VERTEX COLORING BY SUBGRAPH EXPANSION IN UNSUPERVISED GRAPH NEURAL NETWORKS: CONSTRUCTING A CURRICULUM BY ITERATIVE GROWTH OF SUBGRAPHS OF AN INPUT GRAPH

by
SEFA YILDIZ

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
July 2025

VERTEX COLORING BY SUBGRAPH EXPANSION IN UNSUPERVISED GRAPH NEURAL NETWORKS: CONSTRUCTING A CURRICULUM BY ITERATIVE GROWTH OF SUBGRAPHS OF AN INPUT GRAPH

Approved by:

Prof. CAN AKKAN .....
(Thesis Supervisor)

Prof. ENES ERYARSOY .....

Assoc. Prof. AYLA GÜLCÜ .....

Date of Approval: July 18, 2025

ABSTRACT

VERTEX COLORING BY SUBGRAPH EXPANSION IN UNSUPERVISED GRAPH NEURAL NETWORKS: CONSTRUCTING A CURRICULUM BY ITERATIVE GROWTH OF SUBGRAPHS OF AN INPUT GRAPH

SEFA YILDIZ

DATA SCIENCE M.S. THESIS, JULY 2025

Thesis Supervisor: Prof. CAN AKKAN

Keywords: unsupervised learning, graph neural network, curriculum learning, incremental learning, vertex coloring

Vertex coloring is a classic combinatorial optimization problem in which each vertex of a graph must be assigned a color so that no two adjacent vertices share the same color. This problem is known to be NP-complete for determining whether a graph can be colored with k colors, and finding the minimum number of colors without conflicts is NP-hard (Garey & Johnson, 1990). It has important applications in scheduling, register allocation, timetabling, and other domains. In recent years, Graph Neural Networks (GNNs) have emerged as powerful models for graph-structured learning, and can tackle vertex coloring by framing it as an unsupervised node-classification task. In particular, Schuetz, Brubaker, Zhu & Katzgraber (2022) show that a physics-inspired approach optimized a Potts-based loss to encourage valid colorings. This GNN-based method has achieved performance on par with or better than classical solvers, even scaling to graphs with millions of vertices.

In this thesis, I propose a novel curriculum-inspired training strategy for vertex coloring using unsupervised GNNs. The method begins by training on a small subgraph and incrementally expands the training set (through layer-by-layer BFS-based, Breadth-First-Search, expansion, degree-first BFS-based expansion, or random walk expansion) to cover the entire graph. This curriculum-like incremental training exposes the network learn in stages, leveraging pre-trained embeddings and pre-trained model weights from the subgraph of the previous stage. Two GNN archi-

tectures (a Graph Convolution model and a GraphSAGE model) were implemented and trained with a continuous Potts-based loss. We evaluated our approach on a subset of the COLOR benchmark dataset (including Mycielski graphs, n-queen graphs, and a social network graph). The experiments show that the subgraph expansion methods—especially the degree-first BFS variant—reduces total training time by 30–35% while maintaining statistically comparable conflict counts. These results suggest that the curriculum-like incremental training is a promising direction for leveraging GNNs in combinatorial graph tasks.

ÖZET

DENETIMSIZ GRAFIK SINIR AĞLARINDA ALTGRAF GENIŞLETMESIYLE KÖŞE BOYAMASI: GIRDI GRAFININ ALTGRAFLARININ İTERATIF BÜYÜTÜLMESI YOLUYLA BIR MÜFREDAT OLUŞTURMA

SEFA YILDIZ

VERİ BİLİMİ YÜKSEK LİSANS TEZİ, TEMMUZ 2025

Tez Danışmanı: Prof. Dr. CAN AKKAN

Anahtar Kelimeler: gözetimsiz öğrenme, grafik sinir ağları, müfredat öğrenme, artımlı öğrenme, köşe boyama

Köşe boyaması, bir grafın her köşesine öyle renkler atamayı amaçlayan klasik bir kombinatoryel optimizasyon problemidir ki bitişik iki köşe aynı rengi paylaşmasın. Bir grafın k renkle boyanabilir olup olmadığını belirlemenin NP-tam, çatışmasız en az renk sayısını bulmanın ise NP-zor olduğu bilinmektedir (Garey & Johnson, 1990). Köşe boyaması; çizelgeleme, yazmaç tahsisi (register allocation), sınav programlama ve benzeri pek çok alanda önemli uygulamalara sahiptir. Son yıllarda Grafik Sinir Ağları (GNN) graf yapılı veriler üzerinde güçlü öğrenme modelleri olarak öne çıkmış, köşe boyaması problemini de gözetimsiz köşe sınıflandırma görevi biçiminde ele alabilmektedir. Özellikle Schuetz et al. (2022), geçerli boyamaları teşvik eden Potts-tabanlı bir kaybı eniyileyen fizik esinli yaklaşımların, klasik çözücülerle aynı hatta daha iyi performans sergilediğini ve milyonlarca köşeye kadar ölçeklenebildiğini göstermiştir.

Bu tezde, gözetimsiz GNN'lerle köşe boyaması için müfredat-esinli yeni bir eğitim stratejisi öneriyorum. Yöntem, önce küçük bir altgraf üzerinde eğitimi başlatıp eğitim kümesini kademeli olarak tüm grafı kapsayacak biçimde genişletmektedir (katman-katman BFS, derece-öncelikli BFS veya rasgele yürüyüş genişlemesi). Böylece ağ, önceki aşamanın ön-eğitilmiş gömülerini ve model ağırlıklarını devralarak aşamalı bir öğrenme sürecine tabi tutulur. Sürekli Potts kaybıyla eğitilen iki GNN mimarisi (Graph Convolution Model ve GraphSAGE) uygulanmıştır.

Yaklaşımımız, COLOR kıyaslama veri kümesinin bir alt kümesi (Mycielski grafları, n-queens grafları ve bir sosyal ağ grafı) üzerinde değerlendirilmiştir. Deneyler, alt-graf genişletme yöntemlerinin—özellikle derece-öncelikli BFS varyantının—toplam eğitim süresini %30–35 oranında azalttığını, buna karşın çatışma sayılarının istatistiksel olarak karşılaştırılabilir düzeyde kaldığını göstermektedir. Bu bulgular, müfredat benzeri artımlı eğitimin kombinatoryel grafik görevlerinde GNN'lerden yararlanmak için umut verici bir yön olduğunu düşündürmektedir.

ACKNOWLEDGEMENTS

First and foremost, I express my deepest gratitude to my supervisor, Prof. Dr. Can Akkan. From the very first meeting to the final draft, his door was always open and his enthusiasm never faded. Whenever I left his office, setbacks felt lighter, replaced by a clear plan and renewed purpose. His patient guidance, thoughtful questions, and unfailing optimism turned every obstacle into an opportunity to learn. I will always be grateful. More than an advisor, he has been a mentor and role model. The integrity, curiosity, and kindness he demonstrates daily are values I hope to carry forward in my own career.

I also thank the members of my jury committee, Prof. Dr. Enes Eryarsoy, Assoc. Prof. Ayla Gülcü and Assist. Prof. Dr. Zeki Kuş for their helpful comments and support.

My heartfelt appreciation goes to my dear friend Defne, who traveled every step of this master's journey beside me. Her companionship turned challenges into shared adventures and made the workload feel lighter. This thesis simply would not shine as brightly without her unwavering encouragement, brilliance, and friendship.

I am equally grateful to my close friends for their continuous support and encouragement. Their companionship provided balance, laughter, and motivation throughout this challenging journey.

Finally, I owe everything to my parents; my brothers, Emre and Ahmet; and my sister, Merve. Thank you for your love, patience, and faith. Your support carried me through every stage of this work.

To the late nights, the lost weekends,
and every quiet sacrifice that shaped this work.
May these pages stand as proof that the effort was worthwhile.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

1. INTRODUCTION

The graph coloring problem is one of the fundamental problems in graph theory (Bondy & Murty, 2008). Graph theory is a branch of mathematics that studies the properties and applications of graphs. The aim of vertex coloring problem is to find the assignment of colors to nodes of a graph in a way that no two neighboring nodes have the same color.

The vertex coloring problem is a well-known and active research area within combinatorial optimization. The problem of determining whether a graph can be colored with a given number of colors, is known to be NP-complete, and finding the minimum number of colors is NP-hard. According to Garey & Johnson (1990), a decision problem is in NP if any proposed solution can be checked in polynomial time. A problem is NP-hard if every problem in NP can be reduced to it in polynomial time—informally, it is at least as difficult as the hardest problems in NP. If a problem is NP-hard and also lies in NP, then it is said to be NP-complete (Garey & Johnson, 1990).

1.1 Types of Graph Coloring

Graph coloring problem consists of variety of coloring problems, each focusing on different elements and constraints within a graph. Fundamentally, a graph coloring problem is the assignment of colors, sometimes referred to as labels, to an element of the graph, such as nodes, edges, or other structural components of a graph. With this assignment of colors, certain constraints must be satisfied within the graph.

Vertex coloring is one of the well-known graph coloring problems. In the vertex coloring problem, each vertex of a graph is assigned a color such that no two adjacent vertices share the same color. Vertex coloring finds applications in various

areas, such as scheduling, register allocation in compilers, and pattern matching (Formanowicz & Tanaś, 2012; Lewis, 2016). Figure 1.1 (Lewis, 2016) shows an example of vertex coloring with five colors.

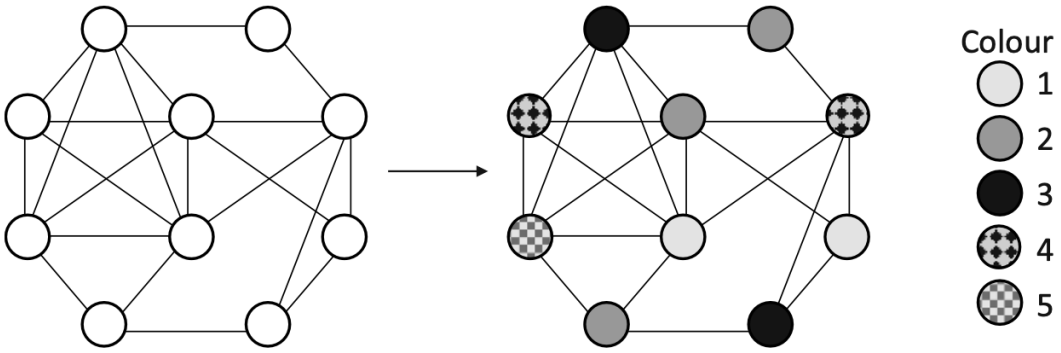


Figure 1.1 A graph with 10 vertices and 21 edges, colored with 5 colors

A notable instance of vertex coloring is the four-color problem, which asks whether it is possible to color the regions of any planar map using no more than four colors in such a way that no two adjacent regions share the same color (Lewis, 2016). The planar graph means that the graph can be drawn on a 2-dimensional plane so that no edges cross one another. The problem was introduced by Francis Guthrie, who first posed it in 1852 while trying to color a map of England's counties. Despite its seemingly simple statement, the conjecture resisted proof for more than a century. An example of the four-color theorem applied (u/DarkerThanAzure, 2023) to the United States Congressional Districts can be seen in Figure 1.2.

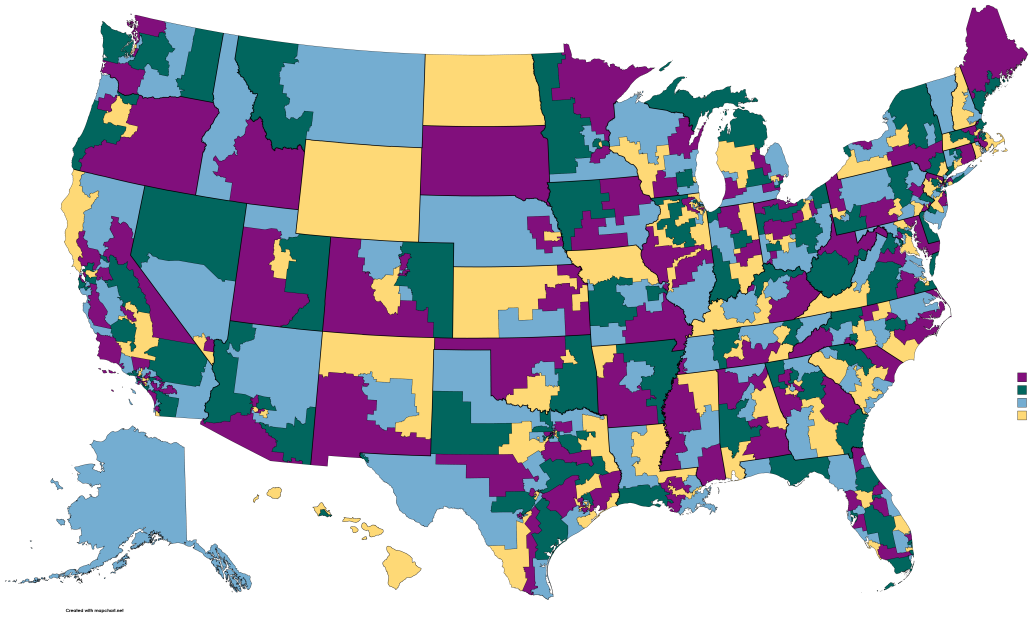


Figure 1.2 US congressional districts colored to uphold the four color theorem

In 1977, Appel and Haken, together with Koch, announced the first successful proof of the four-color theorem in their papers (Appel & Haken, 1977). Their work combined traditional mathematical arguments with extensive computer-based checks, making it one of the first examples of a major theorem proved with the help of a computer. Since then, a variety of alternative proofs and simplified arguments have been proposed, but the essence of the original problem and its computational solution remain the cornerstone of graph theory and combinatorial research (Lewis, 2016).

In addition to vertex coloring, there are several other types of graph coloring. The other well-known graph coloring problem is edge coloring, where two adjacent edges cannot share the same color. Edge coloring has practical application in tasks such as route planning and transfer assignment in computer networks (Formanowicz & Tanaś, 2012).

List coloring is another variation that is similar to the vertex coloring problem in terms of assigning each vertex a color such that no pair of adjacent vertices have the same color. In addition to the vertex coloring property, each vertex's color must come from its own color or label list (Lewis, 2016).

Further types of graph coloring can be found in the literature, such as weighted vertex graph coloring, weighted edge graph coloring, path coloring, and total coloring (Formanowicz & Tanaś, 2012; Lewis, 2016).

1.2 Applications

Graph coloring is used in a variety of different domains, such as scheduling, resource allocation, and map coloring. By converting a problem into a graph coloring problem, the interpretability of the problem is increased due to the visual aspect of graph structures that contributes to a better understanding. In addition, graph coloring is a scalable and efficient technique for solving large-scale problems (Boman, Bozdağ, Catalyurek, Gebremedhin & Manne, 2005). Some of the common applications are discussed below, but these can be extended further due to the flexible and general nature of graph modeling (Formanowicz & Tanaś, 2012; Lewis, 2016).

Scheduling and timetabling problems frequently appear in a variety of contexts such as academic exam scheduling, course timetabling, and sports tournament planning

(Carter, 1986; Ganguli & Roy, 2017; Lewis & Thompson, 2011). Although each problem has a different usage, many of these problems share a fundamental characteristic. The problems require assigning resources to tasks while avoiding conflicts. Resources can be time slots, classrooms or staff, and tasks can be exams, courses, or matches (Lewis, 2016). For instance, in exam timetabling, nodes would represent exams, and colors would be time slots, and any two exams that share students are connected by an edge so that they must not be given the same color.

Graph coloring provides a strong framework for solving resource allocation problems. In general, resource allocation problems are modeled as vertex coloring, but there are also resource allocation scenarios that map more naturally with edge coloring.

Frequency allocation in wireless networks is one of the most classic examples of vertex coloring. Transmitters that are too close to each other or that might otherwise interfere with each other must be assigned different frequencies. This problem is translated into a graph where the vertices represent the transmitters and the edges connect the transmitters that are likely to experience interference (Hale, 1980). Scheduling problems can also be viewed as a form of resource allocation since tasks are assigned to time slots.

Graph coloring has applications in different areas of computer science as well. One of the most well-known applications is register allocation in compilers. Register allocation is the process of assigning variables to registers in a computer program. The goal is to minimize the number of memory accesses, which are slower than register accesses. Register allocation can be modeled as a graph coloring problem, where the vertices represent variables and the edges represent conflicts between the variables that cannot be assigned to the same register (Chaitin, 1982; Lewis, 2016).

1.3 Thesis Outline

This thesis proposes a novel subgraph expansion approach to solve the vertex coloring problem using unsupervised Graph Neural Networks (GNNs). Specifically, a curriculum is constructed by iteratively growing subgraphs of the input graph. In this way, it gradually increases the size and complexity of the problem. To the best of our knowledge, this curriculum-based subgraph expansion strategy is the first of its kind applied to unsupervised learning. It makes an interesting research direction for tackling the NP-complete k-color vertex coloring problem. Unlike clas-

sical exact algorithms, which do not scale well, heuristics and metaheuristics, which lack optimality guarantees, and learning-based methods, which require long training and many random restarts, our method balances scalability, solution quality, and training efficiency.

The remainder of the thesis is organized as follows. Chapter 2 reviews the relevant literature on combinatorial optimization and graph coloring. It discusses classical and learning-based approaches (including exact algorithms, heuristic and metaheuristic methods, and graph neural network-based techniques) and covers incremental and curriculum learning. Chapter 3 describes our proposed methodology, introducing the graph coloring framework based on incremental subgraph expansion, and detailing the various subgraph construction strategies (layer-by-layer breadth-first-search (BFS), degree-first BFS, and random walk expansions) along with the architecture of the GNN models used (Graph Convolutional Network and GraphSAGE). It also defines the objective functions (a loss based on Potts model and a hard coloring cost) and outlines the training procedure, including techniques such as noise injection and the use of pretrained embeddings and use of pretrained model weights. Chapter 4 presents the computational experiments and results, describing the datasets used (Mycielski graphs, queen graphs, and a social network graph), the experimental setup and hyperparameter tuning process, and an analysis of the performance of our approach in comparison to the baseline method. Finally, Chapter 5 concludes the thesis with a discussion of the findings, highlighting the implications of the proposed approach, its limitations, and suggestions for future research directions.

2. LITERATURE REVIEW

2.1 Overview of Combinatorial Optimization

Combinatorial Optimization (CO) involves finding an optimal solution from a finite, mostly exponentially large set of configurations. It is at the core of numerous real-world tasks, including scheduling, routing, and resource management (Papadimitriou & Steiglitz, 1998). Many Combinatorial Optimization problems can be modeled on graphs, where vertices and edges represent elements and constraints of the problem. The following subsections define key concepts in Combinatorial Optimization, discuss classical solution approaches, and highlight notable graph problems, providing the details for understanding graph coloring as a classical example of a Combinatorial Optimization problem.

Combinatorial optimization involves finding optimal solutions from a finite set of configurations, typically these problems show up with discrete constraints (Papadimitriou & Steiglitz, 1998). These problems arise in critical applications ranging from logistics scheduling (Vogiatzis & Pardalos, 2013) to telecommunications network design (Resende, 2003). Graph-based CO problems are of particular significance, as many real-world systems, including transportation infrastructures and social networks, are modeled as graphs (Jin, Yan, Liu & Wang, 2024; Wu, Pan, Chen, Long, Zhang & Yu, 2021). To solve them, classical methods for solving combinatorial optimization problems can be categorized into *exact algorithms* and *heuristics and metaheuristics*.

Traditional exact algorithms such as Integer Linear Programming (ILP) guarantee the finding of an optimal solution by exhaustively searching the feasible region (Papadimitriou & Steiglitz, 1998; Wolsey & Nemhauser, 2014). A problem formulated as ILP can be solved using algorithms such as branch-and-bound, branch-and-cut,

and branch-and-price. Branch-and-bound systematically splits the search space and prunes subproblems using lower and upper bounds in the objective. Branch-and-cut uses cutting-plane methods to tighten the feasible region, and branch-and-price improves these techniques to handle large-scale or more complex structures by generating variables on demand.

The biggest limitation of these methods is their exponential complexity, making them infeasible for large-scale instances. Although modern optimization solvers such as GUROBI or CPLEX can handle moderately sized problems efficiently, they tend to encounter performance bottlenecks as the size or complexity of the problem increases.

Limitations of the exact methods for larger or more complex problems make researchers often use faster heuristics. Simple heuristics, such as local search or constructive algorithms using greedy strategies, can produce solutions quickly, but typically with no guarantee of optimality (Glover, 1986). In addition, *metaheuristics* such as genetic algorithms, simulated annealing, tabu search, and ant colony optimization can be applied to a wide range of combinatorial optimization tasks (Dorigo & Di Caro, 1999; Glover, 1990; Holland, 1975; Kirkpatrick, Gelatt & Vecchi, 1983). Although these methods can scale to very large problem sizes and tend to yield near-optimal results, they lack a provable guarantee of global optimality. Despite that limitation, classical heuristics and metaheuristics remain widely used in practice because of their simplicity, adaptability, and generally good performance in many real-world problems.

2.2 Notable Graph Problems

Most graph-based combinatorial optimization problems are classified as NP-hard, indicating that no known polynomial-time algorithm can solve them optimally for all instances (Garey & Johnson, 1990). This section briefly introduces several well-known graph-based problems, such as *graph coloring, maximum-cut, traveling salesman problem, clique and vertex cover problems*, to illustrate the scope of such problems, solution strategies, and difficulties of such problems.

Graph coloring consists of different types of coloring problems such as vertex coloring, edge coloring, map coloring, and list coloring (see Section 1.1 for details) and has a wide variety of applications, such as scheduling, timetabling, resource

allocation, and register allocation in compilers (see Section 1.2 for details). Despite research on heuristics and metaheuristics (Lewis, 2016) as well as specialized exact methods (Lewis, 2016; Papadimitriou & Steiglitz, 1998; Wolsey & Nemhauser, 2014), large-scale graph coloring instances remain challenging (Garey & Johnson, 1990), encouraging the exploration of more advanced methods.

The **maximum-cut** problem involves partitioning the set of vertices into two disjoint subsets to maximize the total weight of the edges that cross between them, or if the edges do not have any weight, then maximize the number of edges that cross between them (Goemans & Williamson, 1995). Max-cut is an NP-hard problem and due to the combinatorial explosion inherent in the Max-Cut problem, it quickly becomes computationally expensive for large or dense graphs (Garey & Johnson, 1990). Max-cut problem has many real-world applications, such as those in statistical physics and in the design of VLSI circuits (Ben-Ameur, Mahjoub & Neto, 2014).

One of the most famous NP-hard problems is the **Traveling Salesman Problem** (TSP) Garey & Johnson (1990). TSP requires finding a Hamiltonian cycle of minimum total cost (duration, distance, etc.) that visits every vertex exactly once (Laporte, 2010). In the TSP, each node represents a location and edges represent routes with cost. TSP can be solved by heuristics, exact methods (Laporte, 2010) and modern machine learning algorithms (Junior Mele, Maria Gambardella & Montemanni, 2021).

The goal of the **maximum clique problem** is to find the largest subset of nodes in which each node is connected to every other node by an edge. This formalizes finding the largest complete subgraph in a graph (Wu & Hao, 2015). This problem belongs to the class of NP-hard problems (Garey & Johnson, 1990). The maximum clique problem has real-world applications, from network analysis on detecting tightly knit communities to bioinformatics (Wu & Hao, 2015).

The **vertex covering** problem is to find the smallest subset of nodes that touches all edges of the graph. This problem also belongs to the class of NP-hard problems (Garey & Johnson, 1990). The vertex cover problem has a wide variety of applications in society, in computer networks, and in business processes (Gusev, 2020; Sachdeva, 2012).

All of these problems share some common properties. One of them is that, being in the class of NP-hard problems, as the graph grows, the solution space becomes exponentially large in the worst case. Although there is ongoing research on exact and heuristic methods, finding provably optimal solutions for large instances remains an

active area of study. In addition, designing more efficient and modern architectures for NP-hard graph problems remains an active area of study (Schuetz, Brubaker & Katzgraber, 2022).

2.3 Techniques for Vertex Coloring

Graph coloring consists of different types of problems, including vertex coloring, edge coloring, and list coloring (see Section 1.1 for more details). Here we will focus on vertex coloring, where the aim is to assign colors to vertices so that no two adjacent vertices (nodes) share the same color. The smallest number of colors needed is the chromatic number of the graph. The problem has been studied for decades, resulting in a variety of proposed solutions (Appel & Haken, 1977; Lewis, 2016). This section first summarizes classical approaches including exact algorithms and heuristics with their advantages and limitations. Then it continues with learning-based approaches, especially Graph Neural Networks (GNNs), that have recently been applied to graph coloring.

2.3.1 Brief Overview of Classical and Heuristic Approaches

Exact algorithms guarantee finding an optimal solution, but are computationally expensive. Traditional exact methods include dynamic programming (DP) and integer linear programming (ILP) (de Lima & Carmo, 2018). For example, *DSATUR* (Brélaz, 1979) is one of the earliest examples and the well-known exact branch-and-bound type algorithm. *ZykovColor* is one of the recent studies on exact graph coloring (Brand, Faber, Held & Mutzel, 2025). The authors claim superior performance on random Erdos-Renyi graphs, particularly in very sparse and highly dense scenarios. Although these methods give an exact solution, they are computationally expensive with high-density and large graphs (Méndez-Díaz & Zabala, 2006). The state-of-the-art exact algorithms are not suitable for very large graphs such as graphs with thousands or millions of vertices (Malaguti, Monaci & Toth, 2011).

Heuristic algorithms provide solutions faster, but do not always yield an optimal solution. One of the well-known heuristic algorithms is a simple greedy heuristic

(Welsh & Powell, 1967). The algorithm colors vertices sequentially, each time assigning the smallest available color. DSATUR algorithm also has a greedy method for the vertex coloring problem (Brélaz, 1979). It uses the degree of saturation to reduce conflicts. In addition to the degree of saturation, there are other ordering strategies as well such as Largest-Degree-First (LDF) (Hansen, Kubale, Kuszner & Nadolski, 2004), Smallest-Last-Ordering (SLO) (Matula & Beck, 1983), Recursive-Largest-First (RLF) (Palubeckis, 2008). Greedy heuristics are fast compared to exact methods, however, they often produce suboptimal solutions (Lewis, 2016).

The limitation of exact methods for coloring large graphs is partially solved with metaheuristics. Techniques such as simulated annealing, tabu search, genetic and evolutionary algorithms have been tailored for graph coloring (Mostafaie, Modarres Khiyabani & Navimipour, 2020). These approaches can find good colorings for large benchmark graphs, often within a few colors of the optimal. Metaheuristics are more effective than simple heuristics, and they can also handle large problem instances. However, metaheuristics have some limitations. Similarly to simple heuristics, they do not prove optimality, and they may require long run times to reach the best solutions if the graph is dense or large. Most of the metaheuristics require careful tuning of parameters, such as the cooling schedule and population size.

2.3.2 Learning-based Approaches

In recent years, learning-based approaches have been applied to graph coloring problems. These approaches have centered on graph neural networks (GNN) trained with supervised, unsupervised learning approaches, and reinforcement learning. Table 2.1 compares these methods with respect to their key advantages and limitations.

In supervised learning, learning occurs with training on datasets with known solutions. Then, supervised models learn to predict the correct labels for unseen data. To the best of our knowledge, there is not any supervised learning approach developed for vertex coloring. However, there are some studies on predicting the chromatic number of a graph. For instance, a study showed that a GNN-based model could find the chromatic number of large graphs with higher accuracy than traditional heuristics (Ijaz, Ali, Ali, Laique & Ali Khan, 2022).

Compared to the supervised methods that require labeled training examples, unsupervised approaches aim to learn color assignments (label) without explicit ground truth labels. In most of the cases, unsupervised models define an objective, such as

minimizing the number of conflicting edges, and then optimize it.

A recent study shows that the coloring task using an energy function as a loss inspired by physics-based models, often referred to as the continuous Potts or antiferromagnetic Potts model, yields a strong performance compared to heuristics (Schuetz et al., 2022). Such unsupervised GNN approaches have two key advantages. First, they do not require a large set of labeled graphs. Second, they can scale to large graphs. The trade-off is that the model can converge to local minima or remain sensitive to hyperparameter tuning.

In addition to supervised and unsupervised learning, semi-supervised learning takes advantage of both labeled and unlabeled data. This approach is useful when input data are not fully labeled or cannot be fully labeled. One of the pioneering examples for semi-supervised learning using graph data is introduced by Kipf & Welling (2017). In this study, the graph convolution network (GCN) method was also introduced.

Reinforcement learning is a machine learning technique that differs from supervised learning techniques by not initially providing a correct label/action. This correct action/label is found by the RL agent through repeated and incremental interactions with the environment. With these interactions, the agent receives reward or penalty feedback and the agent updates its policy, which is the strategy that learning tries to optimize, to achieve the desired objective (Yang & Whinston, 2023). In RL for vertex coloring, an agent learns to color a graph incrementally, aiming to minimize conflicts. RL methods learn from trial and error rather than requiring a large set of labeled colorings. They provide a flexible coloring strategy. Classical RL only works with manually created features, but deep reinforcement learning (DRL), which is a combination of reinforcement learning and deep neural network, can learn from representation learning. Representation learning is the conversion of high-dimensional data such as raw pixels or graphs into compact data form such as embeddings. This makes reinforcement learning more scalable (Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, Fidjeland, Ostrovski & others, 2015). Research has shown that deep reinforcement learning (DRL) can be utilized to discover competitive construction heuristics for graph coloring, highlighting the potential of RL in this domain (Watkins, Montana & Branke, 2023). Building on these insights, the next section will dive deeper into recent advances in deep learning for graph-structured data, particularly graph neural networks (GNNs).

Table 2.1 Comparison of Graph Coloring Techniques

Approach	Advantages	Limitations	Key References
Exact Methods	• Guaranteed optimal solution • Conceptual clarity (branch-and-bound, ILP)	• Exponential time complexity • Impractical for large/dense graphs	• Malaguti et al. (2011) (Set Covering) • Méndez-Díaz & Zabala (2006) (Branch-and-cut)
Heuristics	• Very fast • Straightforward implementation	• Often produce suboptimal colorings • No optimality guarantee	• Brélaz (1979) (DSATUR) • Welsh & Powell (1967) (Welsh Powell Algorithm)
Metaheuristics	• Improved accuracy vs. simple heuristics • Scale to large instances • Flexible frameworks (e.g., genetic, tabu)	• Parameter tuning required • Still no optimality proof • Potentially long run times	• Galinier & Hao (1999) (Tabu search + Genetic Algorithm) • Costa & Hertz (1997) (Ant Colony)
GNN-Based (Supervised)	• Leverage labeled data for high accuracy • Predict chromatic number or colorability • Automatic feature extraction	• Requires labeled datasets • Typically only classification (yes/no) • May not yield explicit color assignments	• Joshi et al. (2019) (GNN-Based) • Li et al. (2018) (GNN-Based)
GNN-Based (Unsupervised)	• No labels needed • Directly minimizes loss • Potentially good scalability	• Risk of local minima • Hyperparameter sensitive • No guarantee of global optimum	• Schuetz et al. (2022) (GNN-Based) • Lemos et al. (2019) (GNN-Based)
GNN-Based (Semi-Supervised)	• Uses both labeled and unlabeled data • Can reduce label requirements	• Dependent on partial labels • Potential overfitting on small labeled subsets	• Kipf & Welling (2017) (GCN for semi-supervised node classification)
Reinforcement Learning	• Learns coloring policy incrementally • No large labeled sets needed • Adaptable to different graph families	• Possible slow convergence • Performance depends on reward design • Balancing exploration–exploitation	• Zhou et al. (2016) (RL-Based) • Mazyavkina et al. (2021) (RL-Based)

2.4 Deep Learning for Graph-Structured Data

Deep learning has shown remarkable progress in tasks involving unstructured data such as images, text, and audio (LeCun, Bengio & Hinton, 2015). However, many real-world problems are naturally explained by relational structures, which are best represented using graphs. Traditional deep learning frameworks are typically designed for regular grids of data such as pixels in images. They are not directly suitable for data that do not follow a fixed grid or sequential order, such as graphs (Bronstein, Bruna, LeCun, Szlam & Vandergheynst, 2017). This gap is filled with the development of special neural network architectures, commonly referred to as Graph Neural Networks (GNNs)(Zhou, Cui, Hu, Zhang, Yang, Liu, Wang, Li & Sun, 2020). GNNs capture graph topology by leveraging neighborhood information with iteratively updating node representations.

2.4.1 Graph Neural Networks

The first idea of GNNs was introduced by Gori, Monfardini & Scarselli (2005), then it was further developed by Scarselli, Gori, Tsoi, Hagenbuchner & Monfardini (2009), where the main idea was learning node representations through repeated message passing between neighboring nodes. In these approaches, each node's hidden state in a graph is iteratively updated by aggregating (e.g. by summation, averaging, or pooling) information from its neighbors, with the hope of capturing topological and feature-based context.

Modern breakthroughs in GNNs mainly continue the core idea, but introduce new architectures that are more efficient, stable, and expressive. Fundamental architectures are Graph Convolutional Networks (GCNs), GraphSAGE, and Graph Attention Networks (GATs/GANs).

Graph Convolutional Networks (GCNs) was introduced by Bruna, Zaremba, Szlam & LeCun (2014) and developed by Kipf & Welling (2017). As the name suggests, it contains convolution layers like those in the Convolutional Neural Networks (CNNs). The main difference between CNNs and GCNs comes from CNNs' inability of operating with irregular or non-Euclidean structured data, such as graph data (Khemani, Patil, Kotecha & Tanwar, 2024). Therefore, the main idea of GCNs is performing convolutional operations in graph data. Although GCNs are transductive, they have shown good performance in node classification tasks in a semi-supervised learning setting, as demonstrated by Kipf & Welling (2017). Transductive models are trained with the entire graph, including unlabeled test nodes whose labels remain hidden.

The GraphSAGE architecture focuses on inductive learning and was introduced in Hamilton, Ying & Leskovec (2017). In inductive learning, a sample of the data is sufficient to infer embeddings for previously unseen nodes or graphs. Thanks to sampling a fixed number of neighbors, the architecture can scale to very large graphs. The GraphSAGE architecture mainly consists of two stages. First, it samples a fixed number of neighbors. Second, it aggregates the embeddings of the neighboring nodes. In this architecture, four types of aggregators are used: the Simple Neighborhood Aggregator, Mean Aggregator, Long short-term memory (LSTM) Aggregator, and Pooling Aggregator (Khemani et al., 2024).

Attention mechanism was brought to Graph Neural Network community by Veličković, Cucurull, Casanova, Romero, Liò & Bengio (2018). Attention mechanism in GATs uses learnable attention coefficients for each edge compared to uniform or fixed-weight aggregation in GCNs or GraphSAGE. This can be more expressive in some areas than the fixed weight approach in GCNs or GraphSAGE, but it also adds computational complexity.

The attention mechanism is well suited for the TSP because it focuses on the most relevant neighbors while giving less weight to irrelevant ones at different stages of the decision process. GAT performs well in this setting, because it dynamically learns which cities are more important to consider at each decision step (Kool, van Hoof & Welling, 2019).

Table 2.2 compares these architectures with respect to their key advantages and limitations.

Table 2.2 Comparison of Graph Neural Network Architectures

Architecture	Advantages	Limitations	Key Reference
GCN	• Simple and efficient design • Leverages full graph structure for semi-supervised tasks • Effective for node classification	• Transductive; limited generalization to unseen nodes • Requires full adjacency matrix (can be memory intensive)	Kipf & Welling (2017)
GraphSAGE	• Inductive capability; generalizes to new, unseen nodes • Scalable via neighborhood sampling • Flexible aggregation functions (mean, LSTM, pooling)	• Sampling might omit critical neighbor information • Aggregation functions may oversimplify complex interactions	Hamilton et al. (2017)
GAT	• Adaptive weighting using attention mechanisms • Handles heterogeneous neighbor importance effectively	• Increased computational cost due to attention computations • More parameters may require larger datasets • Complexity can reduce interpretability	Veličković et al. (2018)

2.4.2 GNNs for Combinatorial Optimization / Vertex Coloring

With recent developments in GNNs, researchers have applied them to a variety of combinatorial optimization problems. For example, GNNs have been applied to the

traveling salesman problem (TSP), Max-Cut (Dai, Khalil, Zhang, Dilkina & Song, 2018; Schuetz et al., 2022), and vertex coloring (Schuetz et al., 2022). As discussed earlier, in Section 2.2, these problems require searching through an exponentially large solution space to find an optimal or near-optimal solution. GNNs are well-suited to these problems for two reasons. Firstly, they can scale to very large graphs. Secondly, they can operate on irregular graph structures.

The success of GNNs in other combinatorial optimization tasks led to the leveraging of the representational power of GNNs especially for vertex coloring. Recent studies have shown an increasing interest in solving vertex coloring problems using GNNs (Colantonio, Cacioppo, Scarpati & Giagu, 2024; Lemos et al., 2019; Li, Li, Ma, Chan, Pan & Yu, 2022; Pugacheva, Ermakov, Lyskov, Makarov & Zotov, 2024; Schuetz et al., 2022; Wang, Yan & Jin, 2023; Zhang, Zhang & Wu, 2024). GNN-based vertex coloring approaches can be categorized into two groups: direct node classification and graph classification.

One way to approach the vertex coloring problem is to treat the problem as a **multi-class node-classification** task. In this method, each node is assigned to one of the possible colors based on node features and a message passing mechanism (Colantonio et al., 2024; Pugacheva et al., 2024; Schuetz et al., 2022). Specifically, GNNs learn an embedding (i.e. vector representation of nodes) for each node's local structural information, such as the neighborhood structure and degree, as well as any node-related attributes, and then predict a color label for each node.

General steps for direct node classification includes:

- **Feature Encoding:** In a graph, each node is represented by features such as degree or centrality measures. These features can be fed to GNN's initial or continuous input layer and then transformed through GNN layers, such as GCNs (Kipf & Welling, 2017), GATs (Veličković et al., 2018) or GraphSAGE (Hamilton et al., 2017). A study shows that GNNs perform well when there is a strong correlation between node features and labels (Duong, Hoang, Dang, Nguyen & Aberer, 2019). Another way is not using feature encoding and letting the GNN layers learn these representations by aggregating the neighborhood information (Schuetz et al., 2022).

- **Message Passing:** Message passing aims to capture the local structure of the input graph. It refers to updating the embedding of each node by aggregating information from its neighbors (Gilmer, Schoenholz, Riley, Vinyals & Dahl, 2017).

- **Color Prediction:** At the final layer of the GNN, the model outputs a prob-

ability distribution over a predefined set of colors for each of the nodes. Depending on the application, a softmax layer or a Gumbel-Softmax reparameterization Jang, Gu & Poole (2017) may be used to calculate the probability of being in the classes to determine the final coloring.

Although this strategy provides a simple way to learn color assignments, there are some limitations to the k-coloring problem of vertex coloring. The primary issue is that it can be trapped at a local optimum (Wang & Li, 2023). As a result, GNN predicts the same color for adjacent nodes, which is not desired for the k-coloring problem. Post-processing steps are often introduced to solve this issue (Schuetz et al., 2022).

Graph classification is another type of problem for which GNNs are used, where classification is done at a higher level. Rather than a direct assignment of colors to nodes, the GNN provides an embedding that represents the global structural properties of the graph. This embedding is then used to determine:

- **Whether a valid k-coloring exists** (for a given k) (Lemos et al., 2019).
- **What the minimal number of colors is required** for a valid coloring.

In these GNNs, graph pooling or global readout functions techniques are applied to node embeddings to provide a single vector that characterizes the entire graph (Liu, Zhan, Wu, Li, Du, Hu, Liu & Tao, 2023).

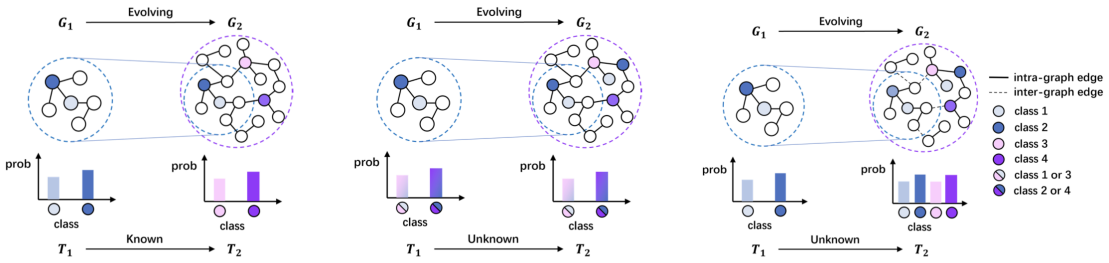
2.5 Incremental and Curriculum Learning in Graph-Based Tasks

Both Incremental Learning and Curriculum Learning offer methods to manage the complexity in the learning process of neural networks. In general, both approaches suggest that, rather than facing a task's full complexity all at once, one can structure a learning process into stages of challenges starting from an easy task to more challenging tasks.

2.5.1 Overview of Incremental Learning

Incremental Learning is known as continual learning or sequential learning. The method trains a model in stages that progressively introduces new data or tasks while keeping the learned knowledge from previous stages. This method is used to address the problem of catastrophic forgetting (van de Ven, Tuytelaars & Tolias, 2022; Zhou & Cao, 2021). In standard neural networks, when a new task or data distribution is added to the model, the model overwrites the representations learned earlier. Incremental learning can be categorized into three scenarios (van de Ven et al., 2022). Figure 2.1 depicts these three scenarios (Yuan, Guan, Ni, Luo, Man, Wong & Chang, 2023).

- **Task-Incremental Learning:** In this scenario, an algorithm sequentially learns a series of distinct tasks. For example, an athlete who learns to play tennis, then table tennis, and then padel experiences Task-Incremental Learning. Knowledge of one sport might be positively transferred to the other sport.
- **Domain-incremental learning:** In this scenario, an algorithm learns to solve the same task, but within different contexts. In other words, the input distribution changes over time, but the output space remains the same for the model. For example, the self-driving car AI system is continuously updated as it encounters various environmental conditions. It initially learns to navigate in clear weather, then in rain, and subsequently in snowy conditions. The core problem of driving remains constant, but the input data from its sensors is changed by the weather.
- **Class-incremental learning:** In this scenario, an algorithm incrementally learns to distinguish between an increasing number of classes that are labeled categories, such as 'tiger', 'dolphin' or 'monkey'. For example, a child first learns to identify common pets, such as cats and dogs. Later, he/she learns to distinguish cows and horses as well. The child is then expected to distinguish between all animals that he/she has learned so far.



(a) Task-incremental (b) Domain-incremental (c) Class-incremental

Figure 2.1 Illustration of Incremental Learning scenarios on evolving graphs: (a) Task-incremental, (b) Domain-incremental, and (c) Class-incremental.

In graph-based learning, Incremental Learning updates the GNN when new nodes or edges are added to the graph, while keeping the knowledge from previous training. Incremental Learning for graph based learning is still relatively new, but there are some existing works (Kim, Yun & Kang, 2022; Su, Zou, Zhang & Wu, 2024; Xu, Zhang, Guo, Guo, Tang & Coates, 2020; Yuan et al., 2023; Zhang, Song & Tao, 2022; Zhou & Cao, 2021). Incremental graph learning methods have approaches such as **regularization-based** that applies constraints on changing important model parameters, **replay-based** that stores some form of past task data and retraining the model to consolidate previous knowledge and **architecture-based** that adapts the neural architecture dynamically with previously learned representations (Yuan et al., 2023).

2.5.2 Overview of Curriculum Learning

Curriculum learning is inspired by human education; Bengio, Louradour, Collobert & Weston (2009) describe Curriculum Learning as training a model on easier tasks or data first and gradually moving to more difficult ones. This stepwise approach can yield better performance because the model is not overwhelmed by the full complexity of the problem in the early steps. One of the earlier examples of this method is Elman (1993). In this work, it is demonstrated that training a neural network on simpler sequences before harder ones improves learning of complex grammar.

In order to apply Curriculum Learning to graph-based tasks, one must define what constitutes easy and hard instances on graph examples. A straightforward approach could consider a graph with lower density, fewer nodes, and less structural complexity as easy, while a higher number of nodes, higher density, and more complex structural graphs could be classified as hard. A curriculum could therefore train a GNN on small or less dense graphs first and progressively increase the size or density of the graph during training. Recently, more complex difficulty measurements were developed. For instance, Vakil & Amiri (2023) defines graph difficulty using measures such as node degree, average neighbor degree, and Katz centrality. This study shows that multiview competency-based curriculum for GNNs achieves notable improvements in final accuracy over standard training on node and link prediction tasks by prioritizing simpler substructures during early epochs.

2.5.3 Comparison with the Proposed Subgraph Expansion Method

The proposed approach (Section 3.3) takes inspiration from the core ideas of both Incremental Learning and Curriculum Learning, but there are some key differences, as well as some similarities.

The first similarity is **the increase in gradual complexity**. Similarly to the Curriculum Learning, the proposed approach starts with an easy version of the problem by using a smaller subgraph of the input graph and later introduces more challenging versions of the problem by adding more nodes and edges to the graph. The second similarity is **knowledge retention**. Like Incremental Learning, the model keeps useful information such as model weights and node embeddings from earlier stages (previous subgraphs) when transitioning to larger subgraphs. The third similarity is the use of a **structured training flow**. As in classical Incremental Learning and typical Curriculum Learning, the proposed approach uses a staged or sequential training procedure, unlike classical training methods, which tackle the entire task at once.

Apart from these similarities with Incremental Learning and Curriculum Learning, there is one key distinction of the proposed approach, which is applying the core ideas of both methods into a single instance of graph coloring rather than across a distribution of instances. This can be seen as a form of search strategy, where the search space is expanded step by step. This stepwise expansion strategy is reminiscent of the divide and conquer approach, because the proposed approach of dividing the problem into a sequence of manageable pieces with increasing size and complexity. Thus, the proposed approach leverages the foundations of Incremental and Curriculum Learning, but is designed for unsupervised learning.

2.6 Summary of the Literature

2.6.1 Key Observations and Trends

Vertex coloring aims to assign a color to each vertex of a graph such that no two adjacent vertices share the same color. Deciding whether a graph can be colored with k colors is NP-complete, and finding the minimum number of colors without any conflicts is NP-hard (Garey & Johnson, 1990). Vertex coloring has broad applications such as scheduling, timetabling, resource allocation, and register allocation in compilers.

Over the years, classical operations research methods have been developed to solve the vertex coloring problem. Exact algorithms (e.g. dynamic programming, branch-and-bound) can find optimal colorings with the trade-off of failing to solve large graphs due to their exponential time complexity. To handle larger graph sizes, researchers have used heuristics and metaheuristics (e.g. greedy heuristics, simulated annealing, tabu search, genetic and evolutionary algorithms). Greedy methods run in polynomial time and scale well, but they often can not produce an optimal solution. Metaheuristic approaches have shown better performance in coloring quality compared to heuristic methods, but still they often produce suboptimal solutions and scale worse than modern neural network techniques.

In recent years, research tended to shift towards GNNs and other deep learning techniques to address combinatorial problems (Colantonio et al., 2024; Dai et al., 2018; Joshi et al., 2019; Lemos et al., 2019; Li et al., 2022,1; Pugacheva et al., 2024; Schuetz et al., 2022,2; Wang et al., 2023). GNNs are similar to the CNNs but they are designed to operate on graph-structured data. They use a message-passing mechanism to aggregate information from each node's neighbors so that the model learns representations that capture the graph's structure and context (Khemani et al., 2024). GNNs have gained popularity because they are more scalable than traditional algorithms. Applying GNNs to vertex coloring problems has multiple forms, such as Node Classification that trains a model to assign a color to each node (Schuetz et al., 2022), and Graph Classification that trains a model to predict whether a graph is colorable with k colors (Lemos et al., 2019). These GNN-based approaches have often shown good results on coloring nodes or estimating colorability of a graph on large or complex graphs.

Some researchers tried to combine some ideas from Curriculum Learning and Incremental Learning with graph neural networks (Kim et al., 2022; Su et al., 2024; Vakil & Amiri, 2023; Xu et al., 2020; Yuan et al., 2023; Zhang et al., 2022; Zhou & Cao, 2021). Curriculum learning (Section 2.5.2) is training a model on easier tasks or data first and gradually moving to more difficult ones (Bengio et al., 2009). One problem in Curriculum Learning for graph-based learning is finding easy and hard tasks on graph examples. Similarly, Incremental Learning (Section 2.5.1) trains a

model in multiple stages, where each stage comes with new data or tasks while keeping the knowledge from previous stage (Yuan et al., 2023). Both Curriculum Learning and Incremental Learning aim to solve issues arising from large-scale or dynamically changing graphs such as catastrophic forgetting and unstable convergence by systematically breaking down the problem.

2.6.2 Research Gaps Identified

Although the vertex coloring problem has solutions in both classical and learning-based approaches, some research gaps remain evident in vertex coloring. Primary challenges are scalability and optimality. There is no perfect approach that balances optimality and scalability perfectly. Even though classical exact approaches are good for finding optimal solutions, they suffer from scalability due to computational complexity. Heuristics and metaheuristics scale better, but do not guarantee an optimal solution. Similarly, learning-based approaches come with their own set of limitations. Reinforcement learning approaches require careful design of a reward mechanism, and they can become time consuming or unstable on large graphs.

Schuetz et al. (2022) introduced an unsupervised Physics-Inspired GNN for vertex coloring. This approach has two key advantages. First, because it is unsupervised, it does not require labeled training data. Second, it can scale to graphs that have millions of vertices, and it matches or outperforms the classical solvers. However, Schuetz et al.'s work also has key limitations. In experiments, the GraphSAGE variant improved solution quality but also increased the training times to ($\sim$ 1–8 hours) from ($\sim$ 0.167–2 hours) for the simpler Graph Convolutional variant on COLOR graphs. Moreover, the raw GNN output of the Schuetz et al.'s work contained a few color conflicts that can be solved with their post-processing heuristic. Also, Wang & Li mentioned that Schuetz et al.'s work tends to trapped into a local minima. In summary, Schuetz et al. (2022) demonstrates the promise of unsupervised GNN coloring in terms of scalability, but its solution quality and speed leave room for improvement.

Subsequent works have addressed parts of these limitations. For example, Wang et al. (2023) introduced negative message passing into the GNN architecture for more effective information exchange and proposed a new loss function to accelerate the learning process, achieving numerical improvements over the original PI-GNN. Pugacheva et al. (2024) developed the QRF-GNN, which uses recurrently updated features to help the GNN escape suboptimal local minima. Similarly, Zhang et al.

(2024) used GNN and then performed post-processing with a metaheuristic to overcome local minima. Overall, these studies close parts of the gap, but the trade-off between achieving optimal coloring, computational speed, and scalability to massive graphs remains an active area of research.

Although Curriculum Learning and Incremental Learning have been recognized as beneficial in principle, GNN-based vertex coloring studies have not explicitly applied Curriculum Learning or Incremental Learning. In this study, it was thought that incorporating a strategy based on Curriculum Learning or Incremental Learning could be beneficial. In summary, it was seen that there is a clear research gap in scalability and optimality for the vertex coloring problem. Addressing this gap could involve combining the strengths of GNNs with Curriculum Learning or Incremental Learning, which is an area that the current literature has only begun to explore for GNNs.

2.6.3 Implications for This Thesis

The trends and gaps in the literature for vertex coloring show the direction of this thesis. The thesis proposes a subgraph expansion strategy, which resembles Curriculum Learning and Incremental Learning, for the vertex coloring problem using GNNs. The approach trains the graph in stages rather than training the full graph in one leap. The model is first trained on a smaller subgraph of the full graph and gradually expands it until the full graph. By expanding the scope of the problem incrementally, the method also manages the complexity of the graph incrementally. This progressive expansion acts as a curriculum because the GNN learns on simpler configurations before tackling the full problem. The approach aims to address the scalability and better color performance issues discussed above. The method limits the search space at each step because only one subgraph is colored. It helps to avoid the combinatorial explosion that a full graph presents. Also, the incremental training provides a smoother learning curve for the GNN, potentially reducing instability. By filling the gap in the literature for vertex coloring, this approach contributes a new perspective. Consequently, this strategy could generalize beyond the vertex coloring to other combinatorial optimization problems that would benefit from a curriculum or an incremental strategy.

3. METHODOLOGY

3.1 Overview of the Graph Coloring Framework

In this chapter, the proposed methodology developed to solve the vertex coloring problem using Graph Neural Networks (GNNs) is detailed. The proposed framework casts the vertex coloring problem as a multiclass classification task at node level, where each node is assigned a color or a label from a fixed set of colors. The proposed methodology is unsupervised and physics inspired, which does not require training labels and uses the POTTS model (a model from statistical physics) based loss function to minimize the number of color conflict.

The proposed framework consists of several components, which are described in the following sections. Through this framework, several concepts from deep learning, physics, curriculum learning, and incremental learning are integrated to address graph coloring problem in a novel way.

3.1.1 The Context of Our Approach within Recent GNN Work

K-coloring vertex coloring problem is NP-complete, and finding a minimum number of colors for a graph is NP-hard. Traditional methods tend to suffer from this complexity in larger graphs, and there is growing interest in applying GNNs to such combinatorial optimization tasks. Recent work shows that GNN-based approaches can match or even outperform in certain areas (Pugacheva et al., 2024; Schuetz et al., 2022), because GNNs naturally process relational (graph) data, scale well and capture structural patterns through message passing. As illustrated in Figure

3.1, the message passing step updates the representation of each node by gathering information from its neighbors and from the node itself (Schuetz et al., 2022).

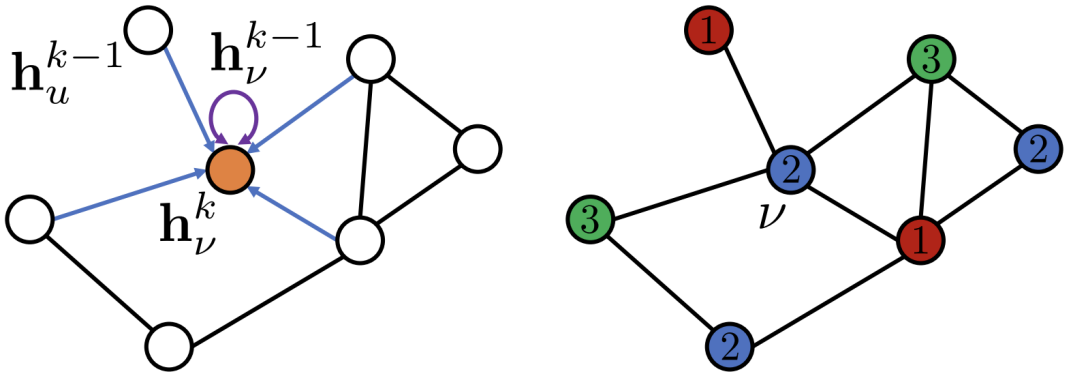


Figure 3.1 Illustration of message passing in Graph Neural Networks (GNNs). The left figure shows the message passing mechanism at the k^{th} layer, where the hidden representation $\mathbf{h}_\nu^k$ of node ν (orange node) is updated by aggregating messages from its neighbors u using their $(k-1)^{\text{th}}$-layer representations $\mathbf{h}_u^{k-1}$. A self-loop is used to incorporate the node's own representation $\mathbf{h}_\nu^{k-1}$. The right figure shows after the color assignment, where each node is assigned a color or a number learned via the GNN.

The proposed method is inspired by Schuetz et al. (2022), which introduced a physics-inspired GNN for vertex coloring that treats it as an unsupervised multiclass classification task at the node level. Their results showed that GNN can produce colorings on par or better with greedy coloring and tabu search. As in Schuetz et al. (2022), the proposed approach is aligned with one that uses unsupervised learning and optimizes a continuous relaxation of the Potts loss compared to the other machine learning method. Moreover, inspired by the perturbation strategy of Colantonio et al. (2024), noise is injected into the node embeddings during training. However, the noise model is different from Colantonio et al. (2024)'s work (see Chapter 3.6.1 for details). In addition, curriculum learning and incremental learning have been explored to improve GNN training. The idea is to feed the model easier sub-problem first, so that it can learn gradually, and more easily, before coloring the entire graph. The proposed approach is similar to curriculum or incremental learning, since it starts with a subgraph of the full graph and expands it until the full graph using specific expansion strategies such as Breadth-First-Search (BFS) or random walk. To the best of our knowledge, this is a novel approach in the context of graph coloring.

In summary, the proposed method combines several elements, such as unsupervised GNN training for combinatorial optimization, physics-inspired loss functions, and

curriculum learning or incremental learning for graphs, to overcome some challenges for vertex coloring using GNN such as the difficulty of training GNNs on large, hard instances from scratch, or getting the model stuck in poor local optima due to the discrete symmetry of color permutations.

3.1.2 Our Contribution

The proposed approach introduces a curriculum learning (incremental learning) strategy specifically for vertex coloring. Instead of training the full graph, it generates a sequence of subgraphs that end up being the full graph. Three expansion methods (layer-by-layer BFS expansion, degree-first BFS expansion, and random-walk expansion) are proposed to create subgraphs for the model. This strategy allows GNN to learn how to color small portions of the graph before facing the complexity of the full graph. As a result, stability of the training and solution quality are expected to be improved compared to the training the full graph. In addition, this approach is expected to reduce the overall training time. Curriculum learning (or incremental learning) in this manner is relatively novel for GNNs and, to the best of our knowledge, has not been applied to graph coloring before.

3.2 Data Loading and Graph Preprocessing

The handling of graph-structured data is the first step in the proposed method. This section details how the input graph is loaded, processed, and how features are extracted before entering the model.

3.2.1 Input Graph

The input for the proposed approach is a graph that needs to be colored. This study focuses on undirected and unweighted graphs. Each node represents an entity to be colored, and each edge represents conflict constraints. All experiments used

instances from the COLOR benchmark dataset, which was described in Section 4.1.

3.3 Subgraph Expansion and Incremental Training

The main contribution of our methodology is the use of subgraph expansion in training. Rather than presenting the full graph to the model, the proposed method divides the graph into a series of small graphs of the full graph. In this way, the model first starts with easy problems, then progressively tackles the harder ones like in curriculum learning. The method also resembles incremental learning by progressively introducing new data while keeping the learned knowledge.

Three strategies were implemented for subgraph expansion: Layer-by-layer BFS-based expansion, degree-first BFS-based expansion, and random walk expansion. All strategies aim to generate a sequence of subgraphs.

$$G_1 \subset G_2 \subset \cdots \subset G_T = G_{\text{full}},$$

where G_T is the full graph. At each step, the model is trained on the current subgraph, and its weights and node embeddings are transferred to the next step as the subgraph grows. This incremental training continues until the subgraph equals the full graph. As a result, the model always has experience with the previous stages, rather than learning the completely full graph without any experience. This approach is analogous to solving the minesweeper by a human. It starts with a small part of the area and then gradually introduces more area. The minesweeper game is the inspiration for this method.

In the following, expansion strategies are detailed, that is, how the subgraphs are generated and how the training proceeds.

3.3.1 Layer-by-Layer BFS-Based Expansion

Layer-by-Layer Breadth-First Search (BFS) strategy grows the subgraph starting from a starting node. The starting node or seed node v_0 is randomly selected. The initial subgraph G_1 contains only v_0. The next subgraph G_2 is formed by adding

v_0's immediate neighbors and all edges among them. Then, to get G_3, all nodes at distance 2 from v_0 (neighbors of neighbors) are added, and include all edges among all nodes. In general, at step k, it includes all nodes that are within $k-1$ hops from the seed v_0. This process continues until it reaches all nodes in the graph. An example of the layer-by-layer BFS expansion is shown in Figure 3.2. The pseudocode can be seen in the Algorithm 1.

Although the algorithm is simple, layer-by-Layer BFS-based expansion offers *no control over the number of nodes* of each subgraph because the layer sizes depend only on the neighbors at that BFS depth with respect to the seed. In heterogeneous graphs, one layer may contain $\ll 1\%$ of the nodes and the next layer 40% with respect to the seed. Such variability causes oscillations in the model load between over-loading and under-loading depending on the local structure of the graph. The *degree-first* variant below is designed to overcome this problem.

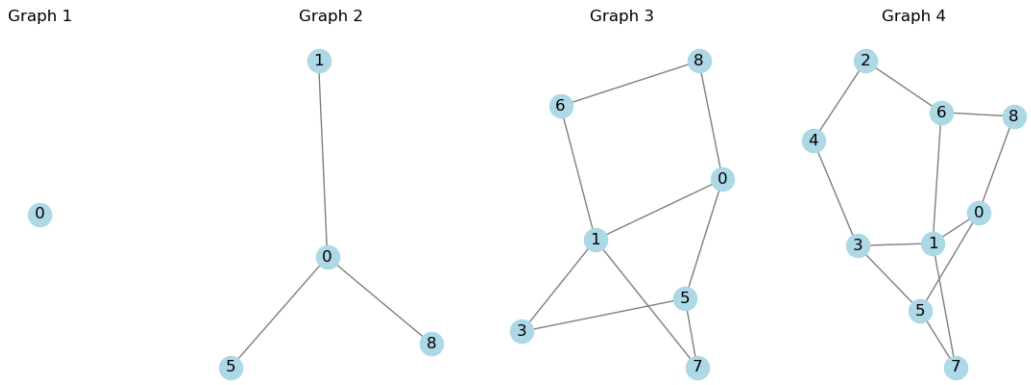


Figure 3.2 **Layer-by-layer Breadth-First Search (BFS) expansion.** Starting from a randomly selected seed node v_0 (node 0), G_1 contains only v_0 (Graph 1). Each subsequent graph G_k adds all nodes that are within $(k-1)$ hops of v_0 and *all* edges connected by those vertices. The process terminates once every node of the original graph is reached (Graph 4).

Algorithm 1: Layer-by-layer BFS Expansion to Generate Subgraphs

Input: $G = (V, E)$ — undirected graph,;
$s \in V$ — start node

Output: $\mathcal{S} = \langle G_0, G_1, \ldots, G_L \rangle$ — list of progressively larger subgraphs

```
1  // 1. Assign BFS layers
2  visited ← { s ↦ 0 };
3  queue ← ⟨s⟩;
4  while queue ≠ ∅ do                                   // standard BFS
5      v ← queue.pop();
6      ℓ ← visited[v];
7      layers[ℓ].append(v);
8      foreach u ∈ Adj(v) do
9          if u ∉ visited then
10             visited[u] ← ℓ + 1;
11             queue.push(u);
12 L ← max{visited[v] | v ∈ V};
13 // 2. Build subgraph sequence, one layer at a time
14 included ← ∅;
15 S ← ⟨⟩;
16 for ℓ ← 0 to L do
17     included ← included ∪ layers[ℓ];
18     G_ℓ ← G[included] ;                  // induced subgraph on current nodes
19     S.append(G_ℓ);
20 return S;
```

Notations (Algorithm 1):

- $G = (V, E)$: input undirected graph with vertex set V and edge set E.
- $s \in V$: start (root) node of the breadth-first search.
- $\mathrm{Adj}(v)$: set of neighbours (adjacent vertices) of node v in G.
- visited: map $v \mapsto \ell$ assigning each discovered node v to its BFS layer index ℓ.
- queue: first-in–first-out (FIFO) queue that holds the current BFS frontier.
- layers[ℓ]: list of vertices that belong to layer ℓ.
- L: deepest layer reached by the BFS; $L = \max_{v \in V} \mathrm{visited}[v]$.
- included: cumulative set of vertices already added to the growing subgraph sequence.

- G_ℓ: induced subgraph G[included] after processing layer ℓ.
- $\mathcal{S} = \langle G_0, G_1, \ldots, G_L \rangle$: ordered list of progressively larger subgraphs returned by the algorithm.

3.3.2 Degree-first BFS-Based Expansion

Degree-first BFS strategy differs from Layer-by-layer BFS expansion in two ways. First, the selection of the seed (root) node is not random, it selects as highest degree node in the graph. Secondly, the size of the subgraphs does not depend on the layer size, it can be determined by a parameter named step percentage ρ. The step percentage ρ determines the number of nodes that will be added at each stage with respect to the total number of nodes. After selecting the highest-degree node as the seed, the method produces a single global ordering O of vertices obtained from BFS whose layers are internally sorted by degree descending and appends all layers to a single list. The stages are formed by first $\lceil \rho n \rceil$, $2\lceil \rho n \rceil$, $3\lceil \rho n \rceil$, $\ldots$ vertices of O (with $n = |V|$). By sorting vertices in descending degree order, the method forces the model to learn feasible colors for hubs before it is distracted by sparsely connected leaves. This is expected to yield fewer color swaps in later stages and faster convergence to a zero-conflict solution. Figures 3.3 and 3.4 visualize this process on a nine-node toy graph. The pseudocode can be seen in the Algorithm 2.

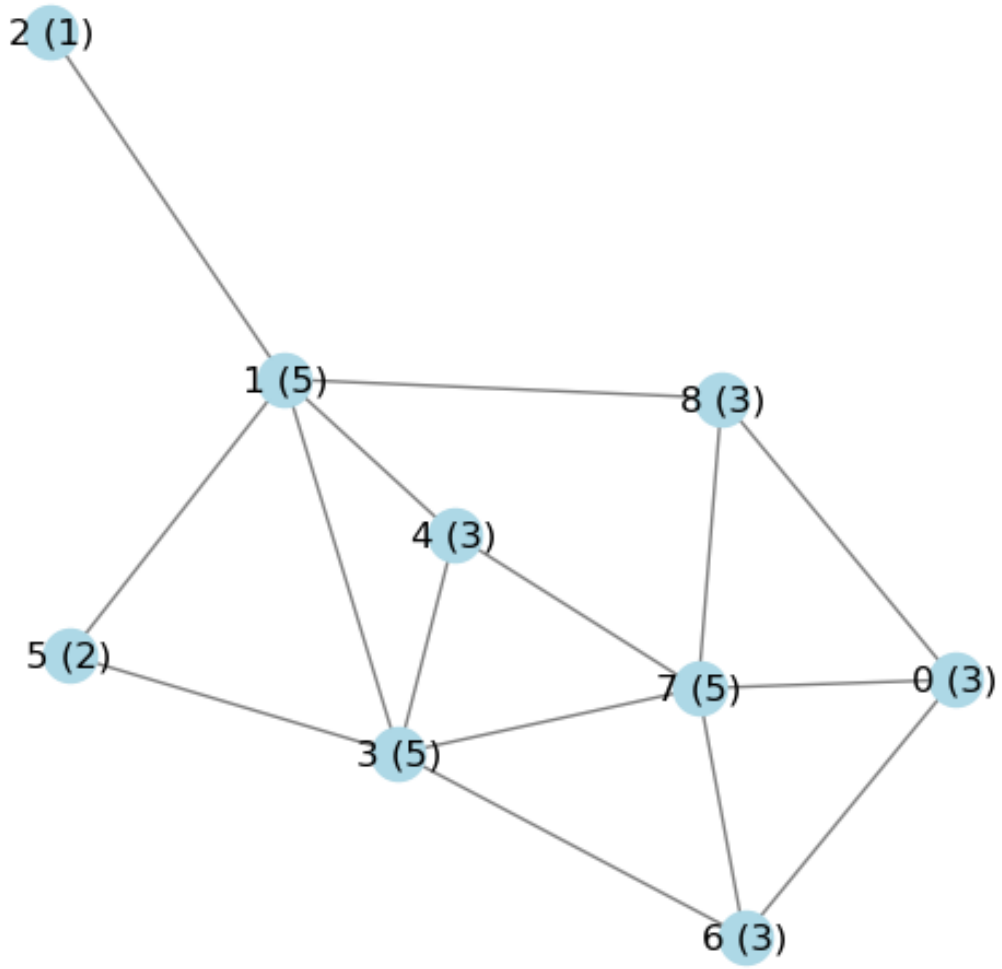


Figure 3.3 **Reference graph with degree annotations**. Each node is labelled *id (degree)*.

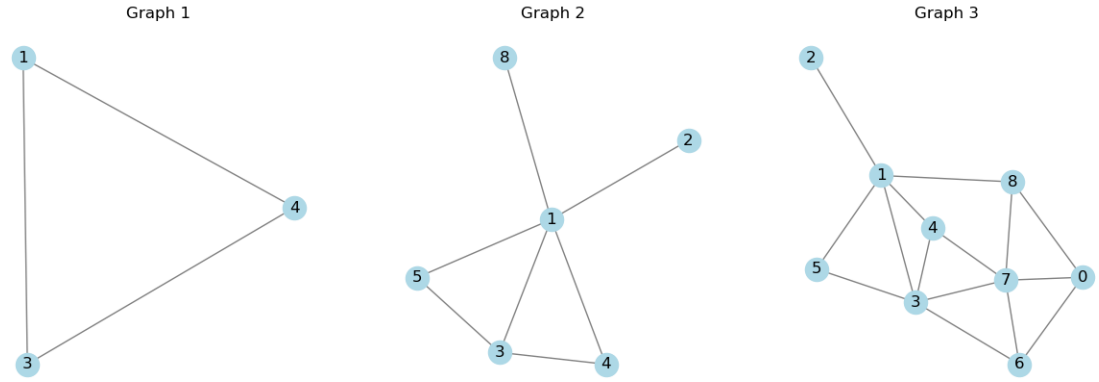


Figure 3.4 **Three curriculum stages produced by the degree-first BFS expansion with step percentage $\rho = 0.33$.**

Algorithm 2: Degree-based BFS Expansion to Generate Subgraphs

Input: $G = (V, E)$ — undirected graph, $\rho = 0.3$ — step percentage (e.g. 0.1 for 10%)

Output: $\mathcal{S} = \langle G_1, G_2, \ldots, G_T \rangle$ — list of progressively larger induced subgraphs

// 1. Select start node by maximum degree
$\deg \leftarrow \{ v \mapsto \deg_G(v) \mid v \in V \}$;
$s \leftarrow \arg\max_v \deg[v]$; `// seed BFS at the highest-degree node`
// 2. Perform BFS layering from start node
visited $\leftarrow \{ s \mapsto 0 \}$;
queue $\leftarrow \langle s \rangle$;
layers $\leftarrow \{\}$;
while queue $\neq \emptyset$ **do**
    $v \leftarrow$ queue.pop();
    $\ell \leftarrow$ visited$[v]$;
    layers$[\ell]$.append(v);
    foreach $u \in \mathrm{Adj}_G(v)$ **do**
        if $u \notin$ visited **then**
            visited$[u] \leftarrow \ell + 1$;
            queue.push(u);
$L \leftarrow \max(\text{visited}[v] \mid v \in V)$;
// 3. Build ordered BFS list sorted by degree within each layer
bfs_order $\leftarrow []$;
for $\ell \leftarrow 0$ **to** L **do**
    nodes $\leftarrow$ layers$[\ell]$;
    sort(nodes, by $\deg_G(\cdot)$ descending); `// highest-degree first`
    bfs_order.extend(nodes);
$n_{\text{total}} \leftarrow |\text{bfs\_order}|$;
// 4. Compute step size and generate subgraphs
step $\leftarrow \lceil n_{\text{total}} \times p \rceil$;
$\mathcal{S} \leftarrow \langle \rangle$;
for $i \leftarrow$ step; $i < n_{total}$; $i \mathrel{+}= $ step **do**
    prefix $\leftarrow$ bfs_order$[1..i]$;
    $G_i \leftarrow G[\text{prefix}]$; `// induced subgraph on first` i `nodes`
    $\mathcal{S}$.append(G_i);
`// ensure full component appears last;`
$G_T \leftarrow G[\text{bfs\_order}]$;
$\mathcal{S}$.append(G_T);
return $\mathcal{S}$

Notations (Algorithm 2):

- $G = (V, E)$: input undirected graph with vertex set V and edge set E.
- $\deg_G(v)$: (unweighted) degree of vertex v in G.
- ρ (or p): step percentage; fraction of vertices to add at each expansion round.
- $s = \arg\max_{v \in V} \deg_G(v)$: start node chosen as the highest–degree vertex.
- $\mathrm{Adj}_G(v)$: set of neighbours of v.
- visited: map $v \mapsto \ell$ giving the BFS layer index ℓ of each discovered vertex.
- queue: FIFO queue holding the current BFS frontier.
- layers[ℓ]: list of vertices that belong to BFS layer ℓ.
- $L = \max_{v \in V} \mathrm{visited}[v]$: deepest layer reached by the BFS.
- bfs_order: concatenated list of all vertices, layer by layer, each layer sorted in descending degree.
- $n_{\text{total}} = |\text{bfs\_order}|$: total number of vertices in the connected component reached from s.
- $\text{step} = \lceil n_{\text{total}} \rho \rceil$: number of new vertices added at each expansion step.
- prefix[1..i]: first i vertices in bfs_order.
- G_i: induced subgraph $G[\text{prefix}[1..i]]$ containing the first i vertices.
- G_T: final induced subgraph containing all vertices in bfs_order (i.e. the full component).
- $\mathcal{S} = \langle G_1, G_2, \ldots, G_T \rangle$: ordered sequence of progressively larger induced subgraphs returned by the algorithm.

3.3.3 Random Walk Expansion

Random walk expansion mimics the random walk, but differs in preferring the unvisited neighbors and guaranteeing the eventual coverage of entire graph. The selection of the seed node is random as in the Layer-by-Layer BFS expansion, and the size of the subgraphs is determined by the step percentage ρ as in the degree-first BFS

expansion. Compared with BFS expansion methods, random walk expansion adds nodes in depth first rather than in concentric shells around the seed. Figure 3.5 illustrates how the random walk expansion gradually exposes the model to increasing structural complexity. The pseudocode can be seen in the Algorithm 3.

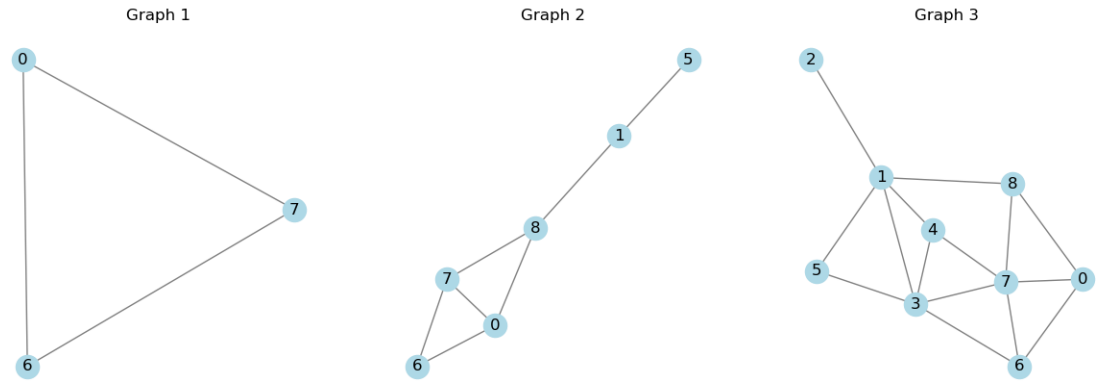


Figure 3.5 **Three curriculum stages produced by the random walk expansion with step percentage** $\rho = 0.3$**.**

Algorithm 3: Random-Walk Expansion to Generate Subgraphs

Input: $G = (V, E)$ — undirected graph, $\rho = 0.3$ — step percentage, r — random seed

Output: $\mathcal{S} = \langle G_1, G_2, \ldots, G_T \rangle$ — list of progressively larger *connected* subgraphs

// 1. Initialise random walk
setSeed(r);
$s \leftarrow \text{UniformRandom}(V)$; `// start node`
visited $\leftarrow \{s\}$; walk $\leftarrow \langle s \rangle$;
// 2. Biased random walk until every node is visited
while |visited| $< |V|$ **do**
  $v \leftarrow$ walk.last();
  $\mathcal{N} \leftarrow \text{Adj}_G(v)$;
  $\mathcal{N}_{\text{new}} \leftarrow \{u \in \mathcal{N} \mid u \notin \text{visited}\}$;
  if $\mathcal{N}_{\text{new}} \neq \varnothing$ **then**
    $w \leftarrow \text{UniformRandom}(\mathcal{N}_{\text{new}})$;
    ; `// prefer unvisited`
  else if $\mathcal{N} \neq \varnothing$ **then**
    $w \leftarrow \text{UniformRandom}(\mathcal{N})$;
    ; `// fallback to any neighbour`
  else
    $U \leftarrow V \setminus$ visited; $w \leftarrow \text{UniformRandom}(U)$;
    ; `// isolated restart`
  walk.append(w); visited.add(w);
// 3. Incremental subgraph construction
$k \leftarrow \lceil |V| \cdot p \rceil$; `// nodes per expansion step`
included $\leftarrow \varnothing$; $i \leftarrow 0$; $\mathcal{S} \leftarrow \langle\rangle$;
while |included| $< |V|$ **do**
  $\Delta \leftarrow \varnothing$; `// nodes to add this round`
  while $|\Delta| < k$ **and** $i <$ |walk| **do**
    $u \leftarrow$ walk$[i]$; $i \leftarrow i + 1$;
    if $u \notin$ included **then**
      Δ.add(u)
  if $\Delta = \varnothing$ **then**
    break
  included $\leftarrow$ included $\cup\, \Delta$;
  $H \leftarrow G$[included] ; `// induced subgraph`
  if !isConnected(H) **then** `// keep only largest component`
    $C^\star \leftarrow \arg\max_{C \in \text{Components}(H)} |C|$;
    $H \leftarrow G[C^\star]$;
  $\mathcal{S}$.append(H);
return $\mathcal{S}$

Notations (Algorithm 3):

- $G = (V, E)$: input undirected graph with vertex set V and edge set E.

- ρ (or p): step percentage; fraction of vertices added at each expansion round.
- r: random seed that fixes the pseudo-random number generator.
- $s \in V$: start node selected uniformly at random.
- $\mathrm{Adj}_G(v)$: neighbor set of vertex v.
- walk: ordered list $\langle v_0, v_1, \dots \rangle$ of vertices visited by the biased random walk.
- visited: set of vertices already encountered by the walk.
- $\mathcal{N}$, $\mathcal{N}_{\text{new}}$: respectively the full and the unvisited neighbor sets of the current vertex.
- $k = \lceil |V|\rho \rceil$: number of new vertices to add at each subgraph expansion step.
- Δ: batch of vertices selected from the walk and added during the current expansion step.
- included: cumulative set of vertices present in the growing subgraph.
- H: induced subgraph G[included]; trimmed to its largest connected component if necessary.
- $C^\star$: vertex set of the largest connected component of H when H is disconnected.
- $\mathcal{S} = \langle G_1, G_2, \dots, G_T \rangle$: ordered list of progressively larger connected subgraphs returned by the algorithm.

3.4 The GNN Architecture

After preparing the data and defining the subgraph expansion method, the next component is the model architecture. In this thesis, two Graph Neural Network architectures were used to solve the vertex coloring problem. Graph Convolutional Network (GCN) and GraphSAGE architectures, both are message-passing neural networks that iteratively propagate and aggregate information on edges of the graph.

3.4.1 GraphConv Model

The implemented GraphConv model is based on the standard Graph Convolutional Network (GCN) introduced by Kipf & Welling. The model is a straightforward feed-forward GNN architecture similar to the one used in Schuetz et al. (2022). In each graph convolutional layer, every node aggregates feature information from its neighbors, followed by a linear transformation. Let $h_i^{(l)}$ be the feature vector of node i at layer l. A GraphConv update can be written as

$$h_i^{(l+1)} = \sigma\Big(W^{(l)} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{d_i\, d_j}}\, h_j^{(l)}\Big),$$

where $\mathcal{N}(i)$ denotes the neighbors of node i, d_i is the degree of node i, and $W^{(l)}$ is a trainable weight matrix for layer l. The term $\frac{1}{\sqrt{d_i d_j}}$ is a normalization to account for varying node degrees, and σ is an activation function (ReLU in our case). In our implementation, the GraphConv model uses two such layers: the first layer maps the d-dimensional input embedding of each node to a hidden dimension H with ReLU activation, and the second layer maps from H to q output logit values (where q is the number of colors/classes). A dropout (with a rate typically around 0.1 to 0.3) is applied between the layers during training. The pseudocode for the GraphConv model is provided in Algorithm 4.

Algorithm 4: Two-Layer **GraphConv** Neural Network

Input: Graph $G = (V, E)$; node feature matrix $X \in \mathbb{R}^{|V| \times d_{in}}$;
weights $W^{(0)} \in \mathbb{R}^{d_{hidden} \times d_{in}}$, $W^{(1)} \in \mathbb{R}^{C \times d_{hidden}}$;
dropout rate p_{do}.

Output: Class-logits $\{z_i\}_{i \in V}$ (or probabilities after softmax).

foreach $i \in V$ **do** // initial node features
  $h_i^{(0)} \leftarrow X_{i,:}$
end

Layer 1:
foreach $i \in V$ **do**
  $m_i \leftarrow \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{d_i \, d_j}} \, h_j^{(0)}$ // message passing with normalized adjacency
  $h_i^{(1)} \leftarrow \text{ReLU}\big(W^{(0)} m_i\big)$ // linear transformation and activation
end

Dropout:
foreach $i \in V$ **do**
  $\tilde{h}_i^{(1)} \leftarrow \text{Dropout}\big(h_i^{(1)}, p_{\text{do}}\big)$ // randomly zero elements with prob. p_{do}
end

Layer 2:
foreach $i \in V$ **do**
  $m_i \leftarrow \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{d_i \, d_j}} \, \tilde{h}_j^{(1)}$ // aggregation from neighbors
  $z_i \leftarrow W^{(1)} m_i$ // final linear transformation
end
return $\{z_i\}_{i \in V}$ (optionally $p_i = \text{softmax}(z_i)$)

Notations (Algorithm 4)

- V: set of nodes; E: set of edges.
- d_i: degree of node i
- $\mathcal{N}(i)$: neighbor set of node i.
- d_{in}: dimension of input features.
- d_{hidden}: dimension of hidden layer.
- C: number of output classes.
- $h_i^{(l)}$: embedding of node i after layer l.

- m_i: aggregated neighborhood message for node i.

3.4.2 GraphSAGE Model

The Graph Sample and Aggregrate (GraphSAGE) model that was implemented in this study is based on the model introduced by Hamilton et al.. It is also similar to the one used in (Schuetz et al., 2022). Instead of full summation of neighbor features as in GraphConv, GraphSAGE uses a specified aggregator such as LSTM, mean, max-pooling or GCN. In our implementation, only LSTM and mean were used. In GraphSAGE, the aggregated message of the neighbors of each node and the node's own features are combined. For a node i, a GraphSAGE layer computes:

$$m_i^{(\ell)} \;=\; \mathrm{AGG}\Big\{\, h_j^{(\ell)} : j \in \mathcal{N}(i) \Big\},$$

$$h_i^{(\ell+1)} \;=\; \sigma\Big(W_{\mathrm{cat}}^{(\ell)} \big[\, h_i^{(\ell)} \,\|\, m_i^{(\ell)} \big] \Big),$$

where AGG denotes the aggregation function (mean or lstm in our case), || is concatenation, and $W_{\mathrm{cat}}^{(l)}$ is a trainable weight matrix. This formulation allows incorporating the node's own previous representation $h_i^{(l)}$ in the update. Similarly to the GraphConv model, a two-layer GraphSAGE was constructed. An input layer (with ReLU) and an output layer, with dropout in between. The hidden dimension H, embedding dimension d, and the number of output classes q are the same as in the GraphConv architecture for a fair comparison. By comparing them, we can evaluate the impact of the neighbor aggregation scheme on the coloring performance.

Algorithm 5: Two-Layer **GraphSAGE** Neural Network (mean & LSTM aggregators)

Input: Graph $G = (V, E)$; node feature matrix $X \in \mathbb{R}^{|V| \times d_{in}}$;
weight matrices $W^{(0)} \in \mathbb{R}^{d_{hidden} \times d_{in}}$, $W^{(1)} \in \mathbb{R}^{C \times d_{hidden}}$;
self-weight matrices $W^{(0)}_{\text{self}} \in \mathbb{R}^{d_{hidden} \times d_{in}}$, $W^{(1)}_{\text{self}} \in \mathbb{R}^{C \times d_{hidden}}$;
LSTM aggregator parameters Θ_{lstm};
dropout rate p_{do};
aggregation type agg_type $\in \{\texttt{mean}, \texttt{lstm}\}$.
Output: Node-level logits $\{z_i\}_{i \in V}$ (or $\{p_i\}$ after `softmax`)

foreach $i \in V$ **do** `// copy raw features`
  $h_i^{(0)} \leftarrow X_{i,:}$
end

Layer 1:
foreach $i \in V$ **do**
  if *agg_type* = ***mean*** **then**
    $m_i \leftarrow \text{MEAN}\big(\{h_j^{(0)} : j \in \mathcal{N}(i)\}\big)$ `// neighborhood mean`
  else `// lstm`
    $m_i \leftarrow \text{LSTM}_{\Theta_{\text{lstm}}}\big(\{h_j^{(0)} : j \in \mathcal{N}(i)\}\big)$ `// LSTM aggregator`
  end
  $h_i^{(1)} \leftarrow \text{ReLU}\big(W^{(0)} m_i + W^{(0)}_{\text{self}} h_i^{(0)}\big)$ `// combine neighbor & self`
end
Dropout:
foreach $i \in V$ **do**
  $\tilde{h}_i^{(1)} \leftarrow \text{Dropout}\big(h_i^{(1)}, p_{\text{do}}\big)$ `// regularization`
end
Layer 2:
foreach $i \in V$ **do**
  if *agg_type* = ***mean*** **then**
    $m_i \leftarrow \text{MEAN}\big(\{\tilde{h}_j^{(1)} : j \in \mathcal{N}(i)\}\big)$
  else
    $m_i \leftarrow \text{LSTM}_{\Theta_{\text{lstm}}}\big(\{\tilde{h}_j^{(1)} : j \in \mathcal{N}(i)\}\big)$
  end
  $z_i \leftarrow W^{(1)} m_i + W^{(1)}_{\text{self}} \tilde{h}_i^{(1)}$ `// final linear transformation`
end
return $\{z_i\}_{i \in V}$

Remark: Although only the `mean` and `lstm` paths are shown, the same template

works with any GraphSAGE-compatible aggregator e.g. `gcn`, `max_pool`, `sum`, or attention-based variants—simply by replacing the definition of m_i.

Notations (Algorithm 5):

- V: set of nodes; E: set of edges.
- $\mathcal{N}(i)$: neighbor set of node i.
- d_{in}: dimension of input features.
- d_{hidden}: dimension of hidden layer.
- C: number of output classes.
- $h_i^{(l)}$: embedding of node i after layer l.
- m_i: aggregated neighborhood message for node i.

3.5 The Objective Function

The objective function is at the heart of the GNN model training, because it defines what the model optimizes. In the vertex coloring problem, the objective is combinatorial, which is to assign a color to every vertex so that no two adjacent vertices share the same color and depending on the formulation, either minimize the total number of colors to achieve the chromatic number or minimize the number of clashes knowing the chromatic number. In this study, the objective is to minimize the number of clashes knowing the true chromatic number. However, directly optimizing the combinatorial objective with gradient-based learning is not straightforward because it is not differentiable. Therefore, a differentiable loss function was used that captures the goal of vertex coloring. This objective function was taken from (Schuetz et al., 2022), and it is the *Potts-Based Loss* function inspired by the Potts model of statistical physics. It is important to note that for evaluating the quality of a solution, the actual graph coloring cost, i.e. *Hard Coloring Cost* is used.

3.5.1 Potts-Based Loss

This study adopts an unsupervised loss function inspired by the anti-ferromagnetic q-state Potts model (Schuetz et al., 2022). The Potts model is a generalization of the Ising model to multiple spins. In a graph coloring context, each node i is associated with a spin (or color) variable $\sigma_i \in 1, \ldots, q$, and an edge between nodes i and j contributes an energy $-J$ if $\sigma_i = \sigma_j$ (same color) or 0 if $\sigma_i \neq \sigma_j$ (different colors). Setting $J = -1$ gives an energy that is lower (better) when adjacent nodes have different colors, and the ground-state energy (minimum) is 0 if and only if the graph is properly q-colorable. The Hamiltonian for Potts model can be expressed as

$$\mathcal{H}\big(\{\sigma\}\big) \;=\; -J \sum_{(i,j)\in E} \delta\big(\sigma_i, \sigma_j\big),$$

where $\delta(\sigma_i, \sigma_j)$ is the Kronecker delta (1 if colors are the same, 0 otherwise). For our purposes, the equivalent cost is minimized (negatively weighted energy), which counts the same-color edges.

To train the neural network, the color assignments are relaxed to soft assignments. The GNN outputs, for each node i, a probability distribution over q colors (via a softmax on the final logits). Let $p_i = (p_{i,1}, ..., p_{i,q})$ denote the probability vector for node i (where $p_{i,c} \geq 0$ and $\sum_c p_{i,c} = 1$). The Potts-based loss, L_{Potts}, is defined as the expected number of conflicts (edge disagreements) under these probabilities.

$$L_{\text{Potts}} \;=\; \frac{1}{2} \sum_{(i,j)\in E} p_i \cdot p_j \;=\; \frac{1}{2} \sum_{(i,j)\in E} \sum_{c=1}^{q} p_{i,c}\, p_{j,c}\,.$$

This loss is essentially the soft version of the Potts model energy. It penalizes pairs of adjacent nodes for placing probability mass on the same color. If two adjacent nodes i and j have identical soft assignments, the term $p_i \cdot p_j$ will be high, increasing the loss. Conversely, if i and j put their probability mass on different colors, the dot product $p_i \cdot p_j$ will be low. The factor $\frac{1}{2}$ avoids double counting edges, as the sum goes over unordered pairs. By minimizing L_{Potts}, the model learns to output distributions that make adjacent nodes as different as possible in the color space.

3.5.2 Hard Coloring Cost

Although the training occurs with Potts-based loss, the goal is to obtain a valid discrete coloring. Therefore, a hard coloring cost C_{hard} is defined that counts the

number of coloring conflicts (adjacent nodes with the same color) in a discrete assignment. Given a coloring vector $c = (\text{color}(1), \text{color}(2), ..., \text{color}(n))$ assigning one of q colors to each of the n nodes, the cost is

$$C_{\text{hard}}(c) \;=\; \sum_{(i,j)\in E} I\Big[\text{color}(i) = \text{color}(j)\Big],$$

where $I[\cdot]$ is an indicator function that equals 1 if the condition is true (if the nodes i and j share the same color).

This cost is simply the number of edges that share the same color. Proper coloring equals to $C_{\text{hard}} = 0$ according to the chromatic number. In the implementation of the proposed method, after each training epoch, a hard assignment is obtained by taking the argmax of the soft probabilities p_i for each node i. This leads to a color choice $\hat{\sigma}i = \arg\max_c pi, c$ for every node. Then C_{hard} is computed for this assignment by iterating over all edges and counting conflicts. Calculating the hard cost after each training epoch serves two purposes. First, it enables one to monitor the training progress; we expect C_{hard} to decrease, ideally reaching 0 if the model finds a proper coloring. Second, it allows an early stop criterion. If C_{hard} reaches 0, training stops to reduce run time.

Since C_{hard} is an evaluative measure, it is not used directly in backpropagation because it is non-differentiable (depends on argmax assignments). In Result Section 4.4, it compares the final C_{hard} achieved by the proposed methods to those of other methods.

3.6 The Training Procedure

The proposed method trains the models in an unsupervised way. The GNN model minimizes the Potts loss (Section 3.5.1) of the initial subgraphs starting from random initial node embeddings and random model weights using gradient descent. The model uses the AdamW optimizer with a moderate weight decay (on the order of 10^{-2}) for regularization. Each training run continues for a maximum of $T_{\max} = 40,000$ epochs for the expansion strategies and $T_{\max} = 100,000$ for baseline with an early stopping strategy. The early stopping strategy works in two ways. First, if the Potts loss does not improve (improvement is below a tolerance $\tau = 10^{-3}$) or loss

increases, then it adds to the patience count (patience $(P) = 500$). If the patience count reaches P, training is stopped on the current subgraph to prevent overfitting, and training continues with the next subgraph(if there is one). Second, if proper coloring is achieved ($C_{\text{hard}} = 0$), training is stopped on the current subgraph, and the training continues with the next subgraph(if there is one). Throughout training, both soft loss L_{Potts} and discrete cost C_{hard} are monitored as described. At the end of the training, the best color found (lowest C_{hard} encountered) is taken as the output solution of the model. Algorithm 6 summarizes the overall training procedure, including forward pass, loss computation, parameter update, early stopping check, and periodic noise injection.

One challenge in this unsupervised training is to avoid poor local minima. Incremental learning by subgraph expansion is expected to mitigate this. However, to further reduce the chance of being stuck at local minima, a noise injection technique is also used.

Algorithm 6: Overall Unsupervised Training Procedure

Input: Graph $G = (V, E)$; # colours q; Potts loss L_{Potts}; curriculum flag f_{cur} with parameters θ_{sub}; noise flag f_{noise} with schedule $\sigma(t)$; early–stopping window P; maximum epochs $T_{\max}$.

Output: Colouring $\hat{c}$ on G with minimum conflict cost C_{hard}.

1. Curriculum setup
if $f_{\text{cur}} =$ *true* **then**
  $\mathcal{S} \leftarrow \textsc{GenerateSubgraphs}(G, \theta_{\text{sub}})$
else
  $\mathcal{S} \leftarrow \{G\}$ `// train directly on full graph`
end
2. Initialise model
$\mathcal{N} \leftarrow \textsc{InitGNN}(\text{GraphConv} \mid \text{GraphSAGE})$
$\mathbf{E} \leftarrow \text{rand}(|V| \times d)$
foreach *subgraph* $S \in \mathcal{S}$ *(small → full)* **do**
  if $S \neq$ *first stage* **then**
    $\textsc{LoadState}\,(\mathcal{N}, \mathbf{E})$
  end
  $p_{\text{cnt}} \leftarrow 0$, $L_{\text{prev}} \leftarrow \infty$
  for $t = 1$ **to** $T_{\max}$ **do**
    if $f_{\text{noise}} =$ *true* **then**
      $\mathbf{X} \leftarrow \mathbf{E} + \mathcal{N}\big(0, \sigma(t)^2\big)$
    else
      $\mathbf{X} \leftarrow \mathbf{E}$
    end
    $\mathbf{P} \leftarrow \text{softmax}\big(\mathcal{N}(\mathbf{X})\big)$
    $L \leftarrow L_{\text{Potts}}\big(\mathbf{P}; E(S)\big)$
    $\textsc{Backprop}(L)$
    $c \leftarrow \arg\max \mathbf{P}$, $C \leftarrow \sum_{(i,j)\in E(S)} I[c_i = c_j]$
    if $C = 0$ **then**
      break `// subgraph solved`
    end
    if $|L - L_{prev}| \leq \tau$ ***or*** $L > L_{prev}$ **then**
      $p_{\text{cnt}}{+}{+}$
    else
      $p_{\text{cnt}} \leftarrow 0$
    end
    if $p_{cnt} \geq P$ **then**
      break `// no progress`
    end
    $L_{\text{prev}} \leftarrow L$
  end
  $\textsc{SaveState}\,(\mathcal{N}, \mathbf{E})$ `// warm-start next stage`
end
3. Final colouring
$\hat{c} \leftarrow \arg\max \mathbf{P}$ on full graph
return $\hat{c}, C_{\text{hard}}(\hat{c})$

Notations (Algorithm 6):

- $G = (V, E)$: undirected input graph with vertex set V and edge set E.
- q: target number of colors (classes) for the coloring task.
- L_{Potts}: unsupervised Potts loss that penalizes same-color neighbors.
- $f_{\text{cur}} \in \{\texttt{true}, \texttt{false}\}$: flag enabling curriculum (subgraph expansion).
- θ_{sub}: hyper-parameters for the subgraph generator (e.g. BFS depth, sampling rate).
- $\mathcal{S} = \langle S_1, S_2, \ldots, S_K \rangle$: ordered list of subgraphs returned by the curriculum; $\mathcal{S} = \{G\}$ if $f_{\text{cur}} = \texttt{false}$.
- $f_{\text{noise}} \in \{\texttt{true}, \texttt{false}\}$: flag enabling Gaussian noise injection.
- $\sigma(t)$: epoch-dependent standard deviation of the injected noise at epoch t.
- $\mathcal{N}$: Graph neural network (GraphConv or GraphSAGE) being trained.
- $\mathbf{E} \in \mathbb{R}^{|V| \times d}$: trainable node-embedding matrix; d = embedding dimension.
- $T_{\max}$: maximum number of training epochs per subgraph stage.
- P: patience window for early stopping (no progress for P epochs $\Rightarrow$ stop).
- p_{cnt}: counter of consecutive epochs without sufficient loss improvement.
- τ: tolerance threshold that defines "no improvement" for the Potts loss.
- L_{prev}: Potts loss value from the previous epoch (used to detect stagnation).
- $\mathbf{X}$: current input embeddings after optional noise injection, $\mathbf{X} = \mathbf{E}$ or $\mathbf{E} + \mathcal{N}(0, \sigma(t)^2)$.
- $\mathbf{P}$: softmax probability matrix produced by $\mathcal{N}$; $\mathbf{P}_{i,c} = p_{i,c}$.
- c: discrete color vector obtained via $c_i = \arg\max_c \mathbf{P}_{i,c}$ for every node i.
- $C = \sum_{(i,j) \in E(S)} I[c_i = c_j]$: hard conflict cost on the current subgraph S.
- LoadState, SaveState: utilities that transfer or persist $(\mathcal{N}, \mathbf{E})$ between successive subgraph stages.
- $\hat{c}$: final colouring returned for the full graph G.
- $C_{\text{hard}}(\hat{c})$: conflict cost of $\hat{c}$ on G (0 for a proper coloring).

3.6.1 Noise Injection

Noise injection is a technique that is used to help the model escape from local minima by injecting random noise into the node features during training. The idea is similar to simulated annealing in physics. In practice, Gaussian noise is added to the node embedding before feeding them into the GNN at each training epoch. If x_i is the current embedding vector for node i, it is replaced with $x_i \cdot (1 + \epsilon)$, where $\epsilon \sim \mathcal{N}\left(0, \left(\frac{1}{4}\alpha\right)^2\right)$ is a Gaussian noise vector. The α can be varied over time according to a schedule. The implementation in this study supports different noise schedules. The experiments contain both a linearly increasing noise schedule and a linearly decreasing noise schedule over the course of training. In both of the noise schedules two parameters are used. These noise parameters (initial $\alpha_{\min}$, final $\alpha_{\max}$) and the choice of schedule are treated as hyperparameters to tune. Linearly decreasing schedule starts with a higher noise level at the beginning to encourage exploration. It gradually reduces the noise towards $\alpha_{\max}$ as training progresses to allow fine-tuning and convergence. This resembles an annealing process in which the system cools down over time. Conversely, a linearly increasing schedule starts with a small noise and increases it. This might be useful if we observe that the model converges too quickly to a local minimum, so that more noise is added later to shake it out of that basin.

3.6.2 Loading Pretrained Weights / Embeddings

In the proposed subgraph expansion technique, the training algorithm of the GNN model starts with a smaller subgraph and gradually expands the subgraph to eventually include the entire graph. At each expansion step, the GNN is initialized for the new larger subgraph using the learned embeddings and weights from the previous step. In this way, the model knowledge of the smaller subgraph is retained and serves as a good starting point for the larger graph. Section 3.3 (Subgraph Expansion and Incremental Training) has already detailed the expansion strategies. This section describes how the learned weights and embeddings are carried over between these stages.

The proposed framework saves two types of information from a trained subgraph model. The first is the weights of the GNN model, and the second is the embedding weights of the node (the learned embedding vector for each node). After training on

a subgraph, the framework saves the model's state dictionary and the embedding matrix to disk. Before training the next (larger) subgraph, which shares many nodes with the previous one, the framework loads these saved weights to initialize the new model. For the embeddings, the learned embedding vectors of all nodes that were present in the previous subgraph are copied into the initial embedding vectors of the new model. This is done by mapping between the old subgraph's node indices and the new subgraph's indices. Any new nodes that were not seen before get randomly initialized embeddings. Similarly, for the GNN's layer weights, the learned model weights from the previous subgraph are initialized for the new model.

The framework allows flexibility in what to carry over between expansions. One can switch on and off between the strategy of reusing node embedding vectors from the previous stage and the reuse of GNN layer parameters. The experiments include three configurations when moving to a larger subgraph. The first is loading both the previous embeddings and GNN weights, the second is loading only the embeddings (and initializing GNN weights randomly for the new run), and the third is loading only the GNN weights (with embeddings reinitialized randomly). In all cases, the idea is to give the new subgraph a warm start by using knowledge from an easier instance.

4. COMPUTATIONAL EXPERIMENTS

The aim of this chapter is to present and analyze the computational experiments conducted on the subgraph expansion approach for graph coloring. This chapter describes the COLOR benchmark dataset and details of the experimental setup that was used in the code, metrics employed to evaluate the model's performance, and analysis of the experimental results.

4.1 Description of the Dataset

In this work, several benchmark graphs, which are commonly used in the literature (Li et al., 2022; Pugacheva et al., 2024; Schuetz et al., 2022), were used, including a subset of the COLOR benchmark dataset (Trick, 2002). These graphs are chosen because they are standard references in the field. The COLOR benchmark dataset consists of around 100 graphs, but in this work, as in the literature, only a few of them were tested, namely 'anna', 'myciel5', 'myciel6', 'queen5_5', 'queen6_6', 'queen7_7', 'queen8_8', 'queen8_12', 'queen9_9', 'queen11_11', 'queen13_13'. These selected graphs show different network properties and vary in size, density, and chromatic number.

Each graph belongs to different groups based on their structural properties and real-world inspirations. Table 4.1 presents the fundamental properties of the graphs, including the number of vertices, edges, density, and the chromatic number. Meanwhile, Table 4.2 provides additional structural properties such as the average degree, maximum degree, clustering coefficient, shortest path length, and diameter (see Newman (2003) for definitions). These properties help to understand the complexity involved in solving the graph coloring problem.

Table 4.1 Basic Properties of the Selected Graphs

Graph	**Vertices (|V|)**	**Edges (|E|)**	**Density (D)**	**Chromatic Number (χ)**
anna.col	138	493	5.00%	11
myciel5.col	47	236	22.00%	6
myciel6.col	95	755	17.00%	7
queen5_5.col	25	160	**53.00%**	5
queen6_6.col	36	290	46.00%	7
queen7_7.col	49	476	40.00%	7
queen8_8.col	64	728	36.00%	9
queen8_12.col	96	1368	30.00%	12
queen9_9.col	81	1056	33.00%	10
queen11_11.col	121	1980	27.00%	11
queen13_13.col	**169**	**3328**	23.00%	**13**

Table 4.2 Structural Properties of the Selected Graphs

Graph	**Avg Degree**	**Max Degree**	**Avg Clustering**	**Avg Shortest Path**	**Diameter**
anna.col	7.14	**71**	**0.65**	**2.45**	**5**
myciel5.col	10.04	23	0.00	1.78	2
myciel6.col	15.89	47	0.00	1.83	2
queen5_5.col	12.80	16	0.50	1.47	2
queen6_6.col	16.11	19	0.46	1.54	2
queen7_7.col	19.43	24	0.43	1.60	2
queen8_8.col	22.75	27	0.41	1.64	2
queen8_12.col	28.50	32	0.39	1.70	2
queen9_9.col	26.07	32	0.39	1.67	2
queen11_11.col	32.73	40	0.37	1.73	2
queen13_13.col	**39.38**	48	0.35	1.77	2

4.1.1 Mycielski Graphs

Mycielski graphs were introduced by Jan Mycielski in 1955 (Mycielski, 1955). They are a family of graphs specifically constructed to have a high chromatic number while remaining sparse. They are triangle-free (containing no 3-cycles) and generally avoid small cliques, yet their chromatic number increases with each iterative construction. This combination of high chromatic number and low edge density makes Mycielski graphs particularly challenging instances for coloring algorithms (Yolcu, Wu & Heule, 2020). Because of their unique characteristics, they remain a classic example of graph coloring problems.

4.1.2 Queen Graphs

Queen graphs are inspired by chess and represent the placement of queens on a chessboard. In a queen graph, each vertex represents a square on the chessboard, and two vertices are connected by an edge if the corresponding squares are in the same row, column, or diagonal. Queen graphs are interesting because they have a high chromatic number, making them challenging instances for coloring algorithms. This structure makes queen graphs highly connected and relatively dense in edges: Each vertex is adjacent to many others. The density and regular connectivity of queen graphs poses a tough challenge for coloring algorithms, as many colors are needed and many constraints (edges) must be satisfied (Bozóki, Gál, Marosi & Weakley, 2019).The queen graph coloring problem is therefore well known in the literature as a challenging case for graph coloring algorithms.

4.1.3 Social Network Graphs

Anna graph is an example of social network graphs extracted from the literature. This graph was introduced by Donald E. Knuth (Knuth, 1993). Anna graph represents Tolstoy's Anna Karenina novel character network, where each vertex represents a character, and an edge represents a relationship between two characters. These types of graphs tend to be medium-sized (on the order of tens to a few hundred vertices) and have a moderate edge density reflecting the pattern of character co-occurrences. The degree distribution in such social graphs is often uneven: main characters (like Anna) form hubs connecting to many others, whereas minor characters connect to only a few. Typically, social networks from single works of fiction provide realistic and irregular structures which makes it easier to color compared to regular graphs like queen or Mycielski graphs.

4.2 Hyperparameter Tuning with Ray

Given the number of components in the proposed method, there are many hyperparameters that can affect performance. An extensive hyperparameter tuning

procedure was conducted using Ray Tune (Liaw, Liang, Nishihara, Moritz, Gonzalez & Stoica, 2018), which is a scalable framework for distributed hyperparameter optimization. The goal is to find hyperparameter configurations that perform well for each type of expansion across various graph instances. Using Ray's distributed tuning, a broad search space is explored efficiently by running many training trials in parallel. The key hyperparameters and design options are shown in Table 4.19.

4.2.1 Ray Tune Setup

Ray Tune was configured with the search space shown in Table 4.19 and used an Optuna optimizer as the search algorithm. In particular, Optuna's Tree-structured Parzen Estimator (TPE) sampler with default parameters was used for suggesting hyperparameter values. Ray Tune's Optuna Search integration was used with objective metric set to the best hard cost achieved by the model. Each trial run trains the GNN model until it stops, and then reports the lowest best hard cost obtained. To account for randomness in training, a repeater strategy was used for the trials. Each suggested hyperparameter set was run three times (with different seeds) and the results were averaged. This ensures that a configuration is good on average, not just coincidence. This was implemented with the Repeater wrapper in Ray Tune. The optimization mode was set to 'min' for the averaged best hard-cost metric.

Ray was run in a distributed setting. A Ray cluster was initialized to utilize multiple CPU cores. Each trial was dispatched as a Ray remote task, allowing many trials to run in parallel. This significantly sped up the search. Maximum 12 concurrent trials were allowed in the setting due to memory limitations. After the tuning phase, the results of the tuning are examined to identify the best hyperparameter settings.

4.2.2 Hyperparameter Insights

An interesting by-product of the extensive tuning is understanding which hyperparameter values were most often selected for the best-performing runs under each strategy. The best-performing run is obtained by first grouping the trials by hyperparameter configuration and computing each hyperparameter configuration's average best hard cost and average runtime of each configuration. Secondly, aggregated results grouped by problem instance and type, then among the configurations that

achieved the lowest best hard cost were selected. Since there might be more than one grouped configuration that can achieve the lowest best hard cost, a secondary criterion was applied, and the one with the shortest runtime (fastest configuration among the lowest best hard cost configuration) was selected. The first set reports categorical hyperparameters (see Tables 4.3—4.6). It shows how often each option appears among the best-performing configurations (with the final selected value highlighted in bold, and below counts shows frequency of other categories; missing values are displayed as nan to indicate that the parameter was not applicable or not present) for each graph and strategy. The second set reports numerical hyperparameters (see Tables 4.7—4.10). The chosen value (in bold) was listed along with the minimum, average, maximum, and standard deviation of that parameter across the best-performing configurations. These tables make it easy to see at a glance which hyperparameter choices were most often associated with the best results.

Across all strategies in the selected configurations, the GraphSAGE GNN model was overwhelmingly the architecture of choice. The only notable exceptions were Myciel graphs where a simpler GraphConv network marginally outperformed GraphSAGE. When GraphSAGE was used, the mean aggregator dominated, the more complex lstm aggregator appeared only for Anna and a few small Queen graphs. Myciel graphs have no layer aggregation (marked nan) because Graph Convolutional does not use it. The use of previous state knowledge turned out to be highly dependent on the expansion strategy employed. Both degree-first BFS-based expansion and random walk expansion mostly use the "loading pre-trained node embeddings" and "loading pre-trained model weights". However, the layer-by-layer BFS-based expansion mostly reused the node embeddings between layers but did not reuse model weights in most of the graphs. The use of noise in training was another key differentiator among the strategies. Random-walk never used noise, so no schedule was was applied. Similarly, degree-first BFS almost never used noise; only Myciel5 used noise in its best run. However, the baseline (no expansion) and layer-by-layer BFS strategies often used noise injection. The baseline (no expansion) and layer-by-layer BFS runs mostly favored a decreasing schedule. Beyond direct observations of selected values, certain obvious patterns emerge when considering the numerical hyperparameters in relation to their defined search spaces. For instance, while the dim_embedding and hidden_dim parameters offer broad integer ranges (5–50 and 5–80 respectively), the selected values rarely converge to the extremes of these ranges, suggesting that excessively small or large dimensions are generally not optimal. Similarly, for dropout, despite a range of 0.1 to 0.3, the selected values frequently cluster toward the higher end, often exceeding 0.2, indicating a consistent preference for a more pronounced regularization effect. In contrast, the learn-

ing_rate (0.01–0.1) consistently sees selected values in the lower half of its range, rarely approaching the upper limit, which points to a preference for more conservative optimization steps. When noise scaling parameters are applicable, $\alpha_{\min}$ (0.1, 0.2, 0.3, 0.4) tends to be selected from its lower end (0.1 or 0.2), while $\alpha_{\max}$ (0.5, 0.6, 0.7, 0.8, 0.9) often falls into its higher end (0.7, 0.8, 0.9), suggesting a common strategy for noise application where the minimum scaling is small but the maximum can be quite high. Lastly, when step_percentage is relevant, its selected values are distributed across its entire categorical range (0.1, 0.2, 0.3, 0.4, 0.5), indicating no single universally optimal step size. To gain insight into the differences between problems, the selected hyperparameters were visualized in the form of radar charts for each expansion strategy in Figures 4.1—4.4. Each small radar subplot corresponds to one problem instance (the instance name is indicated above each plot), allowing us to compare the hyperparameter profiles across different graphs.

To see higher-level trends, we also aggregated selected tuning outcomes by problem family. This grouped summary (see Tables 4.11—4.14 for categorical and see Tables 4.15—4.18 for the corresponding numeric summary by family) consolidates how often each hyperparameter option was selected for the Queen graphs, Myciel graphs, and the single instance Anna graph.

Table 4.3 Distribution of hyper-parameter values explored during tuning for The layer-by-layer BFS-based expansion. For every graph instance (problem file) (**selected**, other categories).

problem file	layer agg	load embedding	load weights	model	noise schedule	use noise
anna.col	**lstm**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	lstm: 17, mean: 1	True: 18	True: 14, False: 4	GraphSAGE: 18	nan: 17, decreasing: 1	False: 17, True: 1
myciel5.col	**nan**	**True**	**False**	**GraphConv**	**nan**	**False**
	nan: 29, lstm: 1, mean: 1	True: 18, False: 13	True: 19, False: 12	GraphConv: 29, GraphSAGE: 2	nan: 20, decreasing: 9, increasing: 2	False: 20, True: 11
myciel6.col	**nan**	**True**	**False**	**GraphConv**	**nan**	**False**
	nan: 24, lstm: 6, mean: 1	True: 21, False: 10	True: 16, False: 15	GraphConv: 24, GraphSAGE: 7	nan: 16, increasing: 12, decreasing: 3	False: 16, True: 15
queen5_5.col	**lstm**	**True**	**False**	**GraphSAGE**	**decreasing**	**True**
	lstm: 12	False: 11, True: 1	True: 11, False: 1	GraphSAGE: 12	decreasing: 12	True: 12
queen6_6.col	**mean**	**False**	**True**	**GraphSAGE**	**nan**	**False**
	mean: 2, lstm: 1	False: 2, True: 1	True: 2, False: 1	GraphSAGE: 3	decreasing: 2, nan: 1	True: 2, False: 1
queen7_7.col	**lstm**	**True**	**False**	**GraphSAGE**	**decreasing**	**True**
	lstm: 1	True: 1	False: 1	GraphSAGE: 1	decreasing: 1	True: 1
queen8_8.col	**mean**	**True**	**True**	**GraphSAGE**	**decreasing**	**True**
	mean: 1	True: 1	True: 1	GraphSAGE: 1	decreasing: 1	True: 1
queen8_12.col	**mean**	**True**	**False**	**GraphSAGE**	**increasing**	**True**
	mean: 1	True: 1	False: 1	GraphSAGE: 1	increasing: 1	True: 1
queen9_9.col	**mean**	**False**	**True**	**GraphSAGE**	**decreasing**	**True**
	mean: 1	False: 1	True: 1	GraphSAGE: 1	decreasing: 1	True: 1
queen11_11.col	**lstm**	**True**	**False**	**GraphSAGE**	**decreasing**	**True**
	lstm: 1	True: 1	False: 1	GraphSAGE: 1	decreasing: 1	True: 1
queen13_13.col	**mean**	**True**	**True**	**GraphSAGE**	**increasing**	**True**
	mean: 1	True: 1	True: 1	GraphSAGE: 1	increasing: 1	True: 1

Table 4.4 Distribution of hyper-parameter values explored during tuning for degree-first BFS-based expansion. For every graph instance (problem file) (**selected**, other categories).

problem file	layer agg	load embedding	load weights	model	noise schedule	use noise
anna.col	**lstm**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	lstm: 19	True: 17, False: 2	True: 19	GraphSAGE: 19	nan: 19	False: 19
myciel5.col	**nan**	**True**	**True**	**GraphConv**	**increasing**	**True**
	nan: 18, lstm: 12	True: 17, False: 13	True: 27, False: 3	GraphConv: 18, GraphSAGE: 12	nan: 16, decreasing: 9, increasing: 5	False: 16, True: 14
myciel6.col	**mean**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	lstm: 23, nan: 5, mean: 2	False: 15, True: 15	True: 28, False: 2	GraphSAGE: 25, GraphConv: 5	nan: 29, increasing: 1	False: 29, True: 1
queen5_5.col	**mean**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	lstm: 11, mean: 7	True: 16, False: 2	True: 18	GraphSAGE: 18	nan: 18	False: 18
queen6_6.col	**mean**	**False**	**True**	**GraphSAGE**	**nan**	**False**
	mean: 1	False: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1
queen7_7.col	**lstm**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	lstm: 1	True: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1
queen8_8.col	**mean**	**False**	**True**	**GraphSAGE**	**nan**	**False**
	mean: 2	False: 2	True: 2	GraphSAGE: 2	nan: 2	False: 2
queen8_12.col	**mean**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	mean: 1	True: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1
queen9_9.col	**mean**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	mean: 1	True: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1
queen11_11.col	**mean**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	mean: 1	True: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1
queen13_13.col	**mean**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	mean: 1	True: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1

Table 4.5 Distribution of hyper-parameter values explored during tuning for random-walk expansion. For every graph instance (problem file) (**selected**, other categories).

problem file	layer agg	load embedding	load weights	model	noise schedule	use noise
anna.col	**lstm**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	lstm: 18, mean: 2	True: 20	True: 20	GraphSAGE: 20	nan: 20	False: 20
myciel5.col	**nan**	**True**	**True**	**GraphConv**	**nan**	**False**
	nan: 20, lstm: 11, mean: 3	True: 20, False: 14	True: 24, False: 10	GraphConv: 20, GraphSAGE: 14	nan: 26, decreasing: 7, increasing: 1	False: 26, True: 8
myciel6.col	**nan**	**True**	**False**	**GraphConv**	**nan**	**False**
	lstm: 19, nan: 12, mean: 2	False: 17, True: 16	True: 31, False: 2	GraphSAGE: 21, GraphConv: 12	nan: 21, decreasing: 11, increasing: 1	False: 21, True: 12
queen5_5.col	**mean**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	mean: 14, lstm: 3	True: 17	True: 17	GraphSAGE: 17	nan: 17	False: 17
queen6_6.col	**lstm**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	lstm: 1	True: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1
queen7_7.col	**mean**	**False**	**True**	**GraphSAGE**	**nan**	**False**
	lstm: 1, mean: 1	False: 1, True: 1	True: 2	GraphSAGE: 2	nan: 2	False: 2
queen8_8.col	**mean**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	mean: 2	True: 2	True: 2	GraphSAGE: 2	nan: 2	False: 2
queen8_12.col	**mean**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	mean: 1	True: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1
queen9_9.col	**lstm**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	lstm: 1	True: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1
queen11_11.col	**mean**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	mean: 1	True: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1
queen13_13.col	**mean**	**True**	**True**	**GraphSAGE**	**nan**	**False**
	mean: 1	True: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1

Table 4.6 Distribution of hyper-parameter values explored during tuning for The baseline (no expansion). For every graph instance (problem file) (**selected**, other categories).

problem file	layer agg	load embedding	load weights	model	noise schedule	use noise
anna.col	**lstm**	**False**	**False**	**GraphSAGE**	**decreasing**	**True**
	lstm: 10	False: 10	False: 10	GraphSAGE: 10	decreasing: 7, nan: 2, increasing: 1	True: 8, False: 2
myciel5.col	**nan**	**False**	**False**	**GraphConv**	**decreasing**	**True**
	nan: 24, lstm: 3, mean: 2	False: 29	False: 29	GraphConv: 24, GraphSAGE: 5	nan: 16, decreasing: 9, increasing: 4	False: 16, True: 13
myciel6.col	**nan**	**False**	**False**	**GraphConv**	**nan**	**False**
	nan: 17, lstm: 9, mean: 1	False: 27	False: 27	GraphConv: 17, GraphSAGE: 10	nan: 15, decreasing: 9, increasing: 3	False: 15, True: 12
queen5_5.col	**mean**	**False**	**False**	**GraphSAGE**	**nan**	**False**
	mean: 5	False: 5	False: 5	GraphSAGE: 5	nan: 3, increasing: 2	False: 3, True: 2
queen6_6.col	**lstm**	**False**	**False**	**GraphSAGE**	**nan**	**False**
	lstm: 7	False: 7	False: 7	GraphSAGE: 7	increasing: 3, nan: 3, decreasing: 1	True: 4, False: 3
queen7_7.col	**mean**	**False**	**False**	**GraphSAGE**	**decreasing**	**True**
	lstm: 1, mean: 1	False: 2	False: 2	GraphSAGE: 2	decreasing: 1, nan: 1	False: 1, True: 1
queen8_8.col	**mean**	**False**	**False**	**GraphSAGE**	**increasing**	**True**
	mean: 1	False: 1	False: 1	GraphSAGE: 1	increasing: 1	True: 1
queen8_12.col	**mean**	**False**	**False**	**GraphSAGE**	**decreasing**	**True**
	mean: 2	False: 2	False: 2	GraphSAGE: 2	decreasing: 1, increasing: 1	True: 2
queen9_9.col	**mean**	**False**	**False**	**GraphSAGE**	**decreasing**	**True**
	mean: 2	False: 2	False: 2	GraphSAGE: 2	decreasing: 2	True: 2
queen11_11.col	**mean**	**False**	**False**	**GraphSAGE**	**increasing**	**True**
	mean: 1	False: 1	False: 1	GraphSAGE: 1	increasing: 1	True: 1
queen13_13.col	**mean**	**False**	**False**	**GraphSAGE**	**decreasing**	**True**
	mean: 1	False: 1	False: 1	GraphSAGE: 1	decreasing: 1	True: 1

Table 4.7 Distribution of hyper-parameter values explored during tuning for The layer-by-layer BFS-based expansion. For every graph instance (problem file) (**selected**, minimum / average / maximum / standard deviation).

problem file	$\alpha_{\max}$	$\alpha_{\min}$	dim embedding	dropout	hidden dim	learning rate	step percentage (ρ)
anna.col	**nan**	**nan**	**35**	**0.2001**	**19**	**0.02897**	**nan**
	0.7 / 0.7 / 0.7 / nan	0.2 / 0.2 / 0.2 / nan	15 / 35.33 / 46 / 7.685	0.1338 / 0.1792 / 0.233 / 0.0283	17 / 29.44 / 41 / 7.073	0.0287 / 0.0434 / 0.056 / 0.0086	nan / nan / nan / nan
myciel5.col	**nan**	**nan**	**45**	**0.1459**	**18**	**0.02923**	**nan**
	0.5 / 0.6636 / 0.9 / 0.1629	0.1 / 0.2182 / 0.4 / 0.0874	5 / 34.26 / 49 / 12.26	0.1043 / 0.2045 / 0.2971 / 0.0521	6 / 23.23 / 72 / 15.96	0.0111 / 0.0316 / 0.0576 / 0.0125	nan / nan / nan / nan
myciel6.col	**nan**	**nan**	**36**	**0.1437**	**36**	**0.02011**	**nan**
	0.5 / 0.7333 / 0.9 / 0.1589	0.1 / 0.2467 / 0.4 / 0.1407	5 / 27.9 / 50 / 15.54	0.1023 / 0.2034 / 0.2969 / 0.0599	5 / 22 / 73 / 16.34	0.01 / 0.0384 / 0.0795 / 0.021	nan / nan / nan / nan
queen5_5.col	**0.8**	**0.2**	**11**	**0.2442**	**16**	**0.0866**	**nan**
	0.8 / 0.8 / 0.8 / 0	0.2 / 0.2167 / 0.4 / 0.0577	8 / 12.17 / 26 / 4.951	0.1647 / 0.2151 / 0.2442 / 0.0249	16 / 19.25 / 24 / 2.379	0.0751 / 0.0861 / 0.093 / 0.0057	nan / nan / nan / nan
queen6_6.col	**nan**	**nan**	**44**	**0.1827**	**35**	**0.02839**	**nan**
	0.7 / 0.75 / 0.8 / 0.0707	0.4 / 0.4 / 0.4 / 0	35 / 40.67 / 44 / 4.933	0.1667 / 0.1767 / 0.1827 / 0.0087	32 / 35.67 / 40 / 4.042	0.0201 / 0.0259 / 0.0293 / 0.0051	nan / nan / nan / nan
queen7_7.col	**0.8**	**0.2**	**9**	**0.2377**	**24**	**0.07377**	**nan**
	0.8 / 0.8 / 0.8 / nan	0.2 / 0.2 / 0.2 / nan	9 / 9 / 9 / nan	0.2377 / 0.2377 / 0.2377 / nan	24 / 24 / 24 / nan	0.0738 / 0.0738 / 0.0738 / nan	nan / nan / nan / nan
queen8_8.col	**0.8**	**0.2**	**16**	**0.1809**	**51**	**0.0247**	**nan**
	0.8 / 0.8 / 0.8 / nan	0.2 / 0.2 / 0.2 / nan	16 / 16 / 16 / nan	0.1809 / 0.1809 / 0.1809 / nan	51 / 51 / 51 / nan	0.0247 / 0.0247 / 0.0247 / nan	nan / nan / nan / nan
queen8_12.col	**0.6**	**0.1**	**16**	**0.1499**	**34**	**0.04698**	**nan**
	0.6 / 0.6 / 0.6 / nan	0.1 / 0.1 / 0.1 / nan	16 / 16 / 16 / nan	0.1499 / 0.1499 / 0.1499 / nan	34 / 34 / 34 / nan	0.047 / 0.047 / 0.047 / nan	nan / nan / nan / nan
queen9_9.col	**0.9**	**0.1**	**11**	**0.2283**	**30**	**0.07491**	**nan**
	0.9 / 0.9 / 0.9 / nan	0.1 / 0.1 / 0.1 / nan	11 / 11 / 11 / nan	0.2283 / 0.2283 / 0.2283 / nan	30 / 30 / 30 / nan	0.0749 / 0.0749 / 0.0749 / nan	nan / nan / nan / nan
queen11_11.col	**0.6**	**0.4**	**10**	**0.2665**	**34**	**0.05785**	**nan**
	0.6 / 0.6 / 0.6 / nan	0.4 / 0.4 / 0.4 / nan	10 / 10 / 10 / nan	0.2665 / 0.2665 / 0.2665 / nan	34 / 34 / 34 / nan	0.0578 / 0.0578 / 0.0578 / nan	nan / nan / nan / nan
queen13_13.col	**0.9**	**0.1**	**20**	**0.1816**	**30**	**0.05099**	**nan**
	0.9 / 0.9 / 0.9 / nan	0.1 / 0.1 / 0.1 / nan	20 / 20 / 20 / nan	0.1816 / 0.1816 / 0.1816 / nan	30 / 30 / 30 / nan	0.051 / 0.051 / 0.051 / nan	nan / nan / nan / nan

Table 4.8 Distribution of hyper-parameter values explored during tuning for degree-first BFS-based expansion. For every graph instance (problem file) (**selected**, minimum / average / maximum / standard deviation).

problem file	$\alpha_{\max}$	$\alpha_{\min}$	dim embedding	dropout	hidden dim	learning rate	step percentage (ρ)
anna.col	**nan**	**nan**	**15**	**0.2352**	**31**	**0.04169**	**0.2**
	nan / nan / nan / nan	nan / nan / nan / nan	8 / 18.68 / 49 / 8.801	0.1179 / 0.1949 / 0.2383 / 0.0343	5 / 32.47 / 48 / 10.39	0.0106 / 0.0365 / 0.0667 / 0.0113	0.2 / 0.3053 / 0.4 / 0.0621
myciel5.col	**0.8**	**0.1**	**16**	**0.1002**	**40**	**0.01457**	**0.5**
	0.5 / 0.6357 / 0.9 / 0.1447	0.1 / 0.2 / 0.4 / 0.0679	5 / 23.8 / 50 / 17.33	0.1002 / 0.2183 / 0.2993 / 0.0575	5 / 34.37 / 80 / 22.25	0.0101 / 0.0474 / 0.0986 / 0.0257	0.3 / 0.3933 / 0.5 / 0.074
myciel6.col	**nan**	**nan**	**21**	**0.1319**	**15**	**0.05034**	**0.4**
	0.6 / 0.6 / 0.6 / nan	0.1 / 0.1 / 0.1 / nan	5 / 13.3 / 28 / 6.103	0.1081 / 0.2022 / 0.299 / 0.0562	5 / 21.3 / 80 / 16.75	0.011 / 0.0394 / 0.0766 / 0.0225	0.1 / 0.3067 / 0.5 / 0.098
queen5_5.col	**nan**	**nan**	**12**	**0.2331**	**5**	**0.07748**	**0.3**
	nan / nan / nan / nan	nan / nan / nan / nan	8 / 14.72 / 30 / 6.134	0.174 / 0.2269 / 0.2598 / 0.0235	5 / 20.83 / 34 / 9.733	0.01 / 0.0597 / 0.0999 / 0.0253	0.1 / 0.2833 / 0.5 / 0.0985
queen6_6.col	**nan**	**nan**	**31**	**0.2092**	**14**	**0.02189**	**0.2**
	nan / nan / nan / nan	nan / nan / nan / nan	31 / 31 / 31 / nan	0.2092 / 0.2092 / 0.2092 / nan	14 / 14 / 14 / nan	0.0219 / 0.0219 / 0.0219 / nan	0.2 / 0.2 / 0.2 / nan
queen7_7.col	**nan**	**nan**	**16**	**0.245**	**33**	**0.04375**	**0.3**
	nan / nan / nan / nan	nan / nan / nan / nan	16 / 16 / 16 / nan	0.245 / 0.245 / 0.245 / nan	33 / 33 / 33 / nan	0.0437 / 0.0437 / 0.0437 / nan	0.3 / 0.3 / 0.3 / nan
queen8_8.col	**nan**	**nan**	**29**	**0.2208**	**17**	**0.05186**	**0.2**
	nan / nan / nan / nan	nan / nan / nan / nan	29 / 30.5 / 32 / 2.121	0.2016 / 0.2112 / 0.2208 / 0.0136	10 / 13.5 / 17 / 4.95	0.0264 / 0.0391 / 0.0519 / 0.018	0.1 / 0.15 / 0.2 / 0.0707
queen8_12.col	**nan**	**nan**	**27**	**0.2557**	**24**	**0.04889**	**0.2**
	nan / nan / nan / nan	nan / nan / nan / nan	27 / 27 / 27 / nan	0.2557 / 0.2557 / 0.2557 / nan	24 / 24 / 24 / nan	0.0489 / 0.0489 / 0.0489 / nan	0.2 / 0.2 / 0.2 / nan
queen9_9.col	**nan**	**nan**	**23**	**0.2623**	**25**	**0.04812**	**0.2**
	nan / nan / nan / nan	nan / nan / nan / nan	23 / 23 / 23 / nan	0.2623 / 0.2623 / 0.2623 / nan	25 / 25 / 25 / nan	0.0481 / 0.0481 / 0.0481 / nan	0.2 / 0.2 / 0.2 / nan
queen11_11.col	**nan**	**nan**	**14**	**0.1484**	**33**	**0.05469**	**0.3**
	nan / nan / nan / nan	nan / nan / nan / nan	14 / 14 / 14 / nan	0.1484 / 0.1484 / 0.1484 / nan	33 / 33 / 33 / nan	0.0547 / 0.0547 / 0.0547 / nan	0.3 / 0.3 / 0.3 / nan
queen13_13.col	**nan**	**nan**	**11**	**0.2167**	**41**	**0.05071**	**0.3**
	nan / nan / nan / nan	nan / nan / nan / nan	11 / 11 / 11 / nan	0.2167 / 0.2167 / 0.2167 / nan	41 / 41 / 41 / nan	0.0507 / 0.0507 / 0.0507 / nan	0.3 / 0.3 / 0.3 / nan

Table 4.9 Distribution of hyper-parameter values explored during tuning for random-walk expansion. For every graph instance (problem file) (**selected**, minimum / average / maximum / standard deviation).

problem file	$\alpha_{\max}$	$\alpha_{\min}$	dim embedding	dropout	hidden dim	learning rate	step percentage (ρ)
anna.col	**nan**	**nan**	**16**	**0.245**	**33**	**0.04375**	**0.3**
	nan / nan / nan / nan	nan / nan / nan / nan	8 / 17.9 / 29 / 5.467	0.174 / 0.235 / 0.2672 / 0.0259	14 / 33.3 / 46 / 8.176	0.0107 / 0.0417 / 0.0667 / 0.0115	0.1 / 0.3 / 0.4 / 0.0649
myciel5.col	**nan**	**nan**	**13**	**0.1701**	**73**	**0.05991**	**0.3**
	0.5 / 0.5875 / 0.9 / 0.1356	0.1 / 0.2375 / 0.4 / 0.0916	5 / 17.88 / 50 / 12.66	0.1019 / 0.1868 / 0.2996 / 0.0613	5 / 45.35 / 80 / 26.38	0.0137 / 0.05 / 0.0821 / 0.0223	0.1 / 0.2971 / 0.4 / 0.1
myciel6.col	**nan**	**nan**	**9**	**0.2641**	**50**	**0.02793**	**0.4**
	0.5 / 0.5917 / 0.9 / 0.1084	0.1 / 0.25 / 0.3 / 0.0798	5 / 16.55 / 50 / 13.4	0.1016 / 0.1997 / 0.2991 / 0.0473	5 / 32.36 / 80 / 22.87	0.0145 / 0.0467 / 0.087 / 0.022	0.1 / 0.3364 / 0.5 / 0.0822
queen5_5.col	**nan**	**nan**	**10**	**0.2397**	**16**	**0.07764**	**0.3**
	nan / nan / nan / nan	nan / nan / nan / nan	5 / 10.18 / 13 / 2.157	0.174 / 0.2282 / 0.2516 / 0.024	5 / 15.94 / 24 / 6.427	0.0119 / 0.0682 / 0.0837 / 0.0182	0.3 / 0.3294 / 0.4 / 0.047
queen6_6.col	**nan**	**nan**	**9**	**0.2325**	**20**	**0.04913**	**0.3**
	nan / nan / nan / nan	nan / nan / nan / nan	9 / 9 / 9 / nan	0.2325 / 0.2325 / 0.2325 / nan	20 / 20 / 20 / nan	0.0491 / 0.0491 / 0.0491 / nan	0.3 / 0.3 / 0.3 / nan
queen7_7.col	**nan**	**nan**	**35**	**0.1574**	**15**	**0.05996**	**0.1**
	nan / nan / nan / nan	nan / nan / nan / nan	29 / 32 / 35 / 4.243	0.1284 / 0.1429 / 0.1574 / 0.0206	15 / 18 / 21 / 4.243	0.0505 / 0.0552 / 0.06 / 0.0067	0.1 / 0.15 / 0.2 / 0.0707
queen8_8.col	**nan**	**nan**	**21**	**0.2154**	**26**	**0.05233**	**0.2**
	nan / nan / nan / nan	nan / nan / nan / nan	21 / 21 / 21 / 0	0.1917 / 0.2035 / 0.2154 / 0.0168	25 / 25.5 / 26 / 0.7071	0.0486 / 0.0505 / 0.0523 / 0.0026	0.2 / 0.2 / 0.2 / 0
queen8_12.col	**nan**	**nan**	**26**	**0.1851**	**25**	**0.05008**	**0.1**
	nan / nan / nan / nan	nan / nan / nan / nan	26 / 26 / 26 / nan	0.1851 / 0.1851 / 0.1851 / nan	25 / 25 / 25 / nan	0.0501 / 0.0501 / 0.0501 / nan	0.1 / 0.1 / 0.1 / nan
queen9_9.col	**nan**	**nan**	**14**	**0.2391**	**32**	**0.04253**	**0.2**
	nan / nan / nan / nan	nan / nan / nan / nan	14 / 14 / 14 / nan	0.2391 / 0.2391 / 0.2391 / nan	32 / 32 / 32 / nan	0.0425 / 0.0425 / 0.0425 / nan	0.2 / 0.2 / 0.2 / nan
queen11_11.col	**nan**	**nan**	**12**	**0.2053**	**41**	**0.05786**	**0.4**
	nan / nan / nan / nan	nan / nan / nan / nan	12 / 12 / 12 / nan	0.2053 / 0.2053 / 0.2053 / nan	41 / 41 / 41 / nan	0.0579 / 0.0579 / 0.0579 / nan	0.4 / 0.4 / 0.4 / nan
queen13_13.col	**nan**	**nan**	**26**	**0.2175**	**24**	**0.03224**	**0.1**
	nan / nan / nan / nan	nan / nan / nan / nan	26 / 26 / 26 / nan	0.2175 / 0.2175 / 0.2175 / nan	24 / 24 / 24 / nan	0.0322 / 0.0322 / 0.0322 / nan	0.1 / 0.1 / 0.1 / nan

Table 4.10 Distribution of hyper-parameter values explored during tuning for The baseline (no expansion). For every graph instance (problem file) (**selected**, minimum / average / maximum / standard deviation).

problem file	$\alpha_{\max}$	$\alpha_{\min}$	dim embedding	dropout	hidden dim	learning rate	step percentage (ρ)
anna.col	**0.7**	**0.2**	**16**	**0.2938**	**46**	**0.05228**	**nan**
	0.7 / 0.75 / 0.9 / 0.0926	0.1 / 0.1875 / 0.2 / 0.0354	14 / 27.7 / 40 / 9.044	0.1776 / 0.2661 / 0.2938 / 0.0332	38 / 51.6 / 64 / 8.003	0.0111 / 0.0359 / 0.0551 / 0.0134	nan / nan / nan / nan
myciel5.col	**0.5**	**0.1**	**50**	**0.2005**	**22**	**0.02951**	**nan**
	0.5 / 0.6308 / 0.9 / 0.1601	0.1 / 0.2846 / 0.4 / 0.1405	5 / 19.21 / 50 / 14.01	0.1005 / 0.1965 / 0.2962 / 0.0572	5 / 37.93 / 80 / 22.64	0.0115 / 0.0351 / 0.0935 / 0.0212	nan / nan / nan / nan
myciel6.col	**nan**	**nan**	**39**	**0.1247**	**60**	**0.0288**	**nan**
	0.5 / 0.675 / 0.9 / 0.1422	0.2 / 0.3333 / 0.4 / 0.0778	5 / 14.11 / 46 / 10.04	0.1031 / 0.2008 / 0.2977 / 0.0616	5 / 45.19 / 75 / 22.88	0.0102 / 0.0404 / 0.0813 / 0.0215	nan / nan / nan / nan
queen5_5.col	**nan**	**nan**	**21**	**0.2513**	**9**	**0.03294**	**nan**
	0.5 / 0.5 / 0.5 / 0	0.2 / 0.2 / 0.2 / 0	5 / 11.2 / 21 / 6.38	0.2266 / 0.2483 / 0.2736 / 0.018	6 / 8.6 / 14 / 3.209	0.018 / 0.0329 / 0.0407 / 0.0089	nan / nan / nan / nan
queen6_6.col	**nan**	**nan**	**22**	**0.2792**	**24**	**0.03599**	**nan**
	0.5 / 0.65 / 0.9 / 0.1915	0.1 / 0.25 / 0.3 / 0.1	17 / 26.14 / 32 / 6.095	0.2631 / 0.2781 / 0.2846 / 0.0072	24 / 30.57 / 43 / 7.368	0.036 / 0.0496 / 0.0649 / 0.0087	nan / nan / nan / nan
queen7_7.col	**0.9**	**0.2**	**16**	**0.2973**	**38**	**0.04824**	**nan**
	0.9 / 0.9 / 0.9 / nan	0.2 / 0.2 / 0.2 / nan	6 / 11 / 16 / 7.071	0.1679 / 0.2326 / 0.2973 / 0.0915	30 / 34 / 38 / 5.657	0.0482 / 0.0547 / 0.0612 / 0.0092	nan / nan / nan / nan
queen8_8.col	**0.5**	**0.2**	**10**	**0.2848**	**11**	**0.07412**	**nan**
	0.5 / 0.5 / 0.5 / nan	0.2 / 0.2 / 0.2 / nan	10 / 10 / 10 / nan	0.2848 / 0.2848 / 0.2848 / nan	11 / 11 / 11 / nan	0.0741 / 0.0741 / 0.0741 / nan	nan / nan / nan / nan
queen8_12.col	**0.9**	**0.3**	**15**	**0.2717**	**57**	**0.04161**	**nan**
	0.9 / 0.9 / 0.9 / 0	0.3 / 0.3 / 0.3 / 0	12 / 13.5 / 15 / 2.121	0.231 / 0.2514 / 0.2717 / 0.0288	57 / 58 / 59 / 1.414	0.0373 / 0.0395 / 0.0416 / 0.003	nan / nan / nan / nan
queen9_9.col	**0.9**	**0.3**	**21**	**0.2807**	**55**	**0.04088**	**nan**
	0.9 / 0.9 / 0.9 / 0	0.3 / 0.3 / 0.3 / 0	18 / 19.5 / 21 / 2.121	0.2742 / 0.2775 / 0.2807 / 0.0046	39 / 47 / 55 / 11.31	0.0409 / 0.0422 / 0.0435 / 0.0019	nan / nan / nan / nan
queen11_11.col	**0.5**	**0.2**	**25**	**0.2323**	**17**	**0.06576**	**nan**
	0.5 / 0.5 / 0.5 / nan	0.2 / 0.2 / 0.2 / nan	25 / 25 / 25 / nan	0.2323 / 0.2323 / 0.2323 / nan	17 / 17 / 17 / nan	0.0658 / 0.0658 / 0.0658 / nan	nan / nan / nan / nan
queen13_13.col	**0.6**	**0.3**	**19**	**0.2977**	**33**	**0.06218**	**nan**
	0.6 / 0.6 / 0.6 / nan	0.3 / 0.3 / 0.3 / nan	19 / 19 / 19 / nan	0.2977 / 0.2977 / 0.2977 / nan	33 / 33 / 33 / nan	0.0622 / 0.0622 / 0.0622 / nan	nan / nan / nan / nan

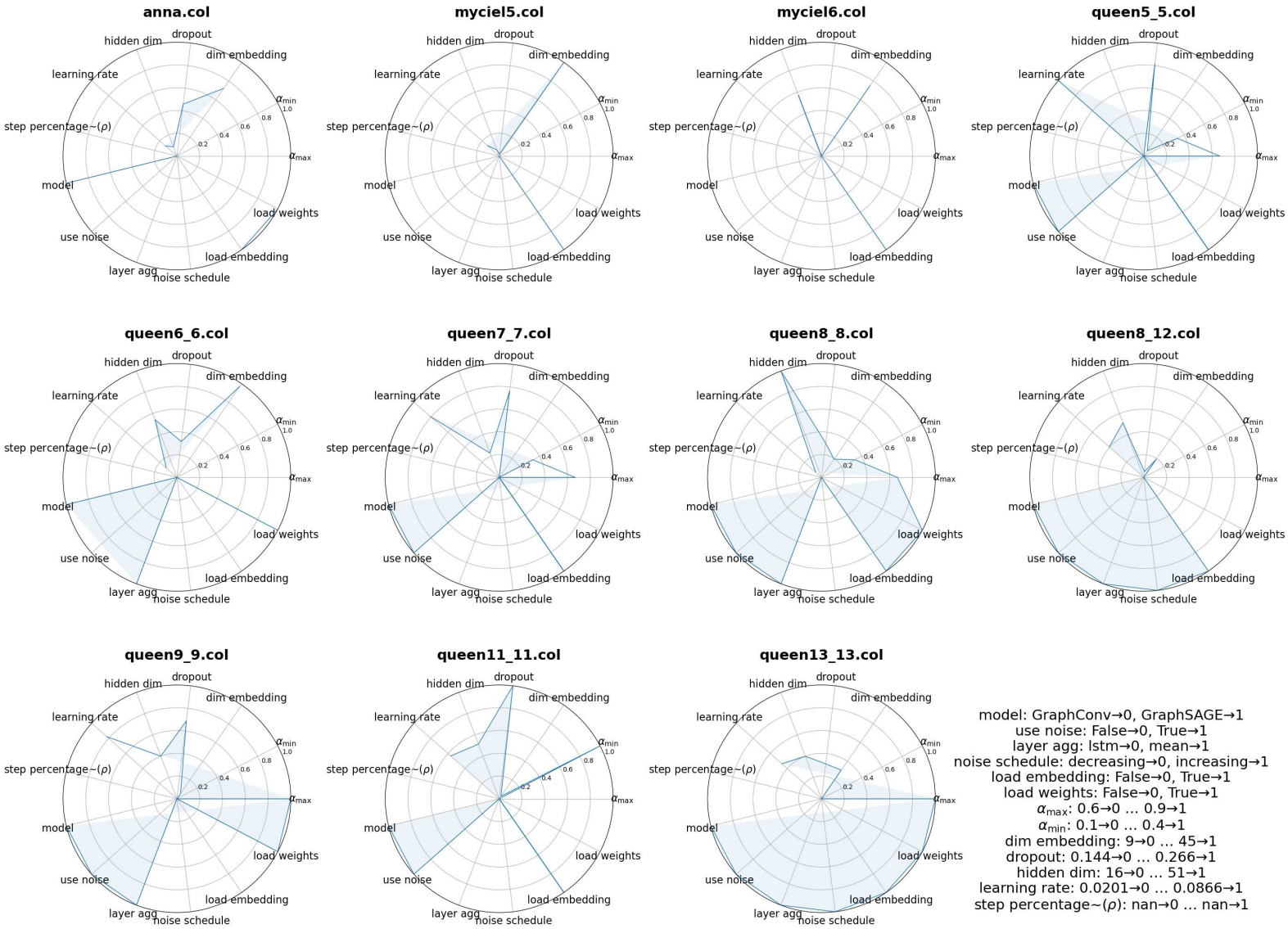


Figure 4.1 Min-max–scaled radar charts for the **layer by layer BFS expansion** strategy. Each subplot shows the selected hyperparameter of the graphs; metric ranges and categorical-code mappings appear beneath chart.

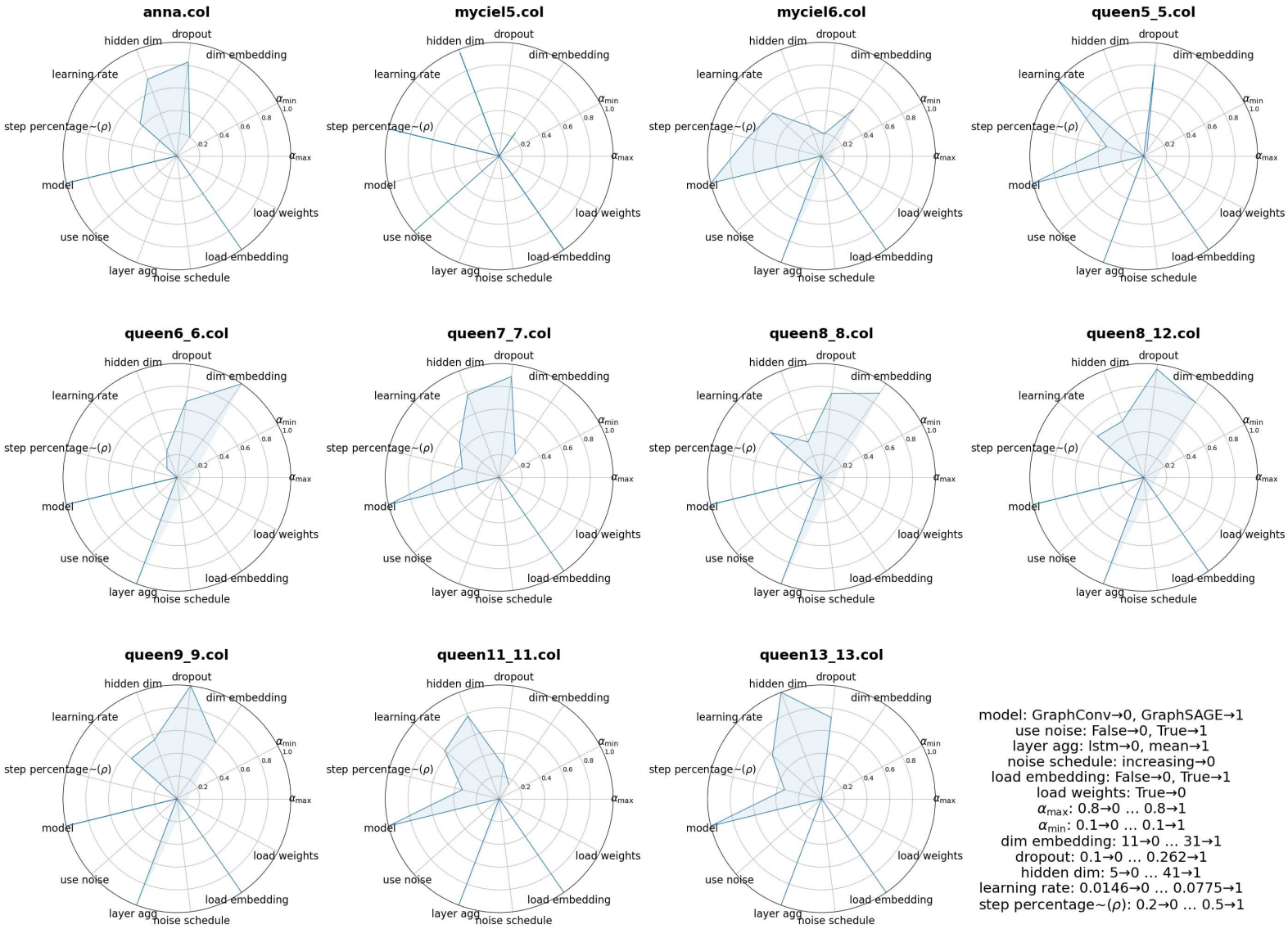


Figure 4.2 Min-max–scaled radar charts for the **degree first expansion** strategy. Each subplot shows the selected hyperparameter of the graphs; metric ranges and categorical-code mappings appear beneath chart.

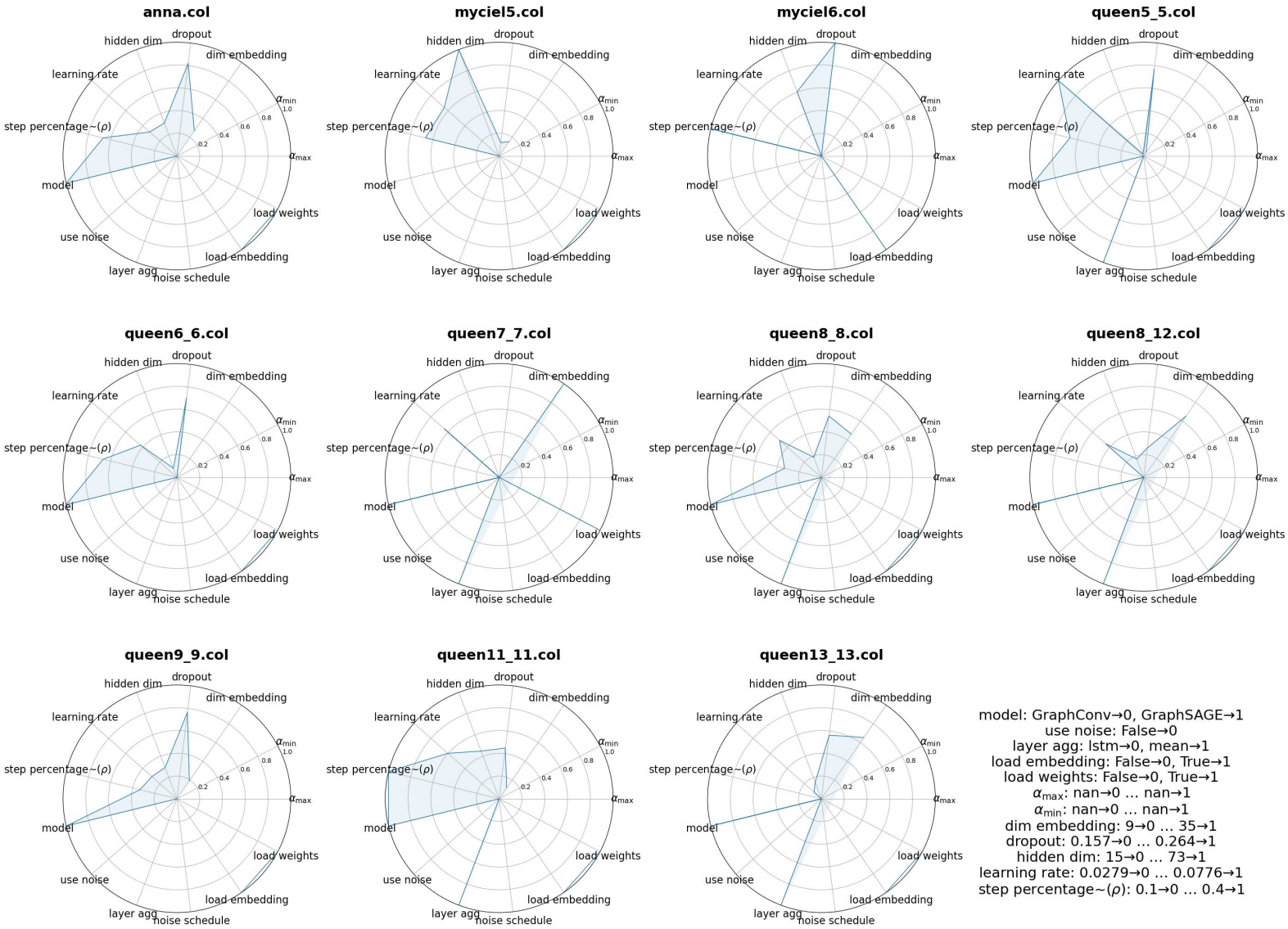


Figure 4.3 Min-max–scaled radar charts for the **biased random walk** strategy. Each subplot shows the selected hyperparameter of the graphs; metric ranges and categorical-code mappings appear beneath chart.

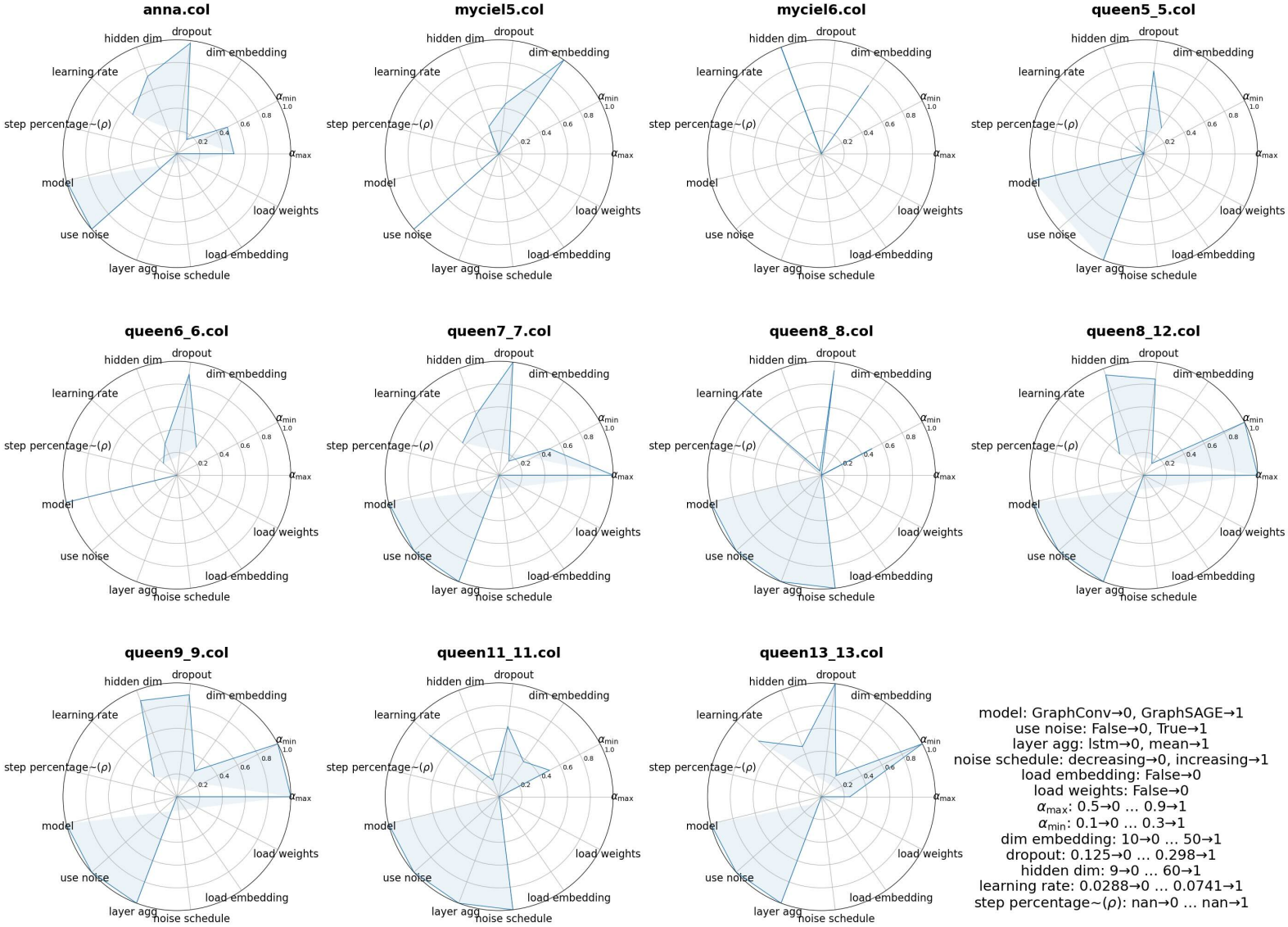


Figure 4.4 Min-max–scaled radar charts for the **full graph** strategy. Each subplot shows the selected hyperparameter of the graphs; metric ranges and categorical-code mappings appear beneath chart.

Table 4.11 Distribution of hyper-parameter values explored during tuning for The layer-by-layer BFS-based expansion. For every problem group (frequency of each category).

problem group	layer agg	load embedding	load weights	model	noise schedule	use noise
anna	lstm: 1	True: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1
myciel	nan: 2	True: 2	False: 2	GraphConv: 2	nan: 2	False: 2
queen	mean: 5, lstm: 3	True: 6, False: 2	False: 4, True: 4	GraphSAGE: 8	decreasing: 5, increasing: 2, nan: 1	True: 7, False: 1

Table 4.12 Distribution of hyper-parameter values explored during tuning for degree-first BFS-based expansion. For every problem group (frequency of each category).

problem group	layer agg	load embedding	load weights	model	noise schedule	use noise
anna	lstm: 1	True: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1
myciel	mean: 1, nan: 1	True: 2	True: 2	GraphConv: 1, GraphSAGE: 1	increasing: 1, nan: 1	False: 1, True: 1
queen	mean: 7, lstm: 1	True: 6, False: 2	True: 8	GraphSAGE: 8	nan: 8	False: 8

Table 4.13 Distribution of hyper-parameter values explored during tuning for random-walk expansion. For every problem group (frequency of each category).

problem group	layer agg	load embedding	load weights	model	noise schedule	use noise
anna	lstm: 1	True: 1	True: 1	GraphSAGE: 1	nan: 1	False: 1
myciel	nan: 2	True: 2	False: 1, True: 1	GraphConv: 2	nan: 2	False: 2
queen	mean: 6, lstm: 2	True: 7, False: 1	True: 8	GraphSAGE: 8	nan: 8	False: 8

Table 4.14 Distribution of hyper-parameter values explored during tuning for The baseline (no expansion). For every problem group (frequency of each category).

problem group	layer agg	load embedding	load weights	model	noise schedule	use noise
anna	lstm: 1	False: 1	False: 1	GraphSAGE: 1	decreasing: 1	True: 1
myciel	nan: 2	False: 2	False: 2	GraphConv: 2	decreasing: 1, nan: 1	False: 1, True: 1
queen	mean: 7, lstm: 1	False: 8	False: 8	GraphSAGE: 8	decreasing: 4, increasing: 2, nan: 2	True: 6, False: 2

Table 4.15 Distribution of hyper-parameter values explored during tuning for The layer-by-layer BFS-based expansion. For every problem group (minimum / average / maximum / standard deviation).

problem group	$\alpha_{\max}$	$\alpha_{\min}$	dim embedding	dropout	hidden dim	learning rate	step percentage (ρ)
anna	nan / nan / nan / nan	nan / nan / nan / nan	35 / 35 / 35 / nan	0.2001 / 0.2001 / 0.2001 / nan	19 / 19 / 19 / nan	0.029 / 0.029 / 0.029 / nan	nan / nan / nan / nan
myciel	nan / nan / nan / nan	nan / nan / nan / nan	36 / 40.5 / 45 / 6.364	0.1437 / 0.1448 / 0.1458 / 0.0015	18 / 27 / 36 / 12.73	0.0201 / 0.0247 / 0.0292 / 0.0064	nan / nan / nan / nan
queen	0.6 / 0.7714 / 0.9 / 0.1254	0.1 / 0.1857 / 0.4 / 0.1069	9 / 17.12 / 44 / 11.49	0.1499 / 0.209 / 0.2665 / 0.0405	16 / 31.75 / 51 / 10.04	0.0247 / 0.0555 / 0.0866 / 0.0222	nan / nan / nan / nan

Table 4.16 Distribution of hyper-parameter values explored during tuning for degree-first BFS-based expansion. For every problem group (minimum / average / maximum / standard deviation).

problem group	$\alpha_{\max}$	$\alpha_{\min}$	dim embedding	dropout	hidden dim	learning rate	step percentage (ρ)
anna	nan / nan / nan / nan	nan / nan / nan / nan	15 / 15 / 15 / nan	0.2352 / 0.2352 / 0.2352 / nan	31 / 31 / 31 / nan	0.0417 / 0.0417 / 0.0417 / nan	0.2 / 0.2 / 0.2 / nan
myciel	0.8 / 0.8 / 0.8 / nan	0.1 / 0.1 / 0.1 / nan	16 / 18.5 / 21 / 3.535	0.1002 / 0.1161 / 0.1319 / 0.0224	15 / 27.5 / 40 / 17.68	0.0146 / 0.0325 / 0.0503 / 0.0253	0.4 / 0.45 / 0.5 / 0.0707
queen	nan / nan / nan / nan	nan / nan / nan / nan	11 / 20.38 / 31 / 8.07	0.1484 / 0.2239 / 0.2623 / 0.0358	5 / 24 / 41 / 11.72	0.0219 / 0.0497 / 0.0775 / 0.0152	0.2 / 0.25 / 0.3 / 0.0535

Table 4.17 Distribution of hyper-parameter values explored during tuning for random-walk expansion. For every problem group (minimum / average / maximum / standard deviation).

problem group	$\alpha_{\max}$	$\alpha_{\min}$	dim embedding	dropout	hidden dim	learning rate	step percentage (ρ)
anna	nan / nan / nan / nan	nan / nan / nan / nan	16 / 16 / 16 / nan	0.245 / 0.245 / 0.245 / nan	33 / 33 / 33 / nan	0.0437 / 0.0437 / 0.0437 / nan	0.3 / 0.3 / 0.3 / nan
myciel	nan / nan / nan / nan	nan / nan / nan / nan	9 / 11 / 13 / 2.828	0.1701 / 0.2171 / 0.2641 / 0.0665	50 / 61.5 / 73 / 16.26	0.0279 / 0.0439 / 0.0599 / 0.0226	0.3 / 0.35 / 0.4 / 0.0707
queen	nan / nan / nan / nan	nan / nan / nan / nan	9 / 19.12 / 35 / 9.357	0.1574 / 0.2115 / 0.2397 / 0.0286	15 / 24.88 / 41 / 8.56	0.0322 / 0.0527 / 0.0776 / 0.0133	0.1 / 0.2125 / 0.4 / 0.1126

Table 4.18 Distribution of hyper-parameter values explored during tuning for The baseline (no expansion). For every problem group (minimum / average / maximum / standard deviation).

problem group	$\alpha_{\max}$	$\alpha_{\min}$	dim embedding	dropout	hidden dim	learning rate	step percentage (ρ)
anna	0.7 / 0.7 / 0.7 / nan	0.2 / 0.2 / 0.2 / nan	16 / 16 / 16 / nan	0.2938 / 0.2938 / 0.2938 / nan	46 / 46 / 46 / nan	0.0523 / 0.0523 / 0.0523 / nan	nan / nan / nan / nan
myciel	0.5 / 0.5 / 0.5 / nan	0.1 / 0.1 / 0.1 / nan	39 / 44.5 / 50 / 7.778	0.1247 / 0.1626 / 0.2005 / 0.0536	22 / 41 / 60 / 26.87	0.0288 / 0.0292 / 0.0295 / 0.0005	nan / nan / nan / nan
queen	0.5 / 0.7167 / 0.9 / 0.2041	0.2 / 0.25 / 0.3 / 0.0548	10 / 18.62 / 25 / 4.749	0.2323 / 0.2744 / 0.2977 / 0.0225	9 / 30.5 / 57 / 18.64	0.0329 / 0.0502 / 0.0741 / 0.0152	nan / nan / nan / nan

4.3 Experimental Setup

This study was implemented based on the work of Schuetz et al. (2022) which runs the experiments in a two-file structure. The first file is a Jupyter notebook that orchestrates the whole workflow, and the second file is a Python file that keeps the essential functions for data processing, subgraph expansion, training loops, and performance evaluations.

4.3.1 Hardware and Operating Environment

MacBook Pro (M1 Pro) 16 inch base model was used for a large portion of the experiments. It was running on macOS 15.4. The system has a 10-core CPU (composed of 8 performance cores and 2 efficiency cores) and has a 16-core GPU integrated to the M1 Pro chip. However, the code was executed on the CPU only, because library constraints prevented the use of the integrated GPU. The device also includes 16 GB unified memory.

Windows Machine setup was used for the additional experiments. The system was running on Windows 11. The setup consisted of an AMD Ryzen 9 7900 (12 cores / 24 threads) CPU, a NVIDIA RTX 4070 12 GB GPU and 32 GB of DDR5 RAM at 6400 MHz.

4.3.2 Code Architecture and Workflow

This subsection provides details of the Jupyter notebook and the Python utility script. It contains details on how the model collaborates to execute the subgraph expansion approaches, train the GNN model, and record the performance metrics.

The Jupyter Notebook is the main file that orchestrates the experimental routine. It includes stages such as

- *Imports and Initialization*: Loads Python packages such as PyTorch, Ray, Optuna, DGL and the Python utility script, and initializes the seed and setup parameters.

- *Dataset Loading and Preprocessing*: Loads and prepares the graphs
- *Subgraph Expansion Setup*: Configures the expansion method and parameters.
- *Hyperparameter Tuning*: Uses Ray for parallel execution and Optuna for the hyperparameter search algorithm. Creates an objective function that runs the training using a set of sampled hyperparameters.

The Python Utility Script was designed so that the Jupyter notebook can invoke them as needed. It is used for essential operations, such as

- *Graph Loading and Parsing*: Reads the graph file (.col) and builds an internal representation on the networkx.
- *Expansion Strategies*: Provides multiple expansion strategies such as Layer-by-Layer BFS Expansion, Degree-first BFS-Based Expansion and Random Walk Expansion to break down a full graph into a sequence of incrementally larger subgraphs until the full graph is obtained.
- *Noise Injection*: Add random Gaussian noise to node embeddings as needed with linear increasing or decreasing functions over training time.
- *Loss Function*: Creates a soft loss function to train the model, and creates a hard loss function to track the conflict count.
- *Training Setup*:
  - Creates a GNN model with configurable architecture (GraphSAGE, GraphConv), hidden layers, aggregation type, and dropout.
  - Runs training loops for each subgraph.
  - Loads previously saved embeddings or model weights if needed.
  - Apply early stopping if a perfect solution (0 conflict) is found for a subgraph or a full graph.
- *Evaluation and Final Reporting*: Shows the final metrics for each subgraph training and final training, such as soft loss, conflict count, assigned colors for each nodes.

4.3.3 Experiment Design

This study consists of a comprehensive set of experiments to evaluate the effect of subgraph expansion. The key points of the experiments include the following.

- *Known Chromatic Numbers*: This is the minimum number of colors with which the graph instance can be colored without any clashes. Each experiment is designed in a way that searches for node coloring so that it finds 0 clashes.
- *Test Instances*: The instances (see Section 4.1) were selected in a way that they cover easy instances such as Myciel graphs, as well as harder instances such as larger Queen graphs. By testing a wide range of graphs, the experiments evaluated the effectiveness of subgraph expansion for a variety of graph types and sizes.
- *Subgraph Expansion Strategies*: The experiments were conducted on different expansion strategies such as layer-by-layer BFS-based expansion, degree-first BFS-based expansion, and random walk expansion. The expansion step size is adjustable for different strategies to examine the sensitivity of the performance to step size.
- *Baseline Comparisons*: For each of the graph instances, experiments include the baseline version without expansion. The non-expansion version tuned over the same hyperparameter search space as expansion strategies.
- *Hyperparameters*: In the experiments, the GNN architecture and hyperparameters were kept the same for expansion and non-expansion runs.

In this study, four different experiments were conducted to evaluate the effect of various subgraph expansion methods on the performance of graph coloring. In all experiments, the same hyperparameter search spaces (including GNN architecture alternatives) were used, as depicted in Table 4.19. Ray framework was used to run all of the experiments in parallel, and the Optuna TPE (Tree-structured Parzen Estimator) algorithm was selected as the hyperparameter search algorithm. The search algorithm parameters were kept at their default values. All experiments were run for 150 times, because 50 distinct hyperparameter configurations were budgeted, and each configuration was trained with the same three seeds to ensure reproducibility and fair comparison.

Experiment 1: The baseline (no expansion) serves as a control experiment by training the GNN model on the full graph without any expansion strategy. The aim of the experiment is to create a performance baseline against the subgraph expansion methods.

Experiment 2: The layer-by-layer BFS-based expansion evaluates the effects

of using a breadth-first search (BFS) approach.

Experiment 3: degree-first BFS-based expansion. In addition to the hyperparameters of Experiment 2, this experiment had the expansion step percentage as a hyperparameter. The objective of this experiment is to see whether an early loading graph with high connectivity nodes improves the learning efficiency and the quality of the solution.

Experiment 4: The random-walk expansion analyzes the impact of a random-walk strategy. As in Experiment 3, this experiment also had the expansion step percentage as a hyperparameter. The objective of this experiment is to see whether a stochastic expansion strategy can better capture the structural diversity of the graph, leading to improved training performance.

Table 4.19 Hyperparameter Search Space for Tuning

Parameter	**Type/Distribution**	**Range/Choices**	**Description**
model	Categorical	GraphSAGE, GraphConv	Model architecture
dim_embedding	Integer	5 – 50	Dimension of node embeddings
dropout	Float	0.1 – 0.3	Dropout probability
learning_rate	Float	0.01 – 0.1	Learning rate for optimization
hidden_dim	Integer	5 – 80	Dimension of hidden layers
step_percentage	Categorical	0.1, 0.2, 0.3, 0.4, 0.5	Parameters for subgraph step size
layer_agg	Categorical	lstm, mean	Aggregation type for neighborhood information
embedding_weights	Categorical	• { "load_embedding": True, "load_weights": True } • { "load_embedding": True, "load_weights": False } • { "load_embedding": False, "load_weights": True }	Options for initializing embeddings and weights
use_noise	Boolean	True, False	Noise injection flag
noise_scale_fn	Function	linear_alpha_schedule, linear_alpha_schedule_decreasing	Function for noise scaling schedule
noise_scale_params alpha_min	Categorical	0.1, 0.2, 0.3, 0.4	Parameter for noise scaling
noise_scale_params alpha_max	Categorical	0.5, 0.6, 0.7, 0.8, 0.9	Parameter for noise scaling

In summary, experiments were conducted systematically to examine the effect of subgraph expansion by comparing against the non-expansion method.

4.4 Results

This section presents the results of the computational experiments outlined in the previous sections. The performance of the three proposed subgraph expansion strategies *Layer-by-Layer BFS-Based Expansion* (`bfs_layer_by_layer`), *Degree-first BFS-Based Expansion* `bfs_degree`, and *Random Walk Expansion* `random_walk` is compared against the *baseline (no expansion)* (`full_graph`). The evaluation is centered on three key performance metrics: the quality of the final solution (hard cost, i.e., the number of conflicting edges), the computational time (runtime), and the convergence speed (total number of training epochs). For each metric, a summary of the descriptive statistics is presented (see Tables 4.20, 4.23, 4.26) for 50 independent runs for each graph. Then, the results of Welch's t-tests are presented (see Tables 4.21, 4.22, 4.24, 4.25, 4.27,4.28) to determine whether the observed differences between the averages for the expansion strategies and the baseline are statistically significant.

The aim of the k-color vertex coloring is to find a valid coloring, which corresponds to a hard cost of zero. If a valid coloring is not found, a lower hard cost indicates a higher quality solution. Table 4.20 provides a summary of the hard costs achieved by each strategy across the benchmark instances. For the anna.col, myciel5.col, myciel6.col and queen5_5.col graphs, all strategies found a valid coloring with a hard cost of 0. In the queen5_5.col instance, the `bfs_degree` and `random_walk` strategies found a valid coloring in every run. A significant improvement over the baseline, which had a mean hard cost of 1.62. However, the solution quality was diminished on more complex graphs. On the larger queen graphs (e.g., queen7_7.col through queen13_13.col), the baseline approach consistently achieved a lower mean hard cost than the expansion strategies with the sole exception of queen11_11.col, where `bfs_degree` obtained the best mean cost. To verify whether the observed differences in hard cost are statistically significant, Welch's t-tests were performed on the hard cost scores. Table 4.21 presents the two-tailed Welch's t-test results, comparing the average hard cost of each expansion strategy with the baseline. The queen9_9.col and queen11_11.col instances have 0.34 and 0.67 two-tailed p-values respectively. In both cases, the null hypothesis is not rejected at the conventional

significance threshold of $\alpha = 0.05$. This means that `bfs_degree` does not produce a statistically different mean hard cost from the `full_graph` on these two graphs. The one-tailed tests in Table 4.22 complement this analysis. Notably, the very small p-values on myciel6.col and queen5_5.col for `bfs_layer_by_layer`, `bfs_degree` and `random_walk` show that expansion strategies improve the solution quality on simpler instances.

Table 4.20 Hard cost (number of colour clashes) obtained by the four search strategies—*Layer-by-Layer BFS-Based Expansion*, *Degree-first BFS-Based Expansion*, *Random Walk Expansion*, and *The Baseline (No Expansion)*—on each vertex-colouring benchmark instance. Each cell gives the **minimum / mean / maximum / standard-deviation** across 50 independent runs; lower values are better.

Best cost hard	bfs_layer_by_layer	bfs_degree	random_walk	full_graph
anna.col	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 0 / 0
myciel5.col	0 / 0 / 0 / 0	0 / 0.04 / 1 / 0.20	0 / 4.72 / 236 / 33.38	0 / 0 / 0 / 0
myciel6.col	0 / 0.08 / 1 / 0.27	0 / 0 / 0 / 0	0 / 0.06 / 1 / 0.24	0 / 0.26 / 2 / 0.49
queen5_5.col	0 / 0.14 / 3 / 0.57	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 1.62 / 6 / 2.45
queen6_6.col	0 / 2.60 / 10 / 1.48	1 / 1.98 / 3 / 0.71	1 / 2.06 / 4 / 0.82	0 / 1.98 / 4 / 0.82
queen7_7.col	5 / 9.50 / 14 / 1.74	5 / 9.26 / 13 / 1.72	0 / 8.46 / 12 / 1.97	0 / 7.60 / 10 / 1.51
queen8_8.col	3 / 6.86 / 11 / 1.65	1 / 4.12 / 7 / 1.47	1 / 3.84 / 7 / 1.22	0 / 2.28 / 5 / 0.99
queen8_12.col	0 / 5.62 / 12 / 2.52	0 / 3.20 / 6 / 1.36	1 / 2.68 / 5 / 1.15	0 / 2.30 / 5 / 1.15
queen9_9.col	2 / 7.02 / 13 / 2.66	3 / 5.38 / 9 / 1.48	5 / 8.64 / 13 / 1.64	2 / 5.08 / 9 / 1.64
queen11_11.col	21 / 28.60 / 33 / 2.47	15 / 20.12 / 25 / 2.07	17 / 21.58 / 27 / 2.37	17 / 20.28 / 24 / 1.65
queen13_13.col	30 / 35.24 / 40 / 2.45	25 / 29.94 / 36 / 2.58	26 / 33.30 / 40 / 2.91	22 / 28.54 / 33 / 2.28

Table 4.21 Comparison of strategies against the *full_graph* baseline: Two-Tailed Welch t-test p-values for *best cost hard* ($n = 50$ runs per strategy). Stars: $* = p < 0.05$, $** = p < 0.01$, $*** = p < 0.001$.

Strategy Problem File	bfs_layer_by_layer	bfs_degree	random_walk
anna.col	–	–	–
myciel5.col	–	1.59e-01	3.22e-01
myciel6.col	2.55e-02*	4.32e-04***	1.12e-02*
queen5_5.col	1.13e-04***	2.31e-05***	2.31e-05***
queen6_6.col	1.17e-02*	1.00e+00	6.27e-01
queen7_7.col	7.51e-08***	1.57e-06***	1.63e-02*
queen8_8.col	5.53e-28***	1.05e-10***	3.32e-10***
queen8_12.col	2.91e-12***	5.35e-04***	1.01e-01
queen9_9.col	3.38e-05***	3.40e-01	1.65e-18***
queen11_11.col	1.23e-33***	6.70e-01	2.01e-03**
queen13_13.col	2.04e-25***	4.98e-03**	1.61e-14***

Table 4.22 Comparison of strategies against the *full_graph* baseline: One-Tailed Welch t-test p-values for *best cost hard* ($n = 50$ runs per strategy). Stars: $* = p < 0.05$, $** = p < 0.01$, $*** = p < 0.001$.

Strategy Problem File	bfs_layer_by_layer	bfs_degree	random_walk
anna.col	–	–	–
myciel5.col	–	9.20e-01	8.39e-01
myciel6.col	1.28e-02*	2.16e-04***	5.58e-03**
queen5_5.col	5.66e-05***	1.15e-05***	1.15e-05***
queen6_6.col	9.94e-01	5.00e-01	6.87e-01
queen7_7.col	1.00e+00	1.00e+00	9.92e-01
queen8_8.col	1.00e+00	1.00e+00	1.00e+00
queen8_12.col	1.00e+00	1.00e+00	9.49e-01
queen9_9.col	1.00e+00	8.30e-01	1.00e+00
queen11_11.col	1.00e+00	3.35e-01	9.99e-01
queen13_13.col	1.00e+00	9.98e-01	1.00e+00

Beyond solution quality, efficiency of an algorithm is also critical, especially for large instances. In this study, efficiency is assessed through the total runtime and the number of epochs required for convergence. As shown in Table 4.23, the subgraph expansion strategies often lead to a substantial reduction in runtime. For example, `bfs_degree` strategy demonstrates a strong overall performance in terms of efficiency. In most cases, it is significantly faster than training for `full_graph`. This general reduction in time is strongly correlated with the number of training epochs(see Table 4.26). The statistical significance of these findings is confirmed by the t-tests presented in Tables 4.24, 4.25, 4.27 and 4.28. The expansion strategies are significantly faster than the baseline with p-values less than 0.001 ($p < 0.001$) for most of the graph instances.

Table 4.23 Runtime in seconds required by each search strategy to reach its final solution on every benchmark graph. Entries show the **minimum / mean / maximum / standard-deviation** over 50 runs; lower values indicate faster execution.

Time total s	bfs_layer_by_layer	bfs_degree	random_walk	full_graph
anna.col	7 / 315 / 3833 / 591	6 / 229 / 608 / 135	14 / 449 / 3324 / 566	234 / 932 / 2590 / 575
myciel5.col	1 / 11 / 98 / 22	1 / 9 / 68 / 18	1 / 4 / 32 / 5	1 / 11 / 160 / 30
myciel6.col	1 / 91 / 567 / 167	1 / 20 / 110 / 27	3 / 75 / 605 / 158	1 / 100 / 1161 / 226
queen5_5.col	6 / 226 / 655 / 167	2 / 17 / 86 / 17	2 / 40 / 209 / 43	1 / 180 / 441 / 195
queen6_6.col	26 / 238 / 312 / 71	255 / 378 / 519 / 86	650 / 1046 / 1212 / 128	1021 / 1167 / 1201 / 34
queen7_7.col	647 / 740 / 893 / 48	1562 / 1937 / 2181 / 189	1017 / 1297 / 1652 / 169	752 / 900 / 937 / 24
queen8_8.col	509 / 534 / 586 / 17	524 / 763 / 924 / 119	576 / 866 / 1222 / 164	757 / 1247 / 1290 / 83
queen8_12.col	963 / 1155 / 1208 / 52	942 / 1460 / 2236 / 342	1636 / 2410 / 3113 / 340	1502 / 2171 / 2224 / 117
queen9_9.col	682 / 836 / 878 / 58	868 / 1269 / 1580 / 150	2359 / 2904 / 3730 / 271	1682 / 1781 / 1903 / 57
queen11_11.col	1968 / 2154 / 2322 / 84	2681 / 3064 / 3325 / 163	2110 / 2233 / 2488 / 78	2918 / 3041 / 3105 / 37
queen13_13.col	2086 / 2287 / 2362 / 55	4152 / 4880 / 5250 / 229	6041 / 6774 / 7360 / 272	4764 / 4982 / 5075 / 59

Table 4.24 Comparison of strategies against the *full_graph* baseline: Two-Tailed Welch t-test p-values for *time total s* ($n = 50$ runs per strategy). Stars: $* = p < 0.05$, $** = p < 0.01$, $*** = p < 0.001$.

Strategy Problem File	bfs_layer_by_layer	bfs_degree	random_walk
anna.col	7.61e-07***	2.06e-11***	5.24e-05***
myciel5.col	9.01e-01	6.30e-01	7.47e-02
myciel6.col	8.28e-01	1.68e-02*	5.31e-01
queen5_5.col	2.11e-01	3.03e-07***	6.97e-06***
queen6_6.col	1.59e-71***	8.45e-58***	2.41e-08***
queen7_7.col	2.67e-32***	4.32e-39***	5.38e-22***
queen8_8.col	3.09e-50***	1.27e-39***	2.64e-23***
queen8_12.col	1.04e-58***	1.76e-20***	1.53e-05***
queen9_9.col	6.19e-92***	7.07e-32***	4.13e-34***
queen11_11.col	6.30e-64***	3.42e-01	1.49e-64***
queen13_13.col	1.77e-136***	3.63e-03**	1.77e-44***

Table 4.25 Comparison of strategies against the *full_graph* baseline: One-Tailed Welch t-test p-values for *time total s* ($n = 50$ runs per strategy). Stars: $* = p < 0.05$, $** = p < 0.01$, $*** = p < 0.001$.

Strategy Problem File	bfs_layer_by_layer	bfs_degree	random_walk
anna.col	3.81e-07***	1.03e-11***	2.62e-05***
myciel5.col	4.50e-01	3.15e-01	3.73e-02*
myciel6.col	4.14e-01	8.38e-03**	2.65e-01
queen5_5.col	8.94e-01	1.52e-07***	3.48e-06***
queen6_6.col	7.95e-72***	4.23e-58***	1.20e-08***
queen7_7.col	1.33e-32***	1.00e+00	1.00e+00
queen8_8.col	1.54e-50***	6.36e-40***	1.32e-23***
queen8_12.col	5.21e-59***	8.80e-21***	1.00e+00
queen9_9.col	3.09e-92***	3.54e-32***	1.00e+00
queen11_11.col	3.15e-64***	8.29e-01	7.43e-65***
queen13_13.col	8.87e-137***	1.82e-03**	1.00e+00

Table 4.26 Total number of epochs executed before termination for each strategy on each benchmark instance. Values are reported as **minimum / mean / maximum / standard-deviation** across 50 runs; fewer epochs denote quicker convergence.

Total epoch num	bfs_layer_by_layer	bfs_degree	random_walk	full_graph
anna.col	156 / 8419 / 96916 / 14926	178 / 9126 / 25187 / 5330	443 / 11874 / 84250 / 14301	5428 / 21593 / 61367 / 13526
myciel5.col	75 / 1661 / 16037 / 3626	36 / 1372 / 11052 / 2925	114 / 531 / 4915 / 876	51 / 1794 / 25799 / 4844
myciel6.col	69 / 6502 / 40011 / 11975	49 / 1611 / 9991 / 2191	236 / 5235 / 41278 / 10816	41 / 7139 / 84288 / 16328
queen5_5.col	639 / 27570 / 80001 / 20575	389 / 5293 / 38392 / 6317	401 / 9364 / 46880 / 9938	214 / 41870 / 100000 / 45312
queen6_6.col	3940 / 37920 / 54379 / 10746	42235 / 66282 / 96418 / 16305	52114 / 77816 / 87685 / 8249	86622 / 99464 / 100000 / 2563
queen7_7.col	40329 / 45872 / 58693 / 4296	86251 / 110782 / 126001 / 12973	123931 / 176770 / 262721 / 34099	83914 / 99678 / 100000 / 2275
queen8_8.col	40039 / 43501 / 51062 / 3287	42056 / 69206 / 89592 / 12986	43166 / 74951 / 120082 / 18817	59628 / 98734 / 100000 / 6507
queen8_12.col	49436 / 78424 / 80001 / 5945	44333 / 88193 / 159325 / 33108	78887 / 124315 / 169531 / 22808	67738 / 98824 / 100000 / 5268
queen9_9.col	40060 / 72306 / 80001 / 14540	57097 / 87304 / 125347 / 15063	77536 / 89757 / 120063 / 8931	100000 / 100000 / 100000 / 0
queen11_11.col	40101 / 48503 / 57334 / 3897	101798 / 138949 / 160000 / 17941	80068 / 88348 / 119892 / 9957	100000 / 100000 / 100000 / 0
queen13_13.col	47988 / 78428 / 80001 / 6296	114880 / 150156 / 160000 / 12865	148548 / 180770 / 215540 / 12385	100000 / 100000 / 100000 / 0

Table 4.27 Comparison of strategies against the *full_graph* baseline: Two-Tailed Welch t-test p-values for *total epoch num* ($n = 50$ runs per strategy). Stars: $* = p < 0.05$, $** = p < 0.01$, $*** = p < 0.001$.

Strategy Problem File	bfs_layer_by_layer	bfs_degree	random_walk
anna.col	1.16e-05***	7.93e-08***	7.23e-04***
myciel5.col	8.77e-01	5.99e-01	7.53e-02
myciel6.col	8.24e-01	2.15e-02*	4.94e-01
queen5_5.col	4.61e-02*	7.17e-07***	7.60e-06***
queen6_6.col	9.26e-42***	1.90e-19***	3.65e-25***
queen7_7.col	2.84e-73***	2.21e-07***	3.94e-21***
queen8_8.col	4.80e-60***	7.19e-23***	7.94e-12***
queen8_12.col	6.57e-33***	2.93e-02*	2.94e-10***
queen9_9.col	4.31e-18***	2.70e-07***	1.29e-10***
queen11_11.col	7.11e-57***	2.36e-20***	7.29e-11***
queen13_13.col	6.28e-29***	1.71e-31***	5.00e-42***

Table 4.28 Comparison of strategies against the *full_graph* baseline: One-Tailed Welch t-test p-values for *total epoch num* ($n = 50$ runs per strategy). Stars: $* = p < 0.05$, $** = p < 0.01$, $*** = p < 0.001$.

Strategy Problem File	bfs_layer_by_layer	bfs_degree	random_walk
anna.col	5.79e-06***	3.97e-08***	3.62e-04***
myciel5.col	4.39e-01	3.00e-01	3.77e-02*
myciel6.col	4.12e-01	1.07e-02*	2.47e-01
queen5_5.col	2.30e-02*	3.58e-07***	3.80e-06***
queen6_6.col	4.63e-42***	9.49e-20***	1.83e-25***
queen7_7.col	1.42e-73***	1.00e+00	1.00e+00
queen8_8.col	2.40e-60***	3.59e-23***	3.97e-12***
queen8_12.col	3.29e-33***	1.46e-02*	1.00e+00
queen9_9.col	2.15e-18***	1.35e-07***	6.47e-11***
queen11_11.col	3.56e-57***	1.00e+00	3.64e-11***
queen13_13.col	3.14e-29***	1.00e+00	1.00e+00

Table 4.29—4.31 reports the *mean percentage reduction* (positive values indicate an improvement) obtained by the three expansion strategies with respect to the `full_graph`.

Table 4.29 Mean percentage reduction of *bfs_layer_by_layer* relative to the *full_graph* baseline (+ values = improvement; 50 runs per instance). An en-dash (—) marks instances where the baseline value is zero, so the reduction percentage is undefined.

metric problem_file	best cost hard	time total s	total epoch num
anna.col	–	66.2	61.0
myciel5.col	–	0.0	7.4
myciel6.col	69.2	9.0	8.9
queen11_11.col	-41.0	29.2	51.5
queen13_13.col	-23.5	54.1	21.6
queen5_5.col	91.4	-25.6	34.2
queen6_6.col	-31.3	79.6	61.9
queen7_7.col	-25.0	17.8	54.0
queen8_12.col	-144.3	46.8	20.6
queen8_8.col	-200.9	57.2	55.9
queen9_9.col	-38.2	53.1	27.7

Table 4.30 Mean percentage reduction of *bfs_degree* relative to the *full_graph* baseline (+ values = improvement; 50 runs per instance). An en-dash (—) marks instances where the baseline value is zero, so the reduction percentage is undefined.

metric problem_file	best cost hard	time total s	total epoch num
anna.col	–	75.4	57.7
myciel5.col	–	18.2	23.5
myciel6.col	100.0	80.0	77.4
queen11_11.col	0.8	-0.8	-38.9
queen13_13.col	-4.9	2.0	-50.2
queen5_5.col	100.0	90.6	87.4
queen6_6.col	0.0	67.6	33.4
queen7_7.col	-21.8	-115.2	-11.1
queen8_12.col	-39.1	32.7	10.8
queen8_8.col	-80.7	38.8	29.9
queen9_9.col	-5.9	28.7	12.7

Table 4.31 Mean percentage reduction of *random_walk* relative to the *full_graph* baseline (+ values = improvement; 50 runs per instance). An en-dash (—) marks instances where the baseline value is zero, so the reduction percentage is undefined.

metric problem_file	best cost hard	time total s	total epoch num
anna.col	–	51.8	45.0
myciel5.col	–	63.6	70.4
myciel6.col	76.9	25.0	26.7
queen11_11.col	-6.4	26.6	11.7
queen13_13.col	-16.7	-36.0	-80.8
queen5_5.col	100.0	77.8	77.6
queen6_6.col	-4.0	10.4	21.8
queen7_7.col	-11.3	-44.1	-77.3
queen8_12.col	-16.5	-11.0	-25.8
queen8_8.col	-68.4	30.6	24.1
queen9_9.col	-70.1	-63.1	10.2

5. Discussion and Conclusions

This chapter consolidates the findings of the computational experiments. The chapter begins with a discussion of key observations, then discusses the limitations of the study, and outlines promising avenues for future research. The chapter concludes with a summary of the thesis and its primary contributions.

5.1 Discussion of Findings

The experiments confirm that the subgraph expansion strategy can reduce computational cost without losing solution quality but the balance depends heavily on the expansion strategy. The expectation was that the subgraph expansion strategies would also increase the quality of the solutions. However, this improvement is not observed in all graph instances, as can be seen in the results. Instead, the experimental data reveal a clear and compelling trade-off between solution quality and computational efficiency. `full_graph` training still secures the lowest hard costs on the complex instances, but `bfs_degree` achieves comparable quality while converging far more quickly for the majority of benchmark instances.

Among the expansion strategies, `bfs_degree` presents a particularly attractive balance between performance and efficiency. As shown in Table 4.21, for several large instances, such as queen6_6.col, queen9_9.col, and queen11_11.col, there is no statistically significant difference in the final hard cost between `bfs_degree` and the `full_graph` baseline. In other cases, `bfs_degree` achieved a significantly better solution quality ($p < 0.001$) for myciel6.col and queen5_5.col. Although the baseline did outperform on some of the most constrained Queen graphs, `bfs_degree` proves that an incremental strategy does not inherently worsen the final solution. In fact, it can improve it.

The superior efficiency of expansion strategies can be interpreted through curriculum learning. The GNN is able to learn more fundamental patterns and node relationships by beginning the training process on small subgraphs and gradually increasing the complexity of the problem. This warm start helps the model converge faster. For example, on queen6_6.col, `bfs_degree` reduced the mean runtime from 1167 seconds to 378 seconds. This improvement is statistically significant ($p < 0.001$), (see Table 4.24) and demonstrates the practical value of the approach. By prioritizing high-degree nodes, the model appears to learn crucial constraints in the early stages, which accelerates convergence without losing its ability to find a high-quality global solution. This combination of not worsening and sometimes improving solution quality while substantially decreasing computational cost makes `bfs_degree` a highly effective and practical strategy.

The role of hyperparameter tuning was also examined. The tables of the selected hyperparameters (see Tables 4.3—4.10) show that the optimal settings were similar across the strategies for some hyperparameters. For instance, nearly all tuning runs favored GraphSAGE layers, often with a mean layer aggregation. The treatment of noise-related hyperparameters showed a clear pattern between the baseline and expansion strategies. `bfs_layer_by_layer` follows the baseline, and the tuner selected use_noise = `True` for the majority of instances, typically paired with a decreasing schedule. In contrast, for `bfs_degree` and `random_walk` strategies, winning configurations almost always had use_noise = `False`. This pattern supports the intuitive argument that subgraph expansion itself acts as an implicit source of noise. In every expansion step, a fresh batch of randomly initialized node embeddings is added to the model embedding for new vertices. In contrast, `bfs_layer_by_layer` variant is an exception because it introduces the entire layer of nodes at once. The two warm-start hyperparameters load_embedding (re-use of a previously trained embedding matrix) and load_weights (initializing the GNN with saved model parameters) showed a clear pattern. In `bfs_degree` and `random_walk` expansions, the tuner frequently chose to enable both switches (Tables 4.4, 4.5). As a result, the parameters learned on the current subgraph are a strong prior to the next subgraph. In `bfs_layer_by_layer` expansion, load_embedding remained true for most graphs, but load_weights was omitted in more than half of the selected trials (Table 4.3). The likely reason is that each step injects an entire layer of vertices and dramatically alters the graph structure. Embedding vectors can still be reused (they act as a memory for already seen nodes), but the higher-level GNN weights optimized for the previous, much smaller subgraph may no longer generalize and could even constrain the optimizer in a sub-optimal region.

5.2 Limitations

Although this thesis presents promising results, it is important to acknowledge its limitations to provide context for the findings. First, the experiments were carried out on a specific set of graph families (Mycielski, Queen, and one social network graph). The performance of the proposed methods may differ on graphs with different topological properties. Consequently, the conclusions may not be universally generalizable to all types of vertex-coloring instances. Second, the study was limited to two GNN architectures, Graph Convolution Networks and GraphSAGE. Although these are foundational models, more advanced architectures like Graph Attention Networks (GATs) or Graph Isomorphism Networks (GINs) might interact differently with the expansion strategies. Third, the subgraph expansion strategies themselves are based on relatively simple heuristics (BFS, degree, random walks). More sophisticated strategies for expanding the graph could potentially yield better results. Fourth, the performance of the proposed methods is sensitive to the choice of hyperparameters, which were selected via an automated tuning process for each problem instance and strategy. The optimal configuration is not universal, and different choices can lead to different outcomes. For example, for the `bfs_degree` strategy on the queen7_7 graph, the selected configuration utilized an lstm aggregator (Table 4.4). This more complex aggregator corresponded to a degradation in performance compared to the baseline, both in solution quality and runtime (Tables 4.20 and 4.23). This highlights that the interaction between the expansion strategy and model architecture choices is complex and highly problem-dependent.

5.3 Future Work

The limitations of this study naturally give rise to several promising directions for future research. The first step is to evaluate the generalization of the proposed method by testing on a more diverse and comprehensive set of benchmark instances. Second, future work should investigate the interplay between subgraph expansion and more expressive GNN architectures. Third, new expansion strategies should be explored, because the three expansion strategies examined here represent only a narrow slice of the design space. Fourth, one can try to add graph features such as degree or

centrality measures to node embeddings before the message passing mechanism figures out. Lastly, in this thesis only the model weights and node embeddings were transferred from one subgraph to the next. A natural incremental learning regularizer would be to carry forward the actual color assignments. By reusing the colors already found, the GNN starts each expansion near a valid solution, might avoid forgetting previous progress, and can concentrate on fixing conflicts from the newly added nodes.

5.4 Conclusion

This thesis investigated the efficacy of using subgraph expansion strategies as a form of curriculum learning for training Graph Neural Networks to solve the vertex coloring problem. The central research question was whether constructing a curriculum by iteratively growing subgraphs of the original during training could improve computational efficiency and solution quality. Through a series of computational experiments on benchmark graph instances, it demonstrated that the proposed expansion strategy based on degree-based breadth-first search leads to substantial reductions in training time and faster convergence compared to the standard approach of training on the `full_graph`. Importantly, this time savings comes at no cost in accuracy. In most of the tested benchmark instances, `bfs_degree` achieved solution qualities that were statistically indistinguishable from, and in several cases slightly better than, those obtained by training on the `full_graph`. In conclusion, the proposed subgraph expansion framework offers a practical and effective way to make GNN-based solvers for graph coloring more efficient, therefore broadening their applicability to real world problems where both solution quality and speed are important.

BIBLIOGRAPHY

Appel, K. & Haken, W. (1977). Every planar map is four colorable. Part I: Discharging. *Illinois Journal of Mathematics*, *21*(3), 429 – 490.

Ben-Ameur, W., Mahjoub, A. R., & Neto, J. (2014). *The Maximum Cut Problem*, chapter 6, (pp. 131–172). John Wiley & Sons, Ltd.

Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, (pp. 41–48)., New York, NY, USA. Association for Computing Machinery.

Boman, E. G., Bozdağ, D., Catalyurek, U., Gebremedhin, A. H., & Manne, F. (2005). A scalable parallel graph coloring algorithm for distributed memory computers. In *Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference, Lisbon, Portugal, August 30-September 2, 2005. Proceedings 11*, (pp. 241–251). Springer.

Bondy, J. & Murty, U. (2008). *Graph Theory* (1st ed.). Springer Publishing Company, Incorporated.

Bozóki, S., Gál, P., Marosi, I., & Weakley, W. D. (2019). Domination of the rectangular queen's graph.

Brand, T., Faber, D., Held, S., & Mutzel, P. (2025). A customized sat-based solver for graph coloring.

Brélaz, D. (1979). New methods to color the vertices of a graph. *Commun. ACM*, *22*(4), 251–256.

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., & Vandergheynst, P. (2017). Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, *34*(4), 18–42.

Bruna, J., Zaremba, W., Szlam, A., & LeCun, Y. (2014). Spectral networks and locally connected networks on graphs.

Carter, M. W. (1986). Or practice—a survey of practical applications of examination timetabling algorithms. *Operations research*, *34*(2), 193–202.

Chaitin, G. J. (1982). Register allocation & spilling via graph coloring. *ACM Sigplan Notices*, *17*(6), 98–101.

Colantonio, L., Cacioppo, A., Scarpati, F., & Giagu, S. (2024). Efficient graph coloring with neural networks: A physics-inspired approach for large graphs.

Costa, D. & Hertz, A. (1997). Ants can colour graphs. *The Journal of the Operational Research Society*, *48*(3), 295–305.

Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., & Song, L. (2018). Learning combinatorial optimization algorithms over graphs.

de Lima, A. M. & Carmo, R. (2018). Exact algorithms for the graph coloring problem. *Revista de Informática Teórica e Aplicada*, *25*(4), 57–73.

Dorigo, M. & Di Caro, G. (1999). Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 2, (pp. 1470–1477 Vol. 2).

Duong, C. T., Hoang, T. D., Dang, H. T. H., Nguyen, Q. V. H., & Aberer, K. (2019). On node features for graph neural networks.

Elman, J. L. (1993). Learning and development in neural networks: the importance

of starting small. *Cognition*, *48*(1), 71–99.

Formanowicz, P. & Tanaś, K. (2012). A survey of graph coloring - its types, methods and applications. *Foundations of Computing and Decision Sciences*, *37*(3), 223–238.

Galinier, P. & Hao, J.-K. (1999). Hybrid Evolutionary Algorithms for Graph Coloring. *Journal of Combinatorial Optimization*, *3*(4), 379–397.

Ganguli, R. & Roy, S. (2017). A study on course timetable scheduling using graph coloring approach. *international journal of computational and applied mathematics*, *12*(2), 469–485.

Garey, M. R. & Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural message passing for quantum chemistry.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, *13*(5), 533–549. Applications of Integer Programming.

Glover, F. W. (1990). Tabu search: A tutorial. *Interfaces*, *20*, 74–94.

Goemans, M. X. & Williamson, D. P. (1995). Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, *42*(6), 1115–1145.

Gori, M., Monfardini, G., & Scarselli, F. (2005). A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, (pp. 729–734 vol. 2).

Gusev, V. V. (2020). The vertex cover game: Application to transport networks. *Omega*, *97*, 102102.

Hale, W. K. (1980). Frequency assignment: Theory and applications. *Proceedings of the IEEE*, *68*(12), 1497–1514.

Hamilton, W., Ying, Z., & Leskovec, J. (2017). Inductive representation learning on large graphs. *Advances in neural information processing systems*, *30*.

Hansen, J., Kubale, M., Kuszner, Ł., & Nadolski, A. (2004). Distributed largest-first algorithm for graph coloring. In Danelutto, M., Vanneschi, M., & Laforenza, D. (Eds.), *Euro-Par 2004 Parallel Processing*, (pp. 804–811)., Berlin, Heidelberg. Springer Berlin Heidelberg.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press.

Ijaz, A. Z., Ali, R. H., Ali, N., Laique, T., & Ali Khan, T. (2022). Solving graph coloring problem via graph neural network (gnn). In *2022 17th International Conference on Emerging Technologies (ICET)*, (pp. 178–183).

Jang, E., Gu, S., & Poole, B. (2017). Categorical reparameterization with gumbel-softmax.

Jin, Y., Yan, X., Liu, S., & Wang, X. (2024). A unified framework for combinatorial optimization based on graph neural networks.

Joshi, C. K., Laurent, T., & Bresson, X. (2019). An efficient graph convolutional network technique for the travelling salesman problem.

Junior Mele, U., Maria Gambardella, L., & Montemanni, R. (2021). Machine learning approaches for the traveling salesman problem: A survey. In *Proceedings of the 2021 8th International Conference on Industrial Engineering and Appli-*

cations (Europe), ICIEA 2021-Europe, (pp. 182–186)., New York, NY, USA. Association for Computing Machinery.

Khemani, B., Patil, S., Kotecha, K., & Tanwar, S. (2024). A review of graph neural networks: concepts, architectures, techniques, challenges, datasets, applications, and future directions. *Journal of Big Data*, *11*(1), 18.

Kim, S., Yun, S., & Kang, J. (2022). Dygrain: An incremental learning framework for dynamic graphs. In Raedt, L. D. (Ed.), *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, (pp. 3157–3163). International Joint Conferences on Artificial Intelligence Organization. Main Track.

Kipf, T. N. & Welling, M. (2017). Semi-supervised classification with graph convolutional networks.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671–680.

Knuth, D. (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press Series. ACM Press.

Kool, W., van Hoof, H., & Welling, M. (2019). Attention, learn to solve routing problems!

Laporte, G. (2010). A concise guide to the traveling salesman problem. *The Journal of the Operational Research Society*, *61*(1), 35–40.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436–444.

Lemos, H., Prates, M., Avelar, P., & Lamb, L. (2019). Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems.

Lewis, R. & Thompson, J. (2011). Thompson, j.: On the application of graph colouring techniques in round-robin sports scheduling. computers & operations research 38(1), 190-204. *Computers & Operations Research*, *38*, 190–204.

Lewis, R. M. R. (2016). *A Guide to Graph Colouring: Algorithms and Applications*. Cham, Switzerland: Springer.

Li, W., Li, R., Ma, Y., Chan, S. O., Pan, D., & Yu, B. (2022). Rethinking graph neural networks for the graph coloring problem.

Li, Z., Chen, Q., & Koltun, V. (2018). Combinatorial optimization with graph convolutional networks and guided tree search.

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., & Stoica, I. (2018). Tune: A research platform for distributed model selection and training.

Liu, C., Zhan, Y., Wu, J., Li, C., Du, B., Hu, W., Liu, T., & Tao, D. (2023). Graph pooling for graph neural networks: Progress, challenges, and opportunities.

Malaguti, E., Monaci, M., & Toth, P. (2011). An exact approach for the vertex coloring problem. *Discrete Optimization*, *8*(2), 174–190.

Matula, D. W. & Beck, L. L. (1983). Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, *30*(3), 417–427.

Mazyavkina, N., Sviridov, S., Ivanov, S., & Burnaev, E. (2021). Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, *134*, 105400.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, *518*(7540), 529–533.

Mostafaie, T., Modarres Khiyabani, F., & Navimipour, N. J. (2020). A systematic study on meta-heuristic approaches for solving the graph coloring problem. *Computers & Operations Research*, *120*, 104850.

Mycielski, J. (1955). Sur le coloriage des graphs. *Colloquium Mathematicae*, *3*(2), 161–162.

Méndez-Díaz, I. & Zabala, P. (2006). A branch-and-cut algorithm for graph coloring. *Discrete Applied Mathematics*, *154*(5), 826–847. IV ALIO/EURO Workshop on Applied Combinatorial Optimization.

Newman, M. E. J. (2003). The structure and function of complex networks. *SIAM Review*, *45*(2), 167–256.

Palubeckis, G. (2008). On the recursive largest first algorithm for graph colouring. *International Journal of Computer Mathematics*, *85*(2), 191–200.

Papadimitriou, C. & Steiglitz, K. (1998). *Combinatorial Optimization: Algorithms and Complexity*. Dover Books on Computer Science. Dover Publications.

Pugacheva, D., Ermakov, A., Lyskov, I., Makarov, I., & Zotov, Y. (2024). Enhancing gnns performance on combinatorial optimization by recurrent feature update.

Resende, M. G. C. (2003). Combinatorial optimization in telecommunications. In P. M. Pardalos & V. Korotkikh (Eds.), *Optimization and Industry: New Frontiers* (pp. 59–112). Boston, MA: Springer US.

Sachdeva, K. (2012). Applications of graphs in real-life. *International Journal of Advanced Research in Computer Science*, *3*, 403–406.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, *20*(1), 61–80.

Schuetz, M. J. A., Brubaker, J. K., & Katzgraber, H. G. (2022). Combinatorial optimization with physics-inspired graph neural networks. *Nature Machine Intelligence*, *4*(4), 367–377.

Schuetz, M. J. A., Brubaker, J. K., Zhu, Z., & Katzgraber, H. G. (2022). Graph coloring with physics-inspired graph neural networks. *Phys. Rev. Res.*, *4*, 043131.

Su, J., Zou, D., Zhang, Z., & Wu, C. (2024). Towards robust graph incremental learning on evolving graphs.

Trick, M. (2002). COLOR02/03/04: Graph Coloring and its Generalizations. Accessed: 2025-02-15.

u/DarkerThanAzure (2023). Current US congressional districts colored to uphold the four color theorem. `https://i.redd.it/u487ejw2vn2b1.png`. Accessed: 2025-03-01.

Vakil, N. & Amiri, H. (2023). Curriculum learning for graph neural networks: A multiview competence-based approach.

van de Ven, G. M., Tuytelaars, T., & Tolias, A. S. (2022). Three types of incremental learning. *Nature Machine Intelligence*, *4*, 1185–1197.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2018). Graph attention networks.

Vogiatzis, C. & Pardalos, P. M. (2013). Combinatorial optimization in transportation and logistics networks. In P. M. Pardalos, D.-Z. Du, & R. L. Graham (Eds.), *Handbook of Combinatorial Optimization* (pp. 673–722). New York, NY: Springer New York.

Wang, H. & Li, P. (2023). Unsupervised learning for combinatorial optimization

needs meta-learning.

Wang, X., Yan, X., & Jin, Y. (2023). A graph neural network with negative message passing for graph coloring.

Watkins, G., Montana, G., & Branke, J. (2023). Generating a graph colouring heuristic with deep q-learning and graph neural networks.

Welsh, D. J. A. & Powell, M. B. (1967). An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, *10*(1), 85–86.

Wolsey, L. & Nemhauser, G. (2014). *Integer and Combinatorial Optimization*. Wiley Series in Discrete Mathematics and Optimization. Wiley.

Wu, Q. & Hao, J.-K. (2015). A review on algorithms for maximum clique problems. *European Journal of Operational Research*, *242*(3), 693–709.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2021). A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, *32*(1), 4–24.

Xu, Y., Zhang, Y., Guo, W., Guo, H., Tang, R., & Coates, M. (2020). Graphsail: Graph structure aware incremental learning for recommender systems. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, CIKM '20, (pp. 2861–2868)., New York, NY, USA. Association for Computing Machinery.

Yang, Y. & Whinston, A. (2023). A survey on reinforcement learning for combinatorial optimization. In *2023 IEEE World Conference on Applied Intelligence and Computing (AIC)*, (pp. 131–136).

Yolcu, E., Wu, X., & Heule, M. J. H. (2020). Mycielski graphs and pr proofs. In Pulina, L. & Seidl, M. (Eds.), *Theory and Applications of Satisfiability Testing – SAT 2020*, (pp. 201–217)., Cham. Springer International Publishing.

Yuan, Q., Guan, S.-U., Ni, P., Luo, T., Man, K. L., Wong, P., & Chang, V. (2023). Continual graph learning: A survey.

Zhang, X., Song, D., & Tao, D. (2022). Cglb: Benchmark tasks for continual graph learning. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., & Oh, A. (Eds.), *Advances in Neural Information Processing Systems*, volume 35, (pp. 13006–13021). Curran Associates, Inc.

Zhang, Y., Zhang, K., & Wu, N. (2024). Using graph isomorphism network to solve graph coloring problem. In *2024 4th International Conference on Consumer Electronics and Computer Engineering (ICCECE)*, (pp. 209–215).

Zhou, F. & Cao, C. (2021). Overcoming catastrophic forgetting in graph neural networks with experience replay.

Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., & Sun, M. (2020). Graph neural networks: A review of methods and applications. *AI Open*, *1*, 57–81.

Zhou, Y., Hao, J.-K., & Duval, B. (2016). Reinforcement learning based local search for grouping problems: A case study on graph coloring. *Expert Systems with Applications*, *64*, 412–422.