfies the WL constraint then
3:     return ParameterGuidanceConstruction($\epsilon + 2\sigma, f$);
4: else
5:     return the result with the best PL in $results$ satisfying the WL constraint under the PL metric;
6: end if
```

treated as a separate class. Therefore, the structure for the parameter prediction is similar with the one for the algorithm selection formulated in Equation (5.4) with $\boldsymbol{y} \in \mathbb{R}^{20}$. Given the output $\boldsymbol{y}$, the predictied parameter $\epsilon$ is calculated by an element-wise summation and can be formulated as:

$$\epsilon = \sum_{i=1}^{20} \epsilon_i \cdot y_i. \tag{5.5}$$

Given the predicted parameter, the guidance follows a simple but effective heuristic rule specified in Algorithm 5. Generally speaking, the selected approach constructs the routing tree using the predicted parameter and other parameters in a set (line 1), where $\sigma$ and $S$ are hyper-parameters. $S$ defines an initial set of parameters that achieve almost the best PL while also almost the worst WL. If the results of all tested parameters fail to satisfy the WL constraint, the range is widened along the direction which decreases the WL (line 2-3); Otherwise, we directly use the best parameter among these tested candidates (line 4-5).

## 5.4 Experimental Results

We implement TreeNet in Python with PyTorch. Other models (PointNet [88], PointNet++ [89], PointCNN [68], DGCNN [111] ) are also implemented. The results of SALT [19] are generated by the source code and the results of PD-II [9] are provided by the authors. The experiments are conducted on the widely used benchmarks of ICCAD 2015 Contest [52]. All the two-pin and three-pin nets are removed. All the experiments are conducted on an Intel Core 2.9 GHz Linux machine with one NVIDIA TITAN Xp GPUs.

**Data Preparation.** We first assign labels to each net in the benchmarks according to the routing tree results of SALT and PD-II. Specifically, given the constraint of WL degradation percentage with respect to MST WL and the PL metric type (the shallowness metric and the normalized PL metric), the net is assigned three labels: 1) *best algorithm*; 2) *best* $\epsilon$ for SALT and 3) *best* $\alpha$ for PD-II. The *best algorithm* is labeled as one of $\{SALT, PDII\}$. Here, the net is labeled as $SALT$ ($PDII$) when the routing tree by SALT (PD-II) achieves the best PL under the chosen PL metric and the WL degradation constraint. Especially, if both SALT and PD-II construct the same routing tree, the *best*

Table 5.2: ICCAD 2015 Benchmark Label Statistics (part)

| WL deg. | PL metric | Label | Small | Med. | Large | Huge | Total |
|---|---|---|---|---|---|---|---|
| 0 | Nor. PL | SALT | 99.9% | 98.8% | 93.5% | 89.6% | 1273012 (98.3%) |
| | | PD-II | 0.1% | 1.2% | 6.5% | 10.4% | 21529 (1.7%) |
| 0 | Shallow. | SALT | 99.9% | 99.1% | 95.6% | 93.1% | 1279428 (98.8%) |
| | | PD-II | 0.1% | 0.9% | 4.4% | 6.9% | 15113 (1.2%) |
| 10% | Shallow. | SALT | 99.8% | 97.0% | 93.6% | 91.4% | 1269095 (98.0%) |
| | | PD-II | 0.2% | 3.0% | 6.4% | 8.6% | 25446 (2.0%) |

*algorithm* is labeled as *SALT*. The *best* $\epsilon$ for SALT is defined by:

$$\bar{y_i} = \begin{cases} \frac{1}{k}, & \text{if } \epsilon_i \text{ is one of the ``best'' candidates;} \\ 0, & \text{otherwise.} \end{cases} \tag{5.6}$$

where $k$ is the number of those "best" candidates, $i \in \{1, ..., 20\}$ and $\epsilon_i = 0.05 \times 1.5^i$ as defined in [19]. Similarly, the best $\alpha$ for PD-II also follows Equation (5.6) with $i \in \{1, ..., 19\}$ and $\alpha_i = 0.05 \cdot i$ as defined in [9]. Note that, increasing the number of data points also improves the quality of our model since the noise label is reduced. Given such label rules, the label distribution of nets in the benchmarks are clearly different based on the WL degradation constraint and PL metric type. A part of the *best algorithm* label statistics for the benchmarks is shown in Table 5.2, where "WL deg." denotes the percentages of permissible WL degradation with respect to MST WL.

**Architecture.** The output point dimension of four TreeConvs is $(32, 32, 64, 128)$ and such the final embedding dimension is $(32 + 32 + 64 + 128) \times 2 = 512$. Then, three fully connected layers $(128, 64, c)$ are used, where $c$ is the number of classes in the label. Dropout is applied and the keep probability is set to 0.5. The number of selected neighbors $k$ is set to 3, which is the maximal neighbor number for a 4-point net. The confidence bar $b$ is set as 0.99. The range $\sigma$ in Algorithm 5 is set to 1 for SALT and -1 for PD-II. The set $S$ is set to {1,2} in SALT and {18,19} in PD-II.

**Training & Testing.** During training, we minimize the binary cross-entropy loss for the algorithm selection and the soft cross-entropy loss for the parameter prediction. We use SGD with an initial learning rate 0.001 and momentum 0.9, and the learning rate is reduced by 30% every 20 epochs. We follow the idea of K-fold cross-validation to set up the test. Specifically, each design in the benchmark is tested using the model trained by other designs following the same configurations.

### 5.4.1 Comparison with other DL models

In this section, we compare TreeNet with other baseline models for the algorithm selection task. Due to the page limit, the result of the parameter prediction task is not shown since it is also formulated as a classification task. The WL degradation is set as 5% and the PL

Table 5.3: Algorithm selection results

| Method | Accuracy | Precision | Recall* |
|---|---|---|---|
| PointNet [88] | 54.13 | 53.95 | 1.91 |
| PointNet++ [89] | 81.31 | 82.50 | 2.65 |
| PointCNN [68] | 62.18 | 64.24 | 1.16 |
| DGCNN [111] | 92.24 | 94.62 | 11.84 |
| TreeNet w.o. Nor | 87.22 | 88.62 | 15.69 |
| TreeNet w.o. global | 92.40 | 94.63 | 25.53 |
| TreeNet w. knn | 92.58 | 94.79 | 26.76 |
| TreeNet | **94.09** | **95.38** | **50.74** |

metric is set as the normalized PL. Formally, we mark the $SALT$ label as positive and the $PDII$ as negative.

The result for the algorithm selection is shown in Section 5.4, where "Accuracy" and "Precision" are defined as usual. "Recall*" is slightly different and defined as the fraction of the total amount of positive instances that were also predicted as positive with the confidence larger than the bar $b$. Therefore, the updated recall is directly related to the runtime performance. We use SALT to construct the routing tree for the predicted positive instances, and use both SALT and PD-II for the predicted negative instances. We compare four state-of-the-art models with our TreeNet and three variations: 1) Remove the root-sensitive normalization and use the original normalization (property 1); 2) Remove the root-related global information in `Encoding` phase (property 3, 4); 3) Use k-nn grouping method instead of k-bbox (property 5). According to Section 5.4, we can see that our TreeNet outperforms other state-of-the-art models on all three metrics. Besides, the comparison with other variations demonstrates the effectiveness of our considerations on the properties of the point cloud for the tree construction.

### 5.4.2 Comparison with routing tree constructors

In this section, we compare our adaptive workflow with SALT and PD-II in terms of both effectiveness and efficiency.

**Effectiveness:** We follow the same result comparison way in [9]: We first select different WL degradation constraints (0%, 5%, 10%, 15%, 20%) and then find the best shallowness and the normalized PL. Each entry in the table is the averaged shallowness (see Table 5.4) and the averaged normalized PL (see Table 5.5) across all test nets. Especially, SALT* executes SALT in a binary search manner, which results in better efficiency but may harm the quality. We also measure the improvement compared with SALT (**Imp. (%)**) and SALT* (**Imp.* (%)**). The improvement is calculated by the percentage improvement after subtracting the lower bound 1. For example, a reduction from 1.10 to 1.09 results in an improvement of 10%, i.e., $(1-(1.09-1.0)/(1.10-1.0))\cdot 100\%$. As Table 5.4 and Table 5.5 show, our adaptive workflow outperforms SALT and PD-II for all classes of nets under all WL constraints and PL metrics. In general, the overall improvement over SALT ranges from 1.97% to 12.16%, depending on the selected WL constraint and PL metric. The improvement over SALT* is more significant, ranging from 2.89% to 19.11%.

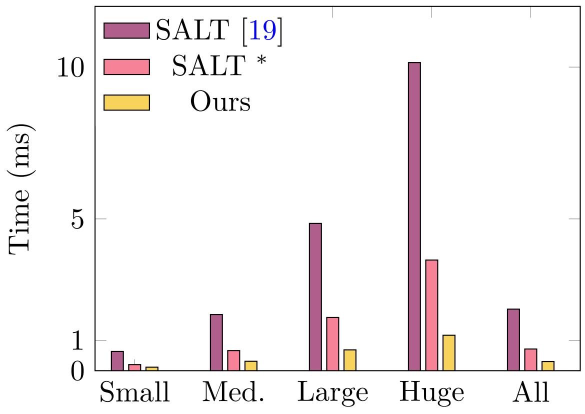


Figure 5.9: Runtime comparison with SALT and SALT*.

**Efficiency:** Since the source code of PD-II is not provided, we only compare the runtime performance of our adaptive workflow with SALT [19]. The runtime of our framework is composed of three parts: 1) The inference time of TreeNet; 2) The execution time of SALT on the input net; 3) The execution time of PD-II on the input net when the algorithm selector does not select SALT as the only tree constructor. Especially, the execution time of PD-II is estimated form the runtime analysis in [9], where PD-II costs 361s and SALT costs 2762s. Therefore, we estimate the runtime of PD-II by SALT with a runtime ratio $361/2762 = 0.1307$. Figure 5.9 shows the average runtime comparison with SALT and SALT*, where adaptive workflow is more efficient than both of them on any size scale. We further profile the runtime of the framework, as shown in Figure 5.10. The inference time of TreeNet only occupies 24.39% of the total runtime.

Table 5.4: Results on shallowness

| \|V\| | Method | WL deg. | | | | |
|---|---|---|---|---|---|---|
| | | 0% | 5% | 10% | 15% | 20% |
| Small | PD-II | 1.0606 | 1.0369 | 1.0240 | 1.0161 | 1.0114 |
| | SALT | 1.0462 | 1.0216 | 1.0078 | 1.0022 | 1.0006 |
| | SALT* | 1.0462 | 1.0216 | 1.0079 | 1.0023 | 1.0006 |
| | Ours | 1.0461 | 1.0210 | 1.0074 | 1.0021 | 1.0005 |
| | **Imp. (%)** | **0.28** | **2.62** | **4.40** | **5.42** | **8.25** |
| | **Imp.* (%)** | **0.32** | **3.04** | **5.14** | **6.75** | **9.94** |
| Med. | PD-II | 1.3849 | 1.2518 | 1.1688 | 1.1176 | 1.0851 |
| | SALT | 1.3456 | 1.1775 | 1.0838 | 1.0391 | 1.0181 |
| | SALT* | 1.3463 | 1.1815 | 1.0868 | 1.0410 | 1.0192 |
| | Ours | 1.3435 | 1.1689 | 1.0790 | 1.0370 | 1.0172 |
| | **Imp. (%)** | **0.62** | **4.85** | **5.72** | **5.57** | **5.41** |
| | **Imp.* (%)** | **0.80** | **6.95** | **8.98** | **9.92** | **10.41** |
| Large | PD-II | 1.9093 | 1.5584 | 1.3595 | 1.2473 | 1.1805 |
| | SALT | 1.7976 | 1.3549 | 1.1568 | 1.0727 | 1.0358 |
| | SALT* | 1.8083 | 1.3689 | 1.1648 | 1.0771 | 1.0382 |
| | Ours | 1.7755 | 1.3339 | 1.1481 | 1.0690 | 1.0341 |
| | **Imp. (%)** | **2.77** | **5.91** | **5.53** | **5.11** | **4.78** |
| | **Imp.* (%)** | **4.06** | **9.50** | **10.12** | **10.52** | **10.77** |
| Huge | PD-II | 2.1660 | 1.7169 | 1.4771 | 1.3438 | 1.2603 |
| | SALT | 2.0111 | 1.4398 | 1.2083 | 1.0987 | 1.0466 |
| | SALT* | 2.0291 | 1.4567 | 1.2183 | 1.1039 | 1.0489 |
| | Ours | 1.9793 | 1.4152 | 1.1975 | 1.0941 | 1.0444 |
| | **Imp. (%)** | **3.15** | **5.61** | **5.17** | **4.69** | **4.64** |
| | **Imp.* (%)** | **4.85** | **9.09** | **9.50** | **9.47** | **9.20** |
| All | PD-II | 1.2921 | 1.1822 | 1.1193 | 1.0827 | 1.0604 |
| | SALT | 1.2531 | 1.1175 | 1.0524 | 1.0236 | 1.0110 |
| | SALT* | 1.2555 | 1.1210 | 1.0546 | 1.0248 | 1.0117 |
| | Ours | 1.2481 | 1.1114 | 1.0495 | 1.0223 | 1.0104 |
| | **Imp. (%)** | **1.97** | **5.18** | **5.43** | **5.21** | **5.08** |
| | **Imp.* (%)** | **2.89** | **7.98** | **9.23** | **9.95** | **10.38** |

Table 5.5: Results on normalized PL

| \|V\| | Method | WL deg. | | | | |
|---|---|---|---|---|---|---|
| | | 0% | 5% | 10% | 15% | 20% |
| Small | PD-II | 1.0156 | 1.0099 | 1.0065 | 1.0044 | 1.0031 |
| | SALT | 1.0113 | 1.0055 | 1.0020 | 1.0006 | 1.0002 |
| | SALT* | 1.0113 | 1.0055 | 1.0020 | 1.0006 | 1.0002 |
| | Ours | 1.0112 | 1.0053 | 1.0019 | 1.0005 | 1.0001 |
| | **Imp. (%)** | **0.25** | **2.86** | **4.88** | **6.57** | **10.55** |
| | **Imp.* (%)** | **0.29** | **3.38** | **5.83** | **8.29** | **12.75** |
| Med. | PD-II | 1.0897 | 1.0579 | 1.0373 | 1.0248 | 1.0170 |
| | SALT | 1.0778 | 1.0428 | 1.0204 | 1.0096 | 1.0044 |
| | SALT* | 1.0780 | 1.0440 | 1.0214 | 1.0102 | 1.0048 |
| | Ours | 1.0773 | 1.0396 | 1.0185 | 1.0086 | 1.0040 |
| | **Imp. (%)** | **0.63** | **7.35** | **9.45** | **10.01** | **10.00** |
| | **Imp.* (%)** | **0.82** | **9.90** | **13.70** | **15.74** | **16.65** |
| Large | PD-II | 1.1968 | 1.1146 | 1.0671 | 1.0413 | 1.0267 |
| | SALT | 1.1665 | 1.0815 | 1.0365 | 1.0172 | 1.0086 |
| | SALT* | 1.1690 | 1.0854 | 1.0390 | 1.0187 | 1.0095 |
| | Ours | 1.1616 | 1.0726 | 1.0318 | 1.0150 | 1.0076 |
| | **Imp. (%)** | **2.95** | **10.92** | **12.81** | **12.91** | **12.49** |
| | **Imp.* (%)** | **4.35** | **15.02** | **18.29** | **19.70** | **20.35** |
| Huge | PD-II | 1.2472 | 1.1415 | 1.0830 | 1.0513 | 1.0328 |
| | SALT | 1.2120 | 1.1054 | 1.0489 | 1.0224 | 1.0105 |
| | SALT* | 1.2160 | 1.1106 | 1.0522 | 1.0242 | 1.0112 |
| | Ours | 1.2045 | 1.0917 | 1.0413 | 1.0190 | 1.0088 |
| | **Imp. (%)** | **3.54** | **13.03** | **15.54** | **15.54** | **16.25** |
| | **Imp.* (%)** | **5.31** | **17.12** | **20.97** | **21.52** | **21.87** |
| All | PD-II | 1.0658 | 1.0398 | 1.0244 | 1.0157 | 1.0105 |
| | SALT | 1.0550 | 1.0278 | 1.0125 | 1.0056 | 1.0026 |
| | SALT* | 1.0555 | 1.0289 | 1.0132 | 1.0061 | 1.0029 |
| | Ours | 1.0538 | 1.0253 | 1.0111 | 1.0050 | 1.0023 |
| | **Imp. (%)** | **2.05** | **9.17** | **11.35** | **11.94** | **12.16** |
| | **Imp.* (%)** | **3.01** | **12.43** | **16.04** | **17.98** | **19.11** |

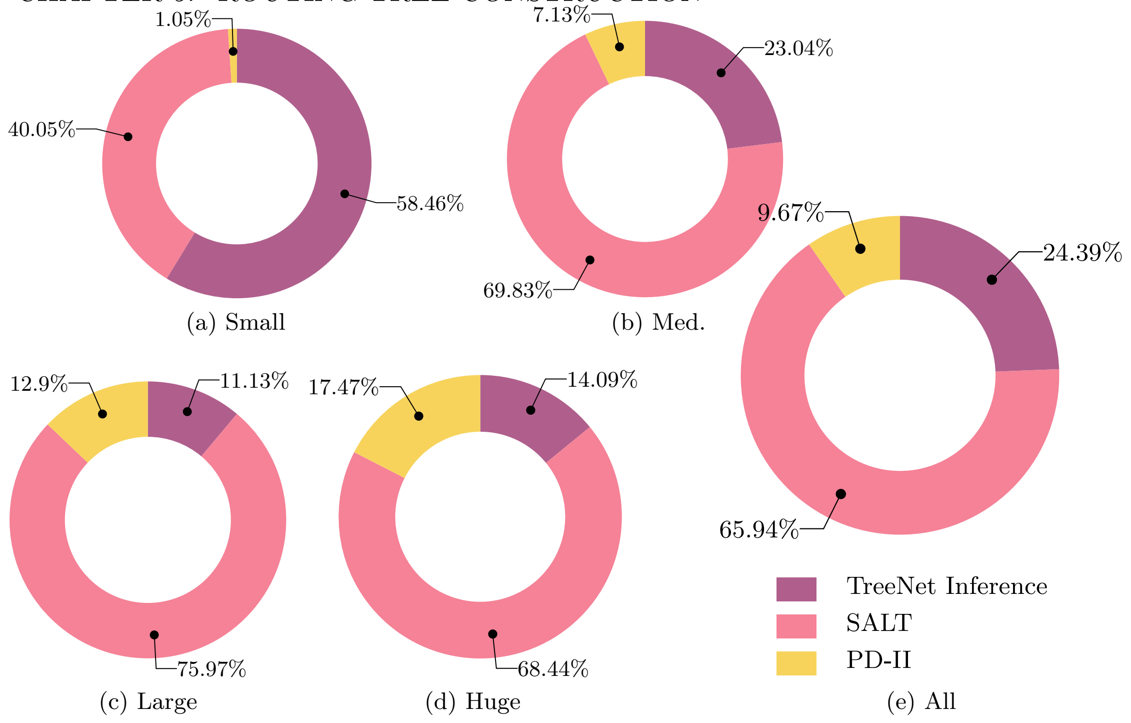


Figure 5.10: Runtime breakdown of the routing tree construction framework.

---

□ **End of chapter.**

# Chapter 6

# Appendix

## .1 Graph Terminology

Here we list the following graph theoretic terms encountered in our work:

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ be graphs on vertex set $\mathcal{V}$ and $\mathcal{V}'$, we define

- *isomorphism*: we say that a bijection $\pi : \mathcal{V} \to \mathcal{V}'$ is an *isomorphism* if any two vertices $u, v \in \mathcal{V}$ are adjacent in $\mathcal{G}$ if and only if $\pi(u), \pi(v) \in \mathcal{V}'$ are adjacent in $\mathcal{G}'$, i.e., $\{u, v\} \in \mathcal{E}$ iff $\{\pi(u), \pi(v)\} \in \mathcal{E}'$.
- *isomorphic nodes*: If there exists the isomorphism between $\mathcal{G}$ and $\mathcal{G}'$, we say that $\mathcal{G}$ and $\mathcal{G}'$ are *isomorphic*.
- *automorphism*: When $\pi$ is an isomorphism of a vertex set onto itself, i.e., $\mathcal{V} = \mathcal{V}'$, $\pi$ is called an *automorphism* of $\mathcal{G}$.
- *topologically equivalent*: We say that the node pair $\{u, v\}$ is *topologically equivalent* if there is an automorphism mapping one to the other, i.e., $v = \pi(u)$.
- *equivalent*: $\{u, v\}$ is *equivalent* if it is topologically equivalent by $\pi$ and $\boldsymbol{x}_w = \boldsymbol{x}_{\pi(w)}$ holds for every $w \in \mathcal{V}$, where $\boldsymbol{x}_w$ is the node attribute of node $w$.
- *$r$-local isomorphism*: A bijection $\pi_r$ is an *$r$-local isomorphism* that maps $u$ to $v$ if $\pi_r$ is an isomorphism that maps $\mathcal{B}_{\mathcal{G}}(u, r)$ to $\mathcal{B}_{\mathcal{G}}(v, r)$.

## .2 Proofs in Chapter 4

### .2.1 Proof of Property 1

We first recall the property.

**Property 1.** *All AC-GNNs cannot discriminate any equivalent node pair.*

*Proof.* Let $\pi$ be the automorphism mapping $u$ to $v$, here, we propose a stronger property:

**Property 4.** *Given an AC-GNN and an equivalent node pair $\{u, v\}$ by $\pi$, $\boldsymbol{h}_w^i = \boldsymbol{h}_{\pi(w)}^i$ holds for any iteration $i$ and any node $w \in \mathcal{V}$.*

This apparently holds for $i = 0$ since $\boldsymbol{x}_w = \boldsymbol{x}_{\pi(w)}, \forall w \in \mathcal{V}$. Suppose this holds for iteration $j$, i.e., $\boldsymbol{h}_w^j = \boldsymbol{h}_{\pi(w)}^j, \forall w \in \mathcal{V}$. By definition, AC-GNN $\mathcal{A}$ produces the feature vector $\boldsymbol{h}_v^{j+1}$ of node $v$ in the $(j+1)_{th}$ iteration as follows:

$$\boldsymbol{h}_v^{(j+1)} = \mathrm{COM}^{(j+1)}(\boldsymbol{h}_v^{(j)}, \mathrm{AGG}^{(j+1)}(\{\boldsymbol{h}_u^{(j)} : u \in \mathcal{N}(v)\})). \tag{1}$$

Since an automorphism $\pi$ remains the set of edges, i.e., $\{u, v\} \in \mathcal{E}$ iff $\{\pi(u), \pi(v)\} \in \mathcal{E}$, the connection relation between two neighbors is preserved after the permutation by $\pi$, that is, $\mathcal{N}(\pi(v)) = \{\pi(u), u \in \mathcal{N}(v)\}$ for any $v \in \mathcal{V}$. Then, the input of $\mathrm{AGG}^{(j+1)}$ for $\pi(v)$ is given by $\{\boldsymbol{h}_u^{(j)} : u \in \mathcal{N}(\pi(v))\}$, which is $\{\boldsymbol{h}_{\pi(u)}^{(j)} : u \in \mathcal{N}(v)\}$. Since $\boldsymbol{h}_w^j = \boldsymbol{h}_{\pi(w)}^j, \forall w \in \mathcal{V}$, the input of $\mathrm{AGG}^{(j+1)}$ for $v$ is equal to the one of $\mathrm{AGG}^{(j+1)}$ for $v$, i.e., $\{\boldsymbol{h}_{\pi(u)}^{(j)} : u \in \mathcal{N}(v)\} = \{\boldsymbol{h}_{\pi(u)}^{(j)} : u \in \mathcal{N}(v)\}$ and makes their output equal, i.e., $\boldsymbol{m}_v^{j+1} = \boldsymbol{m}_{\pi(v)}^{j+1}$. Therefore, the input of $\mathrm{COM}^{(j+1)}$ for $v$, $(\boldsymbol{h}_v^{(j)}, \boldsymbol{m}_v^{j+1})$, is also equal to the one of $\mathrm{COM}^{(j+1)}$ for $\pi(v)$, which makes the vector features of $v$ and $\pi(v)$ equal after $(j+1)_{th}$ iteration for any node $v \in \mathcal{V}$ and proves the property 4. Thus, the AC-GNN $\mathcal{A}$ always produces the same node embeddings for the nodes in the equivalent node pair, which results in the same color.

□

## .2.2 Proof of Property 2

We first recall the property.

**Property 2.** *If a graph $\mathcal{G}$ contains two connected nodes $u$ and $v$ that share the same neighborhood except each other, i.e., $\mathcal{N}(u)\backslash\{v\} = \mathcal{N}(v)\backslash\{u\}$, then an integrated AC-GNN cannot discriminate $\{u, v\}$.*

*Proof.* The proof starts with a simple fact: a classifier $CLS(\cdot)$ always assigns two nodes with the same node embedding to the same category.

First, the node pair $\{u, v\}$ is distinct since they are connected. It follows that the inputs for the node features of $u$ and $v$ after iteration $k$ are exactly the same since $\mathcal{N}(u) \cup \{u\} = \mathcal{N}(u)\backslash\{v\} \cup \{u, v\} = \mathcal{N}(v)\backslash\{u\} \cup \{u, v\} = \mathcal{N}(v) \cup \{v\}$. Therefore, the outputs are the same, which means that $\boldsymbol{h}_u^{(j)} = \boldsymbol{h}_v^{(j)}$ holds for any iteration $k$ and any aggregation and combine functions $\mathrm{AGG}(\cdot), \mathrm{COM}(\cdot)$. Combining with the fact that $CLS(\boldsymbol{h}_u) = CLS(\boldsymbol{h}_v)$ if $\boldsymbol{h}_u = \boldsymbol{h}_v$, the proof is finished. □

## .2.3 Proof of Corollary 1

We first recall the corollary.

**Corollary 1.** *A local coloring method is non-optimal in the random d-regular graph as $d \to \infty$.*

*Proof.* A random $d$-regular graph $\mathcal{G}_d^n$ is a graph with $n$ nodes and each node pair is connected with a probability $d/n$. We start the proof from the following non-trivial property:

**Property 5** ([90])**.** *The largest density of factor of i.i.d. independent sets in a random $d$-regular graph is asymptotically at most $(\log d)/d$ as $d \to \infty$. The density of the largest independent sets in these graphs is asymptotically $2(\log d)/d$.*

The property above limits the size of an independent set produced by local method for the random $d$-regular graph with an upper bound, $n(\log d)/d$ as $d \to \infty$. Given an upper bound of the independent set, the following corollary on the graph coloring problem is introduced:

**Corollary 3.** *The lower bound of $k$ with a zero conflict constraint obtained by a local coloring method for the random $d$-regular graph is $d/\log d$ as $d \to \infty$.*

The proof is based on the Property 5: if a local coloring method $f$ obtains a smaller $k'$, s.t. $k' < d/\log d$ by coloring $\mathcal{G}_d^n$ without conflict using $k'$ colors, all node sets classified by the node color will be independent sets and the size of the maximum one will be larger than $(n \log d)/d$, a contradiction with Property 5.

The Corollary 3 reveals the lower bound of $k$ by local methods for a random $d$-regular graph. Another important observation of $k$ by [8] specifies that exact value of the chromatic number (i.e., the minimum $k$) of a random $d$-regular graph. The property is described as follows:

**Property 6** ([8])**.** *Let $t_d$ be the smallest integer $t$ such that $d < 2t \log t$. The chromatic number of a random $d$-regular graph is either $t_d$ or $t_d + 1$.*

It follows directly from Corollary 3 and Property 6 that, we can finish the proof of Corollary 1 by showing that the lower bound of $k$ by local methods is always greater than the exact chromatic number:

$$d/\log d > t_d + 1 \text{ for } \ d \to \infty. \tag{2}$$

Let $f(t) = 2t \log t$ and define $t_0$ s.t. $d = f(t_0) = 2t_0 \log t_0$. Since $t_d$ is the smallest integer $t$ such that $d < f(t)$, we have $f(t_0) = d \geq f(t_d - 1)$. Since $f$ is monotonically increasing, $t_0 \geq t_d - 1$ and thus $d/\log d - t_d - 1 \geq d/\log d - t_0 - 2$ always holds. Let $d = 2t_0 \log t_0$, we further derive the objective below:

$$\begin{aligned} &d/\log d - t_d - 1 \geq d/\log d - t_0 - 2 \\ =&\frac{2t_0 \log t_0}{\log(2t_0 \log t_0)} - t_0 - 2 > 0, \text{ for } \ d, t_0 \to \infty. \end{aligned}$$

we first prove that

$$\begin{aligned}&\frac{\log t_0}{\log(2t_0\log t_0)} > 2/3\\ \Rightarrow &3\log t_0 > 2\log(2t_0\log t_0)\\ \Rightarrow &3\log t_0 > 2(1+\log t_0+\log(\log t_0))\\ \Rightarrow &\log t_0 > 2+2\log(\log t_0) \quad \text{when } t_0\to\infty.\end{aligned}$$

The above inequality holds obviously. Following the objective, we have:

$$\begin{aligned}&\frac{2t_0\log t_0}{\log(2t_0\log t_0)} - t_0 - 2\\ >&\frac{4}{3}t_0 - t_0 - 2 > 0 \quad \text{when } t_0\to\infty.\end{aligned}$$

Therefore, we finish the proof. □

## .2.4 Proof of Corollary 2

We first recall the corollary.

**Corollary 2.** *L-AC-GNN is a L-local coloring method and thus a local coloring method*

*Proof.* Given an AC-GNN $\mathcal{A}$ with $L$ layers, let's consider a $L$-local equivalent node pair $\{u,v\}$ in $\mathcal{G}$ by an $L$-local automorphism $\pi_L$, which means that two rooted subtrees $\mathcal{B}_{\mathcal{G}}(u,L)$ and $\mathcal{B}_{\mathcal{G}}(v,L)$ are isomorphic and $\boldsymbol{x}_w = \boldsymbol{x}_{\pi_r(w)}$ holds for every $w\in\mathcal{B}_{\mathcal{G}}(u,r)$. Since two rooted subtrees are isomorphic, the WL test [114] decides $\mathcal{B}_{\mathcal{G}}(u,L)$ and $\mathcal{B}_{\mathcal{G}}(v,L)$ are isomorphic and assigns the same color to $w$ and $\pi_L(w)$ for any $w\in\mathcal{B}_{\mathcal{G}}(u,L)$. To connect the WL test with AC-GNN, the following property is used:

**Property 7** ([82, 11, 117]). *If the WL test assigns the same color to two nodes in a graph, then every AC-GNN maps the two nodes into the same node embedding.*

Therefore, $\mathcal{A}$ maps the $u$ and $v$ into the same node embedding. It follows that $\mathcal{A}$ is $L$-local and thus local. □

## .2.5 Proof of Property 3

We first recall the theorem.

**Property 3.** *Let $\{u,v\}$ be a node pair in any graph $\mathcal{G}$, and $L$ be any positive integer. If a L-AC-GNN discriminates $\{u,v\}$, a $L^+$-AC-GNN also discriminates it.*

Here, a $L^+$-AC-GNN is defined as an AC-GNN by stacking injective layers after $L$-AC-GNN (before $CLS(\cdot)$). An injective layer includes a pair of injective aggregation function and injective combination function.

*Proof.* Let $A^L$ be the $L$-AC-GNN that discriminates $\{u, v\}$, and $\boldsymbol{h}_u^L, \boldsymbol{h}_v^L$ are the node embedding generated by $A^L$ (before $CLS(\cdot)$) and correspond to the node $u, v$ respectively. Given the condition that $A^L$ discriminates $\{u, v\}$, i.e., $\boldsymbol{h}_u^L \neq \boldsymbol{h}_v^L$, we here consider the case where $L^+$-AC-GNN $A^{L+}$ is an AC-GNN that only stack one injective layer after $A^L$. Then, $\boldsymbol{h}_v^{L+}$, the node embedding of $v$ generated by $A^{L+}$ is defined as:

$$\boldsymbol{h}_v^{L+} = \mathrm{COM}^{L+}(\boldsymbol{h}_v^L, \mathrm{AGG}^{L+}(\{\boldsymbol{h}_m^L : m \in \mathcal{N}(v)\})), \tag{3}$$

where $\mathrm{COM}^{L+}$ and $\mathrm{AGG}^{L+}$ are the combination and aggregation functions of the newly stacked injective layer. Since $\boldsymbol{h}_u^L \neq \boldsymbol{h}_v^L$, and $\boldsymbol{h}_u^L, \boldsymbol{h}_v^L$ are in the input multisets $\{\boldsymbol{h}_u^L, \mathrm{AGG}^{L+}(\{\boldsymbol{h}_m^L : m \in \mathcal{N}(u)\})\}, \{\boldsymbol{h}_v^L, \mathrm{AGG}^{L+}(\{\boldsymbol{h}_m^L : m \in \mathcal{N}(v)\})\}$ respectively, the input multiset of $\mathrm{COM}^{L+}$ when calculating $\boldsymbol{h}_v^{L+}$ is different with the one when calculating $\boldsymbol{h}_u^{L+}$. Because $\mathrm{COM}^{L+}$ is injective, we can further conclude that the output of $\mathrm{COM}^{L+}$ when calculating $\boldsymbol{h}_v^{L+}$ is different with the one when calculating $\boldsymbol{h}_u^{L+}$, that is, $\boldsymbol{h}_u^{L+} \neq \boldsymbol{h}_v^{L+}$. By induction, the inequality can be applied for any additional stacked layers. We finish the proof. □

### .2.6 Proof of Theorem 2

We first recall the theorem.

**Theorem 2.** *Let $\mathcal{A}$ be a simple AC-GNN and both input and output of each layer in $\mathcal{A}$ be the probability distribution $\boldsymbol{h} \in \mathbb{R}^k$ of $k$ colors, $\mathcal{A}$ is color equivariant if and only if the following conditions hold:*

- *For any layer $i$, all the off-diagonal elements of $\boldsymbol{C}^{(i)}$ are tied together and all the diagonal elements are equal as well. That is,*

$$\begin{aligned}&\boldsymbol{C}^{(i)} = \lambda_C^{(i)} \boldsymbol{I} + \gamma_C^{(i)} (\boldsymbol{1}\boldsymbol{1}^\top)\\ &\lambda_C^{(i)}, \gamma_C^{(i)} \in \mathbb{R} \quad ; \boldsymbol{1} = [1, ..., 1]^\top \in \mathbb{R}^k.\end{aligned} \tag{4}$$

- *For any layer $i$, all the off-diagonal elements of $\boldsymbol{A}^{(i)}$ are also tied together and all the diagonal elements are equal as well. That is,*

$$\begin{aligned}&\boldsymbol{A}^{(i)} = \lambda_A^{(i)} \boldsymbol{I} + \gamma_A^{(i)} (\boldsymbol{1}\boldsymbol{1}^\top)\\ &\lambda_A^{(i)}, \gamma_A^{(i)} \in \mathbb{R} \quad \boldsymbol{1} = [1, ..., 1]^\top \in \mathbb{R}^k.\end{aligned} \tag{5}$$

- *For any layer $i$, all elements in $\boldsymbol{b}^{(i)}$ are equal. That is,*

$$\boldsymbol{b}^{(i)} = \beta^{(i)} \boldsymbol{1} \quad \beta^{(i)} \in \mathbb{R} \quad \boldsymbol{1} = [1, ..., 1]^\top \in \mathbb{R}^k. \tag{6}$$

*Proof.* Let $\mathrm{AGG}^{(i)}$ and $\mathrm{COM}^{(i)}$ be the aggregation and combination functions in the $i_{th}$ layer of $\mathcal{A}$. $\mathcal{A}$ is color equivariant if and only if all functions in $\{\mathrm{AGG}^{(i)}, \mathrm{COM}^{(i)} : i \in 1, ..., L\}$ are color equivariant. the aggregation function is color equivariant clearly and thus we are left to consider the color equivariance of combination functions. Considering the definition of color equivariant in Definition 4, the color equivariance of combination function $\mathrm{COM}^{(i)} = \sigma(\boldsymbol{x}\boldsymbol{C}^{(i)} + \boldsymbol{y}\boldsymbol{A}^{(i)} + \boldsymbol{b}^{(i)})$ is given by:

$$\sigma(\boldsymbol{x}\boldsymbol{C}^{(i)} + \boldsymbol{y}\boldsymbol{A}^{(i)} + \boldsymbol{b}^{(i)})\boldsymbol{P} = \sigma(\boldsymbol{x}\boldsymbol{P}\boldsymbol{C}^{(i)} + \boldsymbol{y}\boldsymbol{P}\boldsymbol{A}^{(i)} + \boldsymbol{b}^{(i)}). \tag{7}$$

$\mathrm{COM}^{(i)}$ is color equivariant if and only if the equation above holds for any permutation matrix $\boldsymbol{P} \in \mathbb{R}^{k\times k}$ and any vectors $\boldsymbol{x}, \boldsymbol{y}$. Considering it holds for any vectors $\boldsymbol{x}, \boldsymbol{y}$, We first find three special cases of $\boldsymbol{x}$ and $\boldsymbol{y}$, which are necessary conditions and correspond to three conditions respectively:

*Case 0.* When $\boldsymbol{y} = \boldsymbol{0}$, we have that $\sigma(\boldsymbol{x}\boldsymbol{C}^{(i)})\boldsymbol{P} = \sigma(\boldsymbol{x}\boldsymbol{P}\boldsymbol{C}^{(i)})$ holds for any $\boldsymbol{P}$ and $\boldsymbol{x}$. That is, $\boldsymbol{x}(\boldsymbol{C}^{(i)}\boldsymbol{P} - \boldsymbol{P}\boldsymbol{C}^{(i)}) = \boldsymbol{0}$ always holds, which reveals that $\boldsymbol{C}^{(i)}\boldsymbol{P} = \boldsymbol{P}\boldsymbol{C}^{(i)}$. $\boldsymbol{C}^{(i)}\boldsymbol{P} = \boldsymbol{P}\boldsymbol{C}^{(i)}$ holds for any $\boldsymbol{P}$ follows that $C^{(i)}_{m,m} = C^{(i)}_{n,n}$ and $C^{(i)}_{m,n} = C^{(i)}_{n,m}$ for any $m, n \in \{1, ..., k\}$. Therefore, all the off-diagonal elements of $\boldsymbol{C}^{(i)}$ are tied together and all the diagonal elements are equal as well.

*Case 1.* When $\boldsymbol{x} = \boldsymbol{0}$, we can prove that all the off-diagonal elements of $\boldsymbol{A}^{(i)}$ are tied together and all the diagonal elements are equal as well following the similar induction in case 1.

*Case 2.* When $\boldsymbol{x} = \boldsymbol{y} = \boldsymbol{0}$, we have that $\sigma(\boldsymbol{b}^{(i)})\boldsymbol{P} = \sigma(\boldsymbol{b}^{(i)})$ holds for any $\boldsymbol{P}$. Therefore, all elements in $\boldsymbol{b}^{(i)}$ are equal.

After proving that these conditions are necessary for a color equivariant $\mathcal{A}$, we proceed to prove that the conditions above are already sufficient. Let $\boldsymbol{C}^{(i)} = \lambda^{(i)}_C\boldsymbol{I} + \gamma^{(i)}_C(\boldsymbol{1}\boldsymbol{1}^\top)$, $\boldsymbol{A}^{(i)} = \lambda^{(i)}_A\boldsymbol{I} + \gamma^{(i)}_A(\boldsymbol{1}\boldsymbol{1}^\top)$ and $\boldsymbol{b}^{(i)} = \beta^{(i)}\boldsymbol{1}$, $\mathrm{COM}^{(i)}$ is then calculated by:

$$\begin{aligned}
\mathrm{COM}^{(i)}\boldsymbol{P} &= \sigma(\boldsymbol{x}\boldsymbol{C}^{(i)} + \boldsymbol{y}\boldsymbol{A}^{(i)} + \boldsymbol{b}^{(i)})\boldsymbol{P} \\
&= \sigma(\boldsymbol{x}\lambda^{(i)}_C\boldsymbol{I}\boldsymbol{P} + \boldsymbol{x}\gamma^{(i)}_C(\boldsymbol{1}\boldsymbol{1}^\top)\boldsymbol{P} + \boldsymbol{y}\lambda^{(i)}_A\boldsymbol{I}\boldsymbol{P} \\
&+ \boldsymbol{y}\gamma^{(i)}_A(\boldsymbol{1}\boldsymbol{1}^\top)\boldsymbol{P} + \beta^{(i)}\boldsymbol{1}\boldsymbol{P}) \\
&= \sigma(\boldsymbol{x}\boldsymbol{P}\lambda^{(i)}_C\boldsymbol{I} + \boldsymbol{x}\boldsymbol{P}\gamma^{(i)}_C(\boldsymbol{1}\boldsymbol{1}^\top) + \boldsymbol{y}\boldsymbol{P}\lambda^{(i)}_A\boldsymbol{I} \\
&+ \boldsymbol{y}\boldsymbol{P}\gamma^{(i)}_A(\boldsymbol{1}\boldsymbol{1}^\top) + \beta^{(i)}\boldsymbol{1}) \\
&= \sigma(\boldsymbol{x}\boldsymbol{P}\boldsymbol{C}^{(i)} + \boldsymbol{y}\boldsymbol{P}\boldsymbol{A}^{(i)} + \boldsymbol{b}^{(i)}).
\end{aligned} \tag{8}$$

Therefore, $\mathrm{COM}^{(i)}$ is color equivariant if and only if the conditions hold, which completes the proof.

□

## .3 Preproces & Postprocess in Chapter 4

In our method, we add preprocess and postprocess procedures to reduce the problem complexity and improve the result quality. Note that these techniques are not necessary for our method, in Appendix .5.4, we list the experimental results without any postprocess procedures. At the same time, we have implemented these techniques in DGL or by a series of tensor operations, so that both of them can be efficiently processed by GPU.

**Algorithm 6** ITERATIVEREMOVAL

```
Input: $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\} \rightarrow$ Target graph.
Input: $k \rightarrow$ Number of available colors.
1: while $\exists u \in \mathcal{V}$ s.t. degree of $u < k$ do
2:     Update degree of the neighbor of $u$ by subtracting one.
3:     Remove $u$ in $\mathcal{G}$.
4: end while
```

**Algorithm 7** POSTPROCESS

```
Input: $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\} \rightarrow$ Target graph.
Input: $f \rightarrow$ Coloring results.
1: Is_changed $\leftarrow$ True;
2: while Is_changed do
3:     Is_changed $\leftarrow$ False;
4:     for $u \in \mathcal{V}$ do
5:         for $r \in \{1, ..., k\}$ do
6:             if the conflict # reduces when set $f(u)$ to $r$ then
7:                 $f(u) \leftarrow r$;
8:                 Is_changed $\leftarrow$ True;
9:             end if
10:        end for
11:    end for
12:    for $e = \{u, v\} \in \mathcal{E}$ do
13:        if the conflict # reduces when swap color of $u, v$ then
14:            swap color of $u, v$;
15:            Is_changed $\leftarrow$ True;
16:        end if
17:    end for
18: end while
```

**Preprocess** In the preprocess part, we remove the node with a degree less than k iteratively. Because a node with degree less than $k$ will always not contribute to a conflict in the optimal solution, this kind of removal will not introduce any redundant conflicts. The algorithm is shown in Algorithm 6. There are many other graph simplification techniques in the practical applications such as bridge detection [65], we do not focus on these techniques, because the aim of our work is not to develop a very effective coloring method by powerful pre-process and post-process procedures, but to study the power of GNNs on the coloring problems.

**Postprocess** In the postprocess part, we iteratively detect 1) whether a color change in a single node will decrease the conflict number or 2) whether a swap of colors between connected nodes will decrease the conflict number. The algorithm is shown in Algorithm 7. Generally, we iteratively check (L2 - L18) each node (L4 - L11) and each conflict edge (L12 - L17), if one better solution is found, we modify the coloring result to the better one and continue the iteration.

## .4 Global method

During the exploration of GNNs, the locality of GNNs has been widely observed as an intrinsic nature. The main concern in previous works is that the locality inhibits GNNs from detecting the global graph structure, thereby harming the representation power. In the paper, we discuss one representative "global" technique: deep layers, and show that it can enhance the discrimination power while still cannot make AC-GNNd always global. In this section, we use the notation of the local method defined in our paper, and look back on previous solutions to see whether they provide a truly global scheme by our definition. We hope that our analysis can provide some insights on the global GNNs for future research. We first recap the definition of a local method:

**Definition 3** (local method [36]). *A coloring method $f$ is $r$-local if it fails to discriminate any $r$-local equivalent node pair. A coloring method $f$ is local if $f$ is $r$-local for at least one positive integer $r$.*

To determine whether a coloring method is local or not, we need to, by definition, determine whether the method is able to discriminate two local equivalent nodes. Consider a local equivalent node pair, say $u, v$, we can exam previous global methods by testing whether the node embeddings of $u$ and $v$ are the same. To simplify the discussion and only focus on the main point, we summarize and distill the most representative techniques as follows:

### .4.1 Distance encoding [63].

Distance Encoding is a general class of structure-related features to enrich the substructure or even global structure information. In their work, the distance can be represented in various forms and the distance encoding can be used in two different ways, i.e., extra node features and a controller for message passing. For simplicity, we only consider the case where shortest path distance is used to measure distance and employed as the extra node features. Formally, the input features with distance encoding is:

$$\boldsymbol{h}_v^0 = \boldsymbol{x}_v^0 \oplus \sum_{v \in \mathcal{S}} d(u, v) \tag{9}$$

Here, $\boldsymbol{x}_v^0$ is the original node attribute, $d(u, v)$ is the shortest path distance between $u$ and $v$, $\oplus$ is the concatenation mark, and $\mathcal{S}$ is the target structure defined in the original paper, which can be the whole graph, i.e., $\mathcal{S} = \mathcal{V}$, or a substructure, i.e., $\mathcal{S} \in \mathcal{V}$. We make the following statements:

**Property 4.** *AC-GNNs enhanced by distance encoding ARE global.*

*Proof.* Note that a local method cannot discriminate *any* local equivalent node pair. We can finish the proof by contradiction. Assume there exists $r > 0$ such that the enhanced AC-GNN is a $r$-local method, i.e., it fails to discriminate any r-local equivalent node pair. We build a connected graph containing $2r+3$ nodes like a linked list. A figure illustration is given in Figure 1, where the number represents the node index. Assume all nodes share the same node attribute, i.e., $\boldsymbol{x}_i^0 = \boldsymbol{x}_j^0, \forall i,j \in \{0, ..., 2r+2\}$. Consider the two nodes, $v_r$ and $v_{r+1}$, their depth-$r$ neighborhood are topologically equivalent, i.e., $\mathcal{B}_{\mathcal{G}}(v_r, r) = \mathcal{B}_{\mathcal{G}}(v_{r+1}, r)$. Therefore, $\{v_r, v_{r+1}\}$ is a $r$-local equivalent node pair. However, the distance encoding of the two nodes are different, where $\sum_{v\in\mathcal{G}} d(v_r, v) = r^2+3r+3$ but $\sum_{v\in\mathcal{G}} d(v_{r+1}, v) = r^2+3r+2$, resulting in a difference between $\boldsymbol{h}_r^0$ and $\boldsymbol{h}_{r+1}^0$. Similarly, the neighbors of $v_r$ and $v_{r+1}$ have the different distance encodings. Therefore, the distance encoding makes the two local equivalent nodes differentiable by providing a different input for both aggregation and combination functions, which completes the proof. [1] □

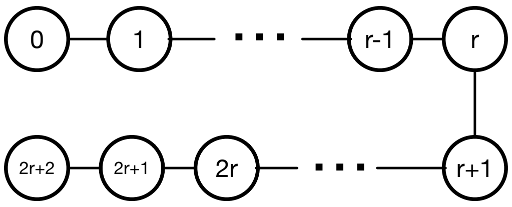


Figure 1: A contradiction example to prove that the distance encoding makes an AC-GNN global.

### .4.2 Readout function [11].

Barcel et al. [11] proposed a scheme to update node features by aggregating not only neighbor information, but also the global attribute vector. The function considering a global attribute vector is also called the readout function. In their work, it is demonstrated that even a very simple readout function, i.e., summation of all node features, can capture all FOC$_2$ classifiers, which means that the representation power is improved. Indeed, the global feature vectors contain some information across the whole graph, and the distance encoding discussed in Section .4.1 is also a kind of readout function in the form of distance measurement. But can we declare that *AC-GNNs become global methods as long as we use a readout function?* Here, we discuss the simplest form used in [11], aggregate-combine-readout GNNs (ACR-GNNs), where the readout is calculated by the summation of all node features. An ACR-GNN is formalized as follows:

$$\begin{aligned}\boldsymbol{h}_v^{(i)} =& \mathrm{COM}^{(i)}(\boldsymbol{h}_v^{(i-1)}, \mathrm{AGG}^{(i)}(\{\boldsymbol{h}_u^{(i-1)} : u \in \mathcal{N}(v)\}), \\ & \mathrm{READ}^{(i)}(\{\boldsymbol{h}_u^{(i-1)} : u \in \mathcal{V}\})),\end{aligned} \tag{10}$$

We make the following statement:

**Property 5.** *ACR-GNNs are NOT global.*

[1] We do not discuss extreme cases here, e.g. $\mathrm{COM}(\cdot) = 0$.

*Proof.* The intuition for proof comes from the fact that the used readout function, i.e., summation of all node features, keeps the same for all nodes. We first prove the following:

**Corollary 4.** *If an ACR-GNN succeeds in discriminating a node pair, an AC-GNN will also discriminate it.*

Given a node pair $\{u, v\}$ in the graph $\mathcal{G}$. Let $\boldsymbol{h}_u^{(k)'}$ and $\boldsymbol{h}_u^{(k)}$ represents the node embedding of $u$ after $k$ layers by an ACR-GNN $\mathcal{A}'$ and an AC-GNN $\mathcal{A}$ respectively. Suppose after $k$ layers, $\mathcal{A}'$ discriminate them, i.e., $\boldsymbol{h}_u^{(k)'} \neq \boldsymbol{h}_v^{(k)'}$, while $\mathcal{A}$ fails to discriminate them, i.e., $\boldsymbol{h}_u^{(k)} = \boldsymbol{h}_v^{(k)}$. It follows that during the layer $t$ from 0 to $k-1$, $\boldsymbol{h}_u^{(t)'} = \boldsymbol{h}_v^{(t)'}$ and $\boldsymbol{h}_u^{(t)} = \boldsymbol{h}_v^{(t)}$. That is, for any $t$ from 0 to $k-1$, we can create a valid mapping $\phi$ such that $\boldsymbol{h}_v^{(t)'} = \phi(\boldsymbol{h}_v^{(t)})$ for any node $v \in \mathcal{V}$.

Consider the inequality after $k$ layers, since node $u$ and $v$ always have the same readout term, i.e., $\text{READ}^{(k)}(\{\boldsymbol{h}_u^{(k-1)'} : u \in \mathcal{V}\})$, combing with Equation 10, it must be the case that:

$$\begin{aligned}(\boldsymbol{h}_v^{(k-1)'}, \{\boldsymbol{h}_s^{(k-1)'} : s \in \mathcal{N}(v)\}) \neq \\ (\boldsymbol{h}_u^{(k-1)'}, \{\boldsymbol{h}_s^{(k-1)'} : s \in \mathcal{N}(u)\})\end{aligned} \tag{11}$$

That is,

$$\begin{aligned}(\phi(\boldsymbol{h}_v^{(k-1)}), \{\phi(\boldsymbol{h}_s^{(k-1)}) : s \in \mathcal{N}(v)\}) \neq \\ (\phi(\boldsymbol{h}_u^{(k-1)}), \{\phi(\boldsymbol{h}_s^{(k-1)}) : s \in \mathcal{N}(u)\})\end{aligned} \tag{12}$$

However, according to the assumption, the AC-GNN fails to discriminate the two nodes, indicating that the inequality above cannot hold. Hence we have reached a contradiction.

Therefore, we can conclude that the ACR-GNN also cannot discriminate any local equivalent node pair, making it a local method. Actually, this proof also demonstrates that such an ACR-GNN is upper-bounded by AC-GNNs in the terms of the discrimination power. □

### .4.3 Identity-aware Graph Neural Networks [125].

Identity-aware Graph Neural Networks (ID-GNNs) focus on solving the problem that the embeddings are only related to the local subtree. The key insight is to inductively consider the root node during message passing, i.e., whether the aggregated node is the target node itself. If the aggregated node is the target node, a different aggregation and combination channel is used so that the ID-GNN is a heterogeneous one. Formally, let the target node is $u$, i.e., we are calculating the node embedding of $u$, then, the mediate features of other nodes are given by:

$$\boldsymbol{h}_{v,u}^{(i)} = \text{COM}^{(i)}(\boldsymbol{h}_{v,u}^{(i-1)}, \{\text{AGG}_{\mathbf{1}[s=u]}^{(i)}(\boldsymbol{h}_{s,u}^{(i-1)}) : s \in \mathcal{N}(v)\}), \tag{13}$$

Here, $\boldsymbol{h}_{v,u}^{(i)}$ represents the mediate feature of node $v$ after $i_{th}$ layer when calculating the node embedding of $u$, $\text{AGG}^{(i)}$ contains two functions, where $\text{AGG}_1^{(i)}$ is applied to the target node, and $\text{AGG}_0^{(i)}$ is for other nodes. The simple heterogeneous scheme makes the target node different from other nodes and therefore sensitive to the identity. However, such a scheme still fails to discriminate $c, d$ in the Figure 1(a) of our paper. Based on this observation, we claim the following property:

**Property 6.** *ID-GNNs are NOT global.*

*Proof.* We can finish the proof by showing that ID-GNNs cannot discriminate any local equivalent node pair. Given an ID-GNNs $\mathcal{A}$ with $L$ layers, let's consider a $L$-local equivalent node pair $\{u, v\}$ in $\mathcal{G}$ by an $L$-local isomorphism $\pi_L$, which means that the two subgraphs $\mathcal{B}_{\mathcal{G}}(u, L)$ and $\mathcal{B}_{\mathcal{G}}(v, L)$ are isomorphic and $\boldsymbol{x}_w = \boldsymbol{x}_{\pi_r(w)}$ holds for every $w \in \mathcal{B}_{\mathcal{G}}(u, r)$. Here, we propose a stronger property:

**Corollary 5.** *Given a L-depth ID-GNN and a L-local equivalent node pair $\{u, v\}$ by $\pi$, $\boldsymbol{h}^i_{s,v} = \boldsymbol{h}^i_{\pi(s),u}$ holds for any iteration $i$ and any node $w \in \mathcal{B}_{\mathcal{G}}(u, r)$ if $i + d(s, v) \leq L$, where $d$ represents the shortest distance.*

We prove the corollary by a nested induction.

**First induction:**

This statement, i.e., $\boldsymbol{h}^i_{s,v} = \boldsymbol{h}^i_{\pi(s),u}$, apparently holds when $i + d(s, v) \leq 0$. Suppose this holds if $i + d(s, v) \leq k$ (first assumption), we now prove that the statement will also hold when $i + d(s, v) = k + 1$ as long as $k + 1 \leq L$.

**Induction in the induction:** For those nodes, say $s_{k+1}$, whose shortest distance with $v$ is $k + 1$, i.e., $d(s_{k+1}, v) = k + 1$, we have $\boldsymbol{h}^0_{s_{k+1},v} = \boldsymbol{h}^0_{\pi(s_{k+1}),u}$ since $\{u, v\}$ is a $L$-local equivalent node pair and $k + 1 \leq L$. Suppose $\boldsymbol{h}^i_{s,v} = \boldsymbol{h}^i_{\pi(s),\pi(v)}$ holds if $i = t$ and $d(s, v) = k+1-t$ (second assumption), we continue to prove that this will hold if $i = t+1$ and $d(s, v) = k - t$.

Consider those nodes, say $s_{k-t}$, whose shortest distance between $v$ is $k - t$, i.e., $d(s_{k-t}, v) = k - t$, then $\boldsymbol{h}^{t+1}_{s_{k-t},v}$ is given by:

$$\begin{aligned}\boldsymbol{h}^{t+1}_{s_{k-t},v} =& \mathrm{COM}^{(t+1)}(\boldsymbol{h}^t_{s_{k-t},v}, \\ & \{\mathrm{AGG}^{(t+1)}_{\mathbf{1}[s=v]}(\boldsymbol{h}^{(t)}_{s,v}) : s \in \mathcal{N}(s_{k-t})\}),\end{aligned} \tag{14}$$

According to the first assumption, $\boldsymbol{h}^t_{s_{k-t},v} = \boldsymbol{h}^t_{\pi(s_{k-t}),u}$ since $i + d(s_{k-t}, v) = k$. We then consider the second term in Equation 14, $\boldsymbol{h}^{(t)}_{s,u} : s \in \mathcal{N}(s_{k-t})$. The distance between the neighbors of $s_{k-t}$ and the root node $v$ ranges from $k - t - 1$ to $k - t + 1$. For the neighbor nodes $s_{k-t-1} \in \mathcal{N}(s_{k-t})$ with a distance $k-t-1$ between $v$, we have $\boldsymbol{h}^t_{s_{k-t-1},v} = \boldsymbol{h}^t_{\pi(s_{k-t-1}),u}$ since $t + d(s_{k-t-1}, v) = k - 1 \leq k$ (first assumption). Similarly, for the neighbor nodes $s'_{k-t} \in \mathcal{N}(s_{k-t})$, the equation still holds since $t + d(s'_{k-t}, v) = k \leq k$ (first assumption). For the neighbor nodes $s_{k-t+1} \in \mathcal{N}(s_{k-t})$, the equation $\boldsymbol{h}^t_{s_{k-t+1},v} = \boldsymbol{h}^t_{\pi(s_{k-t+1}),u}$ also holds since $i = t$ and $d(s_{k-t+1}, v) = k + 1 - t$ (second assumption).

**End of the induction in the induction:** Hence by mathematical induction $\boldsymbol{h}^i_{s,v} = \boldsymbol{h}^i_{\pi(s),\pi(v)}$ is correct for all positive integers $i$ and $d(s, v)$. Therefore, we show that $\boldsymbol{h}^i_{s,v} = \boldsymbol{h}^i_{\pi(s),\pi(v)}$ holds when $i + d(s, v) = k + 1 \leq L$, $d(s, v)$ and $i$ are positive integers.

**End of the induction:** Hence by mathematical induction, $\boldsymbol{h}^i_{s,v} = \boldsymbol{h}^i_{\pi(s),u}$ holds for any iteration $i$ and any node $w \in \mathcal{B}_{\mathcal{G}}(u, r)$ if $i + d(s, v) \leq L$, such completes the proof of Corollary 4.

Based on Corollary 4, we can conclude that $\boldsymbol{h}^L_{v,v} = \boldsymbol{h}^L_{u,u}$, indicating that the $L$-depth ID-GNN fails to discriminate $u$ and $v$, which completes the proof. □

### .4.4 Randomness.

In the development of GNNs, random schemes are widely-used and studied. Ryoma et al. [94] and Andreas et al. [73] both prove that the distinct node attributes (even initialized randomly) enhance the representation power significantly. George et al. [27] propose a randomly coloring methods to distinguish different nodes and break the local equivalence. Position-aware GNN [126] makes use of the distance encoding to design a position-aware GNN, where one of the differences between distance encoding [63] is that the distance is not measured with a pre-defined set $\mathcal{S}$, but with a set of *randomly* selected anchor node sets. In our work, we also demonstrated that the randomness enhances the discrimination power of AC-GNNs, because nodes are not possible to be local equivalent considering that their node attributes are initialized randomly. Therefore, we want to know:

**Q:** *Does randomness make AC-GNNs global?*

Unfortunately, we are not able to answer the question now. We can only declare that a random scheme indeed helps to distinguish local equivalent node pair, but it *may* be still local. The reason is that the AC-GNNs are not deterministic anymore if we add some randomness, therefore, the definition of local methods is not available here. In some cases, the upper bound (or lower bound) remains when the function becomes not deterministic, but a formal proof is needed in our case. We look forward to a deeper discussion on the discrimination power of randomness in AC-GNNs, and leave this as our future work.

## .5 Supplementary Experiments in Chapter 4

### .5.1 Experiment & model settings.

**Experiments.** We implemented our experiments in the PyTorch Deep Graph Library (DGL) [109]. We conducted all experiments on a server with a Titan X GPU and an E5-2630 2.6 GHz CPU. Besides GNN-GCP and tabucol compared in the paper, we also implement integer linear programming (ILP) based solver by Gurobi [83], and three heuristics coloring methods which are used as baselines in [48]. In the supplementary experiments, 80% randomly selected samples in the layout dataset are separated into the training dataset (for trainable models) and the testing dataset contains the remaining samples. In the paper, we only run our model once with a specified $k$ and obtain the cost (conflict number), in the supplementary experiments, we sometimes need to calculate the chromatic number by our model, i.e., calculate the minimum $k$ that achieves a zero cost. To do this, we iteratively run our model and add $k$ by one after each iteration until a zero cost is received.

**Network and training.** A detailed network and training setting is covered in the Appendix. Layout dataset is used as the training data and we will show that our TreeNet trained on these small graphs can generalize to much larger ones such as Citation dataset. Other datasets are comprised of either (1) a single graph, or (2) graphs with varying chromatic numbers, which are not suitable for training, especially for other variations. During training, we initialize the variables of TreeNet in each layer as:

$$\lambda_C^{(i)} = 1, \gamma_C^{(i)} = 0, \lambda_A^{(i)} = -1, \gamma_A^{(i)} = 0, \beta^{(i)} = 0 \tag{15}$$

The initialization is motivated by the truth that neighbors of each node should be assigned as different colors as the node.

**Model hyperparameters.** If not specified, all AC-GNN variations have 2 layers and the hidden dimension is 64. Other training parameters of these variations keep the same with our proposed TreeNet . We use the Adam [53] optimizer with a 1024 batched graph (if batch-graph is possible) and a learning rate 0.001. Training proceeds for 10 epochs and takes about one hour. More detailed default hyperparameter values can be found in our released code, i.e., `color_arg.py`.

**Datasets.** We totally evaluate performance on five datasets.

(1) The layout dataset, which is composed of many simplified small but dense layout graphs transformed from circuit layout. This dataset is widely used as the benchmark for the layout decomposition problem [50, 66, 132], a similar problem in the industry manufacturing based on the graph coloring problem. The number of available colors $k$ is set to 3 following previous works;

(2) The citation networks (Cora, Citeseer, and Pubmed) [98] that contains real-world graphs from academic search engines. We follow the setting in [69] and regard them as the coloring scenario for large but sparse graphs, hence dismissing their original node attributes and edge directions. $k$ in Cora, Citeseer, and Pubmed are set to 5, 6 and 8 respectively;

(3) Three random graph distributions namely: random power-law tree, Watts-Strogatz small-world and Holme and Kim model [112, 45], some works [60] evaluated them on the graph coloring problem. $k$ is set from 2 to 5 due to its randomness. The settings to generate the random graphs are: random power-law tree ($\gamma$=3), Watts- Strogatz small-world ($k = 4$, $p = 0.25$) and Holme and Kim model ($m = 4$, $p = 0.1$), To fit the problem context and prevent miscalculations, we removed all self-loops in the testing graphs and added duplex connections between connected nodes;

(4) COLOR dataset[2] that contains medium sized graphs, which is also the most essential dataset in the graph coloring community [60, 38, 116]. Here, we select instances following [60], other instances show a similar trend in our experiments;

[2] https://mat.tepper.cmu.edu/COLOR02/

(5) Regular dataset that contains $d$-regular graphs with size $n$. We use NetworkX [?] to randomly generate 100 graphs whose density is 16 and graph size is 128. The color number is set to $d/\log d + 1 = 5$.

For some tasks (Random, Citation) whose available color numbers $k$ are not specified, we assign $k$ as the chromatic numbers of graphs, which are obtained from the CSP Solver[3]. Each graph $\mathcal{G}$ is first preprocessed by removing vertexes iteratively following steps in [132, 50].

## .5.2 Comparison with other methods.

We compare our method with the other three heuristics coloring methods use in [48] and ILP based method. We only compare these methods in the layout dataset, because ILP even cannot solve others within 24 hours. The three heuristic algorithms are summarized as follows:

**Static-ordered:** Coloring nodes in the order of node IDs.

**Sorted-ordered:** Coloring nodes in the largest degree first manner.

**Dynamic-ordered:** Coloring nodes in the largest degree first manner, while the degree is updated when coloring nodes, i.e., the neighbors of the colored node will decrease their degree by one.

The results are shown in Figure 2. We measure the average predicted $k$ (minimal color number to be conflict-free) on four different graph size $|\mathcal{V}|$, i.e., small ($|\mathcal{V}|$ <8), Medium ($8 \leq |\mathcal{V}| < 16$), Large ($16 \leq |\mathcal{V}| < 32$), Huge ($32 \leq |\mathcal{V}|$). All methods contain a preprocess procedure for a fair comparison. From the results, we can see that *Our method achieves exactly the same performance with ILP in all graphs except the huge one.* Note that ILP is an optimal coloring solver, indicating that our method reaches the optimality for relatively small graphs. However, even for such small cases, three heuristic algorithms fail to be close to ILP or our method. For large and huge graphs, the average $k$ is increased by more than 10% for static and dynamic algorithms. With the growth of the graph size, our method becomes more and more advanced compared with these heuristic algorithms.

## .5.3 Ablation study

**Model depth** In our paper, we show that a deep AC-GNN is a more powerful coloring solver (Property 3). Here, we validate our conclusion by experiments. The results on layout dataset are shown in Figure 3(a), where the solved ratio is defined as in paper, i.e., the ratio between the number of edges without introducing conflicts and the number of total edges. According to the results, we can conclude that a deeper model indeed has a more positive influence on the results. However, the ratio improvement gradually slows down and eventually stops as the model goes deeper: when the model is deeper enough, it is able to cover all graphs in the layout dataset. The results on regular dataset as shown in Figure 3(b) also align with our proof. With the increase of the model depth, our method

[3]https://developers.google.com/optimization/cp/cp_solver

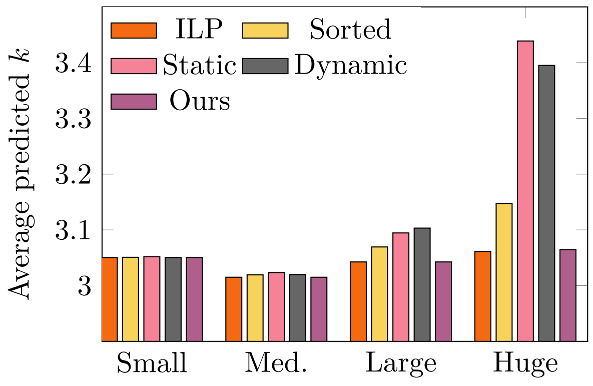


Figure 2: Comparison with other heuristic methods.

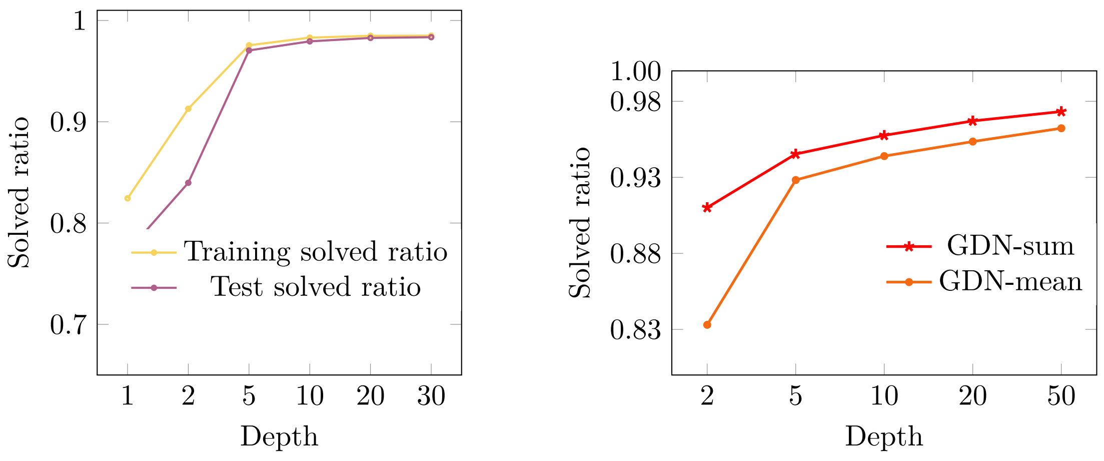
becomes more powerful regardless of what the aggregation function is. The phenomenon also demonstrates the theorem in [73], i.e., the product of the GNN' s depth and width must exceed a polynomial of the graph size to obtain an optimal result.

**Injective function** In Property 3, we demonstrate that an injective aggregation and combination function guarantee a more powerful AC-GNN. In our proposed GDN, we use summation as the aggregation function since sum aggregators can represent injective function over multisets (Lemma 5, [117]). Here, we replace summation with mean aggregator to see its performance in the regular dataset. The results are shown in Figure 3(b), we can see that our method with sum aggregator is always better than the mean aggregator among all depths.

**Integrated AC-GNN & Equivalent nodes** In Property 1 and Property 2, we state the drawbacks of integrated scheme and equivalent nodes in the coloring problem. To solve these issues, we respectively summarize two rules to make AC-GNN more powerful: Do not use integrated AC-GNN and avoid equivalent nodes by assigning nodes different attributes. We also try two variations of our methods, where the first one integrates the aggregation and combination:

$$\boldsymbol{h}_v^{(i)} = \lambda_C^{(i)}(\boldsymbol{h}_v^{(i-1)} + \sum_{u \in \mathcal{N}(v)} \boldsymbol{h}_u^{(i-1)}) + \beta^{(i)}\mathbf{1} \tag{16}$$

The second one set the node attribute as the same one while keep other steps the same. However, both variations fail to discriminate any two nodes, resulting in a zero solved ratio on all datasets after several layers.



(a) layout dataset.

(b) regular dataset.

Figure 3: Comparison with different model depth and mean aggregation.

### .5.4 Other discussion.

**Postprocess.** We discuss the influence of our proposed GPU-friendly postprocess on the result quality and runtime. We compare our method with postprocess and without postprocess on layout dataset (Table 2 ) and normal dataset (Table 1). In the layout dataset, the relatively simple one, our post process can reduce the average predicted $k$ by 1%. More importantly, the postprocess part makes our method optimal for more than 99.9% layout graphs except for the huge one, which occupies less than 0.1% in the total dataset. In the harder normal dataset, the conflict will increase by 73.4% if postprocess is not used, as a scarifies, the runtime is increased by 12.8% when using postprocess. However, compared with the significant accuracy improvement, the time loss is acceptable, especially under the occasion that our method is 500× faster than a heuristic algorithm with a similar-quality. In Table 1, our method without postprocess sometimes even results in a better solution than the one with postprocess, this happens because we randomly initialize our node attribute, resulting in a slightly different solution everytime. Actually, we can further improve our performance by a repeated running like previous coloring methods [137, 15, 103], but our target is to provide insights for powerful GNNs on coloring problems instead of developing a powerful coloring solver by some simple tricks.

**Other techniques for heterophily.** In [139], they propose a concatenation technique for tasks under heterophily, i.e., concatenate all features in the middle layers, and compute the final embedding using the concatenated result. We also implement it for comparison, the results are shown in Table 2. According to the table, we can see that it fails to be effective in the coloring problem, which even increases $k$ a little bit. Nevertheless, it is still an open and interesting question to find the effective techniques in the coloring tasks,

Table 1: The results of our method without postprocess on the normal dataset.

| | Ours | | Ours w.o. post | |
|---|---|---|---|---|
| | cost | time | cost | time |
| jean | 0 | 0.13 | 0 | 0.11 |
| anna | 0 | 0.17 | 0 | 0.15 |
| huck | 0 | 0.13 | 5 | 0.11 |
| david | 1 | 0.19 | 0 | 0.17 |
| homer | 1 | 0.29 | 1 | 0.26 |
| myciel5 | 0 | 0.12 | 0 | 0.10 |
| myciel6 | 0 | 0.21 | 1 | 0.18 |
| games120 | 0 | 0.08 | 0 | 0.07 |
| Mug88_1 | 0 | 0.01 | 0 | 0.01 |
| 1-Insertions_4 | 0 | 0.07 | 0 | 0.07 |
| 2-Insertions_4 | 2 | 0.08 | 1 | 0.07 |
| Queen5_5 | 0 | 0.05 | 7 | 0.04 |
| Queen6_6 | 4 | 0.05 | 5 | 0.04 |
| Queen7_7 | 11 | 0.06 | 15 | 0.05 |
| Queen8_8 | 7 | 0.06 | 16 | 0.05 |
| Queen9_9 | 10 | 0.09 | 18 | 0.06 |
| Queen8_12 | 7 | 0.09 | 14 | 0.08 |
| Queen11_11 | 24 | 0.07 | 38 | 0.07 |
| Queen13_13 | 42 | 0.08 | 68 | 0.08 |
| ratio | 1.000 | 1.000 | 1.734 | 0.872 |

Table 2: The results of our methods without postprocess and with concatenation on the layout dataset. $k$ is the average predicted chromatic number, and the $\uparrow$ (%) is the increase compared with Our original model.

| | | Small | Med. | Large | Huge |
|---|---|---|---|---|---|
| ILP | $k$ | 3.0505 | 3.0151 | 3.0425 | 3.0612 |
| Ours | $k$ | 3.0505 | 3.0151 | 3.0425 | 3.0645 |
| Ours | $k$ | 3.0548 | 3.0181 | 3.0451 | 3.0669 |
| w.o. post | $\uparrow$ (%) | 1.4 | 1.0 | 0.9 | 0.8 |
| Ours | $k$ | 3.0512 | 3.0151 | 3.0425 | 3.0653 |
| w. concat | $\uparrow$ (%) | 0.3 | 0 | 0 | 0.3 |

and even in the general tasks under heterophily.

---

□ **End of chapter.**

# Bibliography

[1] Boost C++ Library. http://www.boost.org. 17, 33

[2] CBC. http://www.coin-or.org/projects/Cbc.xml. 18

[3] LEMON. http://lemon.cs.elte.hu/trac/lemon. 18

[4] Limbo. http://yibolin.com/Limbo/docs/html/index.html. 17

[5] OpenMP. http://www.openmp.org/. 19

[6] OpenMPL. https://github.com/limbo018/OpenMPL. 13, 19, 35

[7] S. Abu-El-Haija, B. Perozzi, A. Kapoor, N. Alipourfard, K. Lerman, H. Harutyunyan, G. V. Steeg, and A. Galstyan. Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. *arXiv preprint arXiv:1905.00067*, 2019. 7

[8] D. Achlioptas and A. Naor. The two possible values of the chromatic number of a random graph. In *ACM Symposium on Theory of computing (STOC)*, pages 587–593, 2004. 95

[9] C. J. Alpert, W.-K. Chow, K. Han, A. B. Kahng, Z. Li, D. Liu, and S. Venkatesh. Prim-Dijkstra revisited: Achieving superior timing-driven routing trees. In *ACM International Symposium on Physical Design*, pages 10–17, 2018. 77, 80, 82, 83, 86, 87, 88, 89

[10] C. J. Alpert, T. Hu, J. Huang, A. B. Kahng, and D. Karger. Prim-Dijkstra tradeoffs for improved performance-driven routing tree design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(7):890–896, 1995. 77

[11] P. Barceló, E. V. Kostylev, M. Monet, J. Pérez, J. Reutter, and J. P. Silva. The logical expressiveness of graph neural networks. In *International Conference on Learning Representations (ICLR)*, 2019. , 6, 64, 66, 67, 69, 70, 96, 101

[12] E. Bengoetxea. *Inexact Graph Matching Using Estimation of Distribution Algorithms*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, France, Dec 2002. 45

[13] B. Borchers. CSDP, a C library for semidefinite programming. *Optimization Methods and Software*, 11:613–623, 1999. 18

[14] A. Boulch. ConvPoint: Continuous convolutions for point cloud processing. *Computers & Graphics*, 2020. 10

[15] A. Braunstein, M. Mézard, M. Weigt, and R. Zecchina. Constraint satisfaction by survey propagation., 2006. 11, 71, 108

[16] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017. 1, 2

[17] H.-Y. Chang and I. H.-R. Jiang. Multiple patterning layout decomposition considering complex coloring rules. In *ACM/IEEE Design Automation Conference (DAC)*, pages 40:1–40:6, 2016. , 16, 26, 27, 29, 32, 33, 34, 37, 38, 39, 44, 47

[18] G. Chen, P. Tu, and E. F. Young. SALT: provably good routing topology by a novel Steiner shallow-light tree algorithm. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 569–576, 2017. 80

[19] G. Chen and E. F. Young. Salt: provably good routing topology by a novel steiner shallow-light tree algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019. 77, 86, 87, 89

[20] H.-Y. Chen, Y.-T. Hou, K. Yu-Hsiang, K.-H. Hsieh, R.-G. Liu, and L.-C. Lu. Layout optimization for integrated circuit design, 2017. US Patent. 15

[21] M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li. Simple and deep graph convolutional networks. In *International Conference on Machine Learning*, pages 1725–1735. PMLR, 2020. 70

[22] Z. Chen, S. Villar, L. Chen, and J. Bruna. On the equivalence between graph isomorphism testing and function approximation with gnns. *arXiv preprint arXiv:1905.12560*, 2019. 6

[23] H.-A. Chien, S.-Y. Han, Y.-H. Chen, and T.-C. Wang. A cell-based row-structure layout decomposer for triple patterning lithography. In *ACM International Symposium on Physical Design (ISPD)*, pages 67–74, 2015. 15, 41

[24] C. Chu and Y.-C. Wong. FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(1):70–83, 2008. 77

[25] H. Dai, B. Dai, and L. Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711. PMLR, 2016. 7

[26] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*, 2017. 7

[27] G. Dasoulas, L. D. Santos, K. Scaman, and A. Virmaux. Coloring graph neural networks for node disambiguation. *arXiv preprint arXiv:1912.06058*, 2019. 70, 104

[28] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *arXiv preprint arXiv:1606.09375*, 2016. 7

[29] S. S. Du, K. Hou, B. Póczos, R. Salakhutdinov, R. Wang, and K. Xu. Graph neural tangent kernel: Fusing graph neural networks with graph kernels. *arXiv preprint arXiv:1905.13192*, 2019. 64

[30] Y. Duan, Y. Zheng, J. Lu, J. Zhou, and Q. Tian. Structural relational reasoning of point clouds. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 949–958, 2019. 10

[31] M. Elkin and S. Solomon. Steiner shallow-light trees are exponentially lighter than spanning ones. In *IEEE Symposium on Foundations of Computer Science*, pages 373–382, 2011. 77

[32] S.-Y. Fang, Y.-W. Chang, and W.-Y. Chen. A novel layout decomposition algorithm for triple patterning lithography. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 33(3):397–408, March 2014. 15, 16, 19, 34, 39, 40, 41

[33] W. Fang, S. Arikati, E. Cilingir, M. A. Hug, P. De Bisschop, J. Mailfert, K. Lucas, and W. Gao. A fast triple-patterning solution with fix guidance. In *Proceedings of SPIE*, volume 9053, 2014. 16

[34] Y. Feng, Z. Zhang, X. Zhao, R. Ji, and Y. Gao. GVCNN: Group-view convolutional neural networks for 3D shape recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 264–272, 2018. 8

[35] F. Gama, J. Bruna, and A. Ribeiro. Stability properties of graph neural networks. *IEEE Transactions on Signal Processing*, 68:5680–5695, 2020. 64, 71

[36] D. Gamarnik and M. Sudan. Limits of local algorithms over sparse random graphs. In *Proceedings on Innovations in theoretical computer science*, pages 369–376, 2014. 64, 66, 68, 69, 70, 100

[37] F. Groh, P. Wieschollek, and H. P. Lensch. Flex-Convolution. In *Asian Conference on Computer Vision (ACCV)*, pages 105–122. Springer, 2018. 9

[38] S. Gualandi and F. Malucelli. Exact solution of graph coloring problems via constraint programming and column generation. *INFORMS Journal on Computing*, 24(1):81–100, 2012. 105

[39] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun. Deep learning for 3D point clouds: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2020. 8, 9

[40] Gurobi Optimization Inc. Gurobi optimizer reference manual. `http://www.gurobi.com`, 2016. 18

[41] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1024–1034, 2017. 7, 68, 74

[42] W. L. Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017. 64

[43] P. Hermosilla, T. Ritschel, P.-P. Vázquez, À. Vinacua, and T. Ropinski. Monte carlo convolution for learning on non-uniformly sampled point clouds. *ACM Transactions on Graphics (TOG)*, 37(6):1–12, 2018. 9

[44] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987. 74

[45] P. Holme and B. J. Kim. Growing scale-free networks with tunable clustering. *Physical review E*, 65(2):026107, 2002. 105

[46] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018. 83

[47] B.-S. Hua, M.-K. Tran, and S.-K. Yeung. Pointwise convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 984–993, 2018. 10

[48] J. Huang, M. Patwary, and G. Diamos. Coloring big graphs with alphagozero. *arXiv preprint arXiv:1902.10162*, 2019. 11, 64, 104, 106

[49] I. H.-R. Jiang and H.-Y. Chang. Multiple patterning layout decomposition considering complex coloring rules and density balancing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 36(12):2080–2092, 2017. 41, 42, 57

[50] I. H.-R. Jiang and H.-Y. Chang. Multiple patterning layout decomposition considering complex coloring rules and density balancing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 36(12):2080–2092, 2017. 71, 105, 106

[51] A. B. Kahng, C.-H. Park, X. Xu, and H. Yao. Layout decomposition approaches for double patterning lithography. *IEEE Transactions on Computer-Aided Design*

*of Integrated Circuits and Systems (TCAD)*, 29:939–952, June 2010. 15, 16, 18, 19, 24, 41, 44, 46, 52

[52] M.-C. Kim, J. Hu, J. Li, and N. Viswanathan. ICCAD-2015 CAD Contest in incremental timing-driven placement and benchmark suite. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 921–926, 2015. 86

[53] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 105

[54] T. N. Kipf et al. *Deep learning with graph-structured representations*. 2020. 1

[55] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. 3, 67, 68, 74

[56] W. Klotz. *Graph coloring algorithms*. Verlag nicht ermittelbar, 2002. 39, 40

[57] J. Kuang and E. F. Y. Young. An efficient layout decomposition approach for triple patterning lithography. In *ACM/IEEE Design Automation Conference (DAC)*, pages 69:1–69:6, 2013. 16, 19, 20, 39, 41, 42, 48, 52, 54, 57

[58] J. Kuang and E. F. Y. Young. Fixed-parameter tractable algorithms for optimal layout decomposition and beyond. In *ACM/IEEE Design Automation Conference (DAC)*, pages 61:1–61:6, 2017. 15, 41

[59] H. Lei, N. Akhtar, and A. Mian. Octree guided CNN with spherical kernels for 3D point clouds. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 9631–9640, 2019. 9

[60] H. Lemos, M. Prates, P. Avelar, and L. Lamb. Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 879–885. IEEE, 2019. 11, 71, 74, 105

[61] G. Li, M. Muller, A. Thabet, and B. Ghanem. DeepGCNs: Can GCNs go as deep as CNNs? In *IEEE International Conference on Computer Vision (ICCV)*, pages 9267–9276, 2019. 70

[62] H. Li, G. Chen, B. Jiang, J. Chen, and E. F. Young. Dr. cu 2.0: A scalable detailed routing framework with correct-by-construction design rule satisfaction. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7. IEEE, 2019. 34

[63] P. Li, Y. Wang, H. Wang, and J. Leskovec. Distance encoding–design provably more powerful GNNs for structural representation learning. *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020. , 6, 67, 100, 104

[64] W. Li, Y. Ma, Q. Sun, Y. Lin, I. H.-R. Jiang, B. Yu, and D. Z. Pan. OpenMPL: An open source layout decomposer. In *IEEE International Conference on ASIC (ASICON)*, 2019. 52, 57

[65] W. Li, Y. Ma, Q. Sun, Y. Lin, I. H.-R. Jiang, B. Yu, and D. Z. Pan. Openmpl: An open source layout decomposer. In *2019 IEEE 13th International Conference on ASIC (ASICON)*, pages 1–4. IEEE, 2019. 99

[66] W. Li, J. Xia, Y. Ma, J. Li, Y. Liny, and B. Yu. Adaptive layout decomposition with graph embedding neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020. 59, 60, 73, 105

[67] X. Li, Z. Zhu, and W. Zhu. Discrete relaxation method for triple patterning lithography layout decomposition. *IEEE Transactions on Computers*, 66(2):285–298, 2017. 15, 41

[68] Y. Li, R. Bu, M. Sun, W. Wu, X. Di, and B. Chen. PointCNN: Convolution on x-transformed points. In *Annual Conference on Neural Information Processing Systems (NIPS)*, pages 820–830, 2018. 8, 9, 10, 78, 82, 83, 85, 86, 88

[69] Z. Li, Q. Chen, and V. Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *Annual Conference on Neural Information Processing Systems (NIPS)*, pages 539–548, 2018. 6, 64, 73, 105

[70] Y. Lin, X. Xu, B. Yu, R. Baldick, and D. Z. Pan. Triple/quadruple patterning layout decomposition via linear programming and iterative rounding. *Journal of Micro/Nanolithography, MEMS, and MOEMS (JM3)*, 16(2), 2017. 15, 16, 19, 39, 41

[71] Y. Liu, B. Fan, S. Xiang, and C. Pan. Relation-shape convolutional neural network for point cloud analysis. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8895–8904, 2019. 8, 9, 10

[72] Z. Lou, J. You, C. Wen, A. Canedo, J. Leskovec, et al. Neural subgraph matching. *arXiv preprint arXiv:2007.03092*, 2020. 7, 64

[73] A. Loukas. What graph neural networks cannot learn: depth vs width. *arXiv preprint arXiv:1907.03199*, 2019. 6, 64, 67, 69, 104, 107

[74] A. Loukas. How hard is to distinguish graphs with graph neural networks? Technical report, 2020. 64

[75] Y. Ma, J.-R. Gao, J. Kuang, J. Miao, and B. Yu. A unified framework for simultaneous layout decomposition and mask optimization. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 81–88, 2017. 16

[76] Y. Ma, X. Zeng, and B. Yu. Methodologies for layout decomposition and mask optimization: A systematic review. In *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6, 2017. 15

[77] H. Maron, H. Ben-Hamu, H. Serviansky, and Y. Lipman. Provably powerful graph networks. *arXiv preprint arXiv:1905.11136*, 2019. 6

[78] H. Maron, H. Ben-Hamu, N. Shamir, and Y. Lipman. Invariant and equivariant graph networks. *arXiv preprint arXiv:1812.09902*, 2018. 6, 71

[79] T. Matsui, Y. Kohira, C. Kodama, and A. Takahashi. Positive semidefinite relaxation and approximation algorithm for triple patterning lithography. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 365–375, 2014. 18, 47

[80] D. Maturana and S. Scherer. Voxnet: A 3D convolutional neural network for real-time object recognition. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 922–928. IEEE, 2015. 8

[81] B. D. McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26(2):306–324, 1998. 54

[82] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609, 2019. 6, 67, 96

[83] G. OPTIMIZATION. Inc. gurobi optimizer reference manual, 2015. *URL: http://www. gurobi. com*, page 29, 2014. 104

[84] D. Z. Pan, L. Liebmann, B. Yu, X. Xu, and Y. Lin. Pushing multiple patterning in sub-10nm: Are we ready? In *ACM/IEEE Design Automation Conference (DAC)*, pages 197:1–197:6, 2015. 15

[85] D. Z. Pan, B. Yu, and J.-R. Gao. Design for manufacturing with emerging nanolithography. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 32(10):1453–1472, 2013. 41

[86] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NeurIPS Autodiff Workshop*, 2017. 57

[87] H. Pei, B. Wei, K. C.-C. Chang, Y. Lei, and B. Yang. Geom-GCN: Geometric graph convolutional networks. *arXiv preprint arXiv:2002.05287*, 2020. 6

[88] C. R. Qi, H. Su, K. Mo, and L. J. Guibas. PointNet: Deep learning on point sets for 3D classification and segmentation. In *IEEE Conference on Computer Vision*

*and Pattern Recognition (CVPR)*, pages 652–660, 2017. 3, 8, 9, 71, 78, 83, 85, 86, 88

[89] C. R. Qi, L. Yi, H. Su, and L. J. Guibas. PointNet++: Deep hierarchical feature learning on point sets in a metric space. In *Annual Conference on Neural Information Processing Systems (NIPS)*, pages 5099–5108, 2017. 8, 9, 10, 78, 82, 83, 85, 86, 88

[90] M. Rahman, B. Virag, et al. Local algorithms for independent sets are half-optimal. *The Annals of Probability*, 45(3):1543–1577, 2017. 64, 68, 70, 95

[91] G. Riegler, A. Osman Ulusoy, and A. Geiger. OctNet: Learning deep 3D representations at high resolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3577–3586, 2017. 8

[92] R. B. Rusu and S. Cousins. 3D is here: Point cloud library (pcl). In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–4. IEEE, 2011. 9

[93] R. Sato, M. Yamada, and H. Kashima. Approximation ratios of graph neural networks for combinatorial problems. *arXiv preprint arXiv:1905.10261*, 2019. 7

[94] R. Sato, M. Yamada, and H. Kashima. Random features strengthen graph neural networks. *arXiv preprint arXiv:2002.03155*, 2020. 69, 104

[95] R. Scheifele. Steiner trees with bounded RC-delay. *Algorithmica*, 78(1):86–109, 2017. 77

[96] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018. 53

[97] J. M. Schwarz, D. N. Cooper, M. Schuelke, and D. Seelow. Mutationtaster2: mutation prediction for the deep-sequencing age. *Nature methods*, 11(4):361–362, 2014. 85

[98] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008. 105

[99] Y. Shen, C. Feng, Y. Yang, and D. Tian. Mining point cloud local structures by kernel correlation and graph pooling. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4548–4557, 2018. 8, 9

[100] M. Simonovsky and N. Komodakis. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3693–3702, 2017. 8, 9, 10

[101] D. Stolee. Isomorph-free generation of 2-connected graphs with applications. *arXiv preprint arXiv:1104.5261*, 2011. 54, 55

[102] H. Su, S. Maji, E. Kalogerakis, and E. Learned-Miller. Multi-view convolutional neural networks for 3D shape recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 945–953, 2015. 8

[103] Y. Takefuji and K. C. Lee. Artificial neural networks for four-coloring map problems and k-colorability problems. *IEEE Transactions on Circuits and Systems I*, 38(3):326–333, 1991. 11, 108

[104] G. Te, W. Hu, A. Zheng, and Z. Guo. Rgcnn: Regularized graph cnn for point cloud segmentation. In *acmmm*, pages 746–754, 2018. 10

[105] H. Tian, H. Zhang, Q. Ma, Z. Xiao, and M. D. F. Wong. A polynomial time triple patterning algorithm for cell based row-structure layout. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 57–64, 2012. 15, 41

[106] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017. 68

[107] C. Wang, B. Samari, and K. Siddiqi. Local spectral graph convolution for point set feature learning. In *eccv*, pages 52–66, 2018. 10

[108] M. Wang, L. Yu, D. Zheng, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. 57

[109] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315*, 2019. 104

[110] P.-S. Wang, Y. Liu, Y.-X. Guo, C.-Y. Sun, and X. Tong. O-CNN: Octree-based convolutional neural networks for 3D shape analysis. *ACM Transactions on Graphics (TOG)*, 36(4):1–11, 2017. 8

[111] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon. Dynamic graph CNN for learning on point clouds. *ACM Transactions on Graphics (TOG)*, 38(5):1–12, 2019. 8, 9, 10, 78, 82, 83, 85, 86, 88

[112] D. J. Watts and S. H. Strogatz. Collective dynamics of ‘small-world’ networks. *nature*, 393(6684):440–442, 1998. 105

[113] X. Wei, R. Yu, and J. Sun. View-GCN: View-based graph convolutional network for 3D shape analysis. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1850–1859, 2020. 8

[114] B. Weisfeiler and A. A. Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968. 96

[115] F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger. Simplifying graph convolutional networks. In *International conference on machine learning*, pages 6861–6871. PMLR, 2019. 67

[116] Q. Wu and J.-K. Hao. Coloring large graphs based on independent set extraction. *Computers & Operations Research*, 39(2):283–290, 2012. 105

[117] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018. 6, 64, 67, 68, 71, 74, 96, 107

[118] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning*, pages 5453–5462. PMLR, 2018. 7, 70

[119] K. Xu, J. Li, M. Zhang, S. S. Du, K.-i. Kawarabayashi, and S. Jegelka. What can neural networks reason about? *arXiv preprint arXiv:1905.13211*, 2019. 6

[120] Y. Xu and C. Chu. GREMA: graph reduction based efficient mask assignment for double patterning technology. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 601–606, 2009. 15, 19, 41

[121] Y. Xu, T. Fan, M. Xu, L. Zeng, and Y. Qiao. SpiderCNN: Deep learning on point sets with parameterized convolutional filters. In *European Conference on Computer Vision (ECCV)*, pages 87–102, 2018. 9

[122] J. Yang, Q. Zhang, B. Ni, L. Li, J. Liu, M. Zhou, and Q. Tian. Modeling point clouds with self-attention and gumbel subset sampling. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3323–3332, 2019. 8, 9, 10

[123] Y. Yang, C. Feng, Y. Shen, and D. Tian. Foldingnet: Point cloud auto-encoder via deep grid deformation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 206–215, 2018. 8, 10

[124] Z. Yang and L. Wang. Learning relationships for multi-view 3D object recognition. In *IEEE International Conference on Computer Vision (ICCV)*, pages 7505–7514, 2019. 8

[125] J. You, J. Gomes-Selman, R. Ying, and J. Leskovec. Identity-aware graph neural networks. *arXiv preprint arXiv:2101.10320*, 2021. , 6, 102

[126] J. You, R. Ying, and J. Leskovec. Position-aware graph neural networks. *arXiv preprint arXiv:1906.04817*, 2019. 6, 70, 104

[127] J. You, R. Ying, and J. Leskovec. Design space for graph neural networks. *arXiv preprint arXiv:2011.08843*, 2020. 6, 7

[128] B. Yu, Y.-H. Lin, G. Luk-Pat, D. Ding, K. Lucas, and D. Z. Pan. A high-performance triple patterning layout decomposer with balanced density. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 163–169, 2013. 15, 19, 20, 41

[129] B. Yu and D. Z. Pan. Layout decomposition for quadruple patterning lithography and beyond. In *ACM/IEEE Design Automation Conference (DAC)*, pages 53:1–53:6, 2014. 18, 47

[130] B. Yu, S. Roy, J.-R. Gao, and D. Z. Pan. Triple patterning lithography layout decomposition using end-cutting. *Journal of Micro/Nanolithography, MEMS, and MOEMS (JM3)*, 14(1):011002–011002, 2015. 15, 41

[131] B. Yu, K. Yuan, D. Ding, and D. Z. Pan. Layout decomposition for triple patterning lithography. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 34(3):433–446, March 2015. , 15, 16, 17, 18, 19, 20, 23, 24, 25, 34, 35, 38, 39, 40, 41, 44, 46, 47, 52, 57

[132] B. Yu, K. Yuan, D. Ding, and D. Z. Pan. Layout decomposition for triple patterning lithography. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(3):433–446, 2015. 105, 106

[133] K. Yuan, J.-S. Yang, and D. Z. Pan. Double patterning layout decomposition for simultaneous conflict and stitch minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 29(2):185–196, Feb. 2010. 15, 16, 41, 52

[134] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola. Deep sets. In *Annual Conference on Neural Information Processing Systems (NIPS)*, pages 3391–3401, 2017. 71, 72

[135] Y. Zhang, W.-S. Luk, H. Zhou, C. Yan, and X. Zeng. Layout decomposition with pairwise coloring for multiple patterning lithography. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 170–177, 2013. 16

[136] W. Zhong, S. Hu, Y. Ma, H. Yang, X. Ma, and B. Yu. Deep learning-driven simultaneous layout decomposition and mask optimization. In *ACM/IEEE Design Automation Conference (DAC)*, 2020. 16

[137] Y. Zhou, J.-K. Hao, and B. Duval. Reinforcement learning based local search for grouping problems: A case study on graph coloring. *Expert Systems with Applications*, 64:412–422, 2016. 12, 73, 108

[138] J. Zhu, R. A. Rossi, A. Rao, T. Mai, N. Lipka, N. K. Ahmed, and D. Koutra. Graph neural networks with heterophily. *arXiv preprint arXiv:2009.13566*, 2020. 71

[139] J. Zhu, Y. Yan, L. Zhao, M. Heimann, L. Akoglu, and D. Koutra. Beyond homophily in graph neural networks: Current limitations and effective designs. *Advances in Neural Information Processing Systems*, 33, 2020. 7, 108