VColRL: Learn to solve the Vertex Coloring Problem using Reinforcement Learning

Anonymous authors
Paper under double-blind review

Abstract

We propose VColRL, a deep reinforcement learning framework for solving the Vertex Coloring Problem (VCP), which aims to color the vertices of a graph using the minimum number of colors such that no two adjacent vertices share the same color. VColRL is based on a novel Markov Decision Process (MDP) formulation, identified through a systematic evaluation of multiple configurations. It employs a reduction-based neural architecture and a reward mechanism designed to minimize the highest-numbered color used from an ordered set. Experiments on synthetic and benchmark graphs show that VColRL consistently outperforms greedy and learning-based methods in terms of color usage, while achieving competitive performance with advanced optimization solvers and search-based baselines. In addition to delivering high-quality solutions, VColRL achieves significantly faster runtimes than the baselines, demonstrating strong scalability and generalization across diverse graphs.

1 Introducion

The Vertex Coloring Problem (VCP, (Birkhoff, 1912)) involves assigning colors to the vertices of a graph using the minimum number of colors, such that no two adjacent vertices share the same color. VCP is a classic NP-hard problem with diverse real-world applications, including *frequency assignment in wireless networks*, *register allocation in compilers*, *exam timetabling*, *sports scheduling*, *aircraft scheduling*, *layout segmentation*, and *map coloring* (Ahmed, 2012). Two domains where efficient solutions are especially critical are wireless communications and compiler design. In wireless networks, VCP models the channel allocation task where transmitters within interference range must be assigned different frequencies to avoid signal collisions (Hale, 1980). This is vital in dynamic environments such as Mobile and Vehicular Ad hoc Networks (Hoebeke et al., 2004; Al-Sultan et al., 2014), where rapid reallocation is needed to adapt to changing topologies. In compilers, VCP underpins register allocation, where variables with overlapping lifetimes must be assigned distinct hardware registers (Chaitin et al., 1981). Efficient solving is crucial in just-in-time (JIT) compilation systems like Numba (Lam et al., 2015), where delays can degrade runtime performance. The solution to the VCP is a classic example of combinatorial optimization, a subfield of mathematical optimization concerned with selecting the best solution from a finite, often vast, set of discrete possibilities. Other examples of combinatorial optimization problems include the traveling salesman problem (TSP (Voigt, 1831)), the maximum independent set problem (MIS (Miller & Muller, 1960)). However, most combinatorial optimization problems are NP-hard, making finding exact solutions in many real-world scenarios impractical.

Over the years, researchers have developed efficient heuristics (e.g. (Knuth, 1997)) to address the combinatorial problems. However, these heuristics are often problem-specific and require effective application of domain knowledge. In addition to heuristic approaches, researchers have developed advanced optimization solvers such as the Gurobi Optimization Studio (Gurobi Optimization, 2024), and IBM ILOG CPLEX Optimization Studio (IBM, 2024), which excel at finding exact solutions for complex problems. These solvers utilize sophisticated mathematical techniques and algorithms, such as branch-and-bound (Morrison et al., 2016) and cutting-plane methods (Lee et al., 2015), (Goffin & Vial, 2002), to handle large-scale integer and mixed-integer programming problems with high precision. Despite their powerful capabilities, the compute

and time required for these solvers to find optimal solutions increase substantially as the problem size grows (Luppold et al., 2018).

An alternative approach to tackling complex optimization problems is through data-driven methods. These methods provide a simpler structure and do not require extensive domain expertise. By leveraging Machine Learning, data-driven approaches focus on learning from data and performing straightforward computations, such as matrix multiplications (Burkov, 2019). This simplicity often results in significant time saving, as these techniques can process large datasets and yield solutions more rapidly compared to traditional solvers for bigger problem instances. Over the past decades, the rise of neural networks and advanced ML methods has further enhanced these data-driven techniques, making them more powerful and versatile.

While supervised learning offers one path, obtaining labeled datasets can be challenging. As a result, alternative methods are needed. Reinforcement learning presents a promising option as it does not rely on labeled data. Instead, it frames the problem as a Markov Decision Process (MDP), allowing the system to learn optimal strategies through iterative interactions with the environment and feedback (Sutton & Barto, 2018), (Ernst & Louette, 2024).

Deep reinforcement learning (DRL), a subset of reinforcement learning, has gained prominence due to its ability to handle high-dimensional state and action spaces using deep neural networks. DRL combines reinforcement learning with deep learning to model complex environments and discover intricate patterns that are difficult to capture with traditional methods (Lapan, 2020). By leveraging deep neural networks, DRL can learn sophisticated policies and make decisions based on large and complex datasets, further enhancing its effectiveness in solving complex optimization problems. This approach has proven particularly useful in areas such as robotics (Morales et al., 2021), autonomous systems (Kiran et al., 2021), and game playing (Dong et al., 2020), where it can achieve remarkable performance by learning from interactions with the environment.

For graph-related problems, Graph Neural Networks (GNNs (Scarselli et al., 2008)) provide a significant advancement. GNNs are tailored to work with graph-structured data, capturing vertices' intricate relationships and dependencies. They utilize a message-passing approach to collect information from neighboring vertices and update the representation of each vertex accordingly. This method allows GNNs to effectively model complex graph structures. Such capabilities are particularly beneficial for vertex coloring problems, where a deep understanding of graph topology is crucial for achieving optimal color assignments. Furthermore, integrating GNNs with DRL can enhance their effectiveness by combining learned graph representations with reinforcement learning strategies, thereby improving performance.

Our Contribution: This paper introduces VColRL, a novel reinforcement learning framework for addressing the Vertex Coloring Problem (VCP). VColRL uses the Proximal Policy Optimization (PPO) algorithm (Schulman et al., 2017) with the GraphSAGE architecture (Hamilton et al., 2017). A key contribution of VColRL is its innovative reward strategy, which treats the color set as ordered and penalizes non-contiguous assignments. This design ensures unique and efficient color usage by focusing on minimizing the highest-numbered color used, rather than the total number of colors. This approach enhances learning stability and reduces ambiguities in the solution space. In addition, we perform a comprehensive analysis of rollback strategies, reward mechanisms, and action models defined in Section 3.1 to identify the best-performing Markov Decision Process (MDP) configuration for the VCP. Through extensive testing on synthetic and benchmark graphs, we show that VColRL demonstrates competitive performance against state-of-the-art optimization solvers and other baselines while being significantly faster. It also generalizes well across diverse graph types and scales efficiently to larger graphs.

2 Related Work

The vertex coloring problem is a well-known NP-hard combinatorial optimization problem (Garey & Johnson, 1976). Exact solvers for this problem require exploring an exponentially large solution space to guarantee optimality, which becomes computationally infeasible as the graph size grows (de Lima & Carmo, 2018). Solutions to the vertex coloring problem can be categorized into two main approaches: conventional methods and machine learning-based approaches.

Conventional Methods: This category includes approximation algorithms such as DSatur, Welsh-Powell (Aslan & Baykan, 2016), as well as mathematical solvers like Gurobi and CPLEX, which provide exact solutions but at a high computational cost, especially for large graphs. Additionally, metaheuristic approaches such as Simulated Annealing, Genetic Algorithms, and Tabu Search are commonly used for solving the VCP (Mostafaie et al., 2020), (Dokeroglu & Sevinc, 2021). They are capable of generating high-quality and near-optimal results.

Machine Learning-Based Approaches: Several studies have used supervised learning for solving the vertex coloring problem. Das et al. (2019) have introduced a supervised learning approach where a deep learning model, using Long Short-Term Memory layers followed by a correction phase, is trained on optimal coloring of graphs to handle invalid color assignments. Over time, researchers have recognized the need for graph-specific architectures to better address the complexities inherent in vertex coloring problems. Lemos et al. (2019) has employed Graph Neural Networks (GNN) (Scarselli et al., 2008) to predict the chromatic number of graphs, leveraging GNNs' ability to capture graph structures. Similarly, Ijaz et al. (2022) has utilized GNNs to solve the vertex coloring problem, focusing on efficiently finding the chromatic number of large graphs.

However, as graph sizes increase, obtaining ground truth becomes impractical, leading to the emergence of reinforcement learning as a promising solution. Huang et al. (2019) has adapted AlphaGo Zero (Silver et al., 2017) with graph embeddings, introducing a novel deep neural network architecture known as FastColorNet for the vertex coloring problem. Cummins & Veras (2024) have highlighted the potential of RL to tackle the vertex coloring problem but also pointed out its limitations without label-invariant representations. They have emphasized the importance of integrating GNNs to enhance RL performance by providing essential structural insights. Gianinazzi et al. (2021) have proposed a greedy combined probabilistic heuristic for the vertex coloring problem that integrates reinforcement learning and an attention mechanism (Vaswani, 2017) for vertex selection. In their framework, they only used a terminal step reward based solely on color count. Similarly, Watkins et al. (2023) introduced ReLCol, a method that combines Q-learning with GNN for feature extraction.

The most closely related work is by Ahn et al. (2020), which proposes the *deferral* action strategy for solving the Maximum Independent Set (MIS) problem using reinforcement learning. In MIS, each vertex requires a binary decision of whether to include it in the independent set or not, resulting in only two possible actions per vertex. In contrast, the VCP involves selecting a specific color for each vertex, leading to a much larger action space. After evaluating various MDP configurations, we find that the *deferral* action strategy is also effective for the VCP, providing flexibility to defer decisions for certain vertices. Consequently, we incorporate this strategy into VColRL. Although both frameworks utilize GraphSAGE architecture, it is not central to VColRL's approach. GraphSAGE is used as one of many possible architectures for capturing graph structure and can be replaced without affecting VColRL's core contributions, which lie in designing a unique reward strategy for the MDP. We use GraphSAGE primarily to scale VColRL to larger graphs, as its ability to sample subgraphs of uncolored vertices for processing in MDP steps significantly improves the framework's speed. The Vertex Coloring Problem (VCP) poses several unique challenges for reinforcement learning compared to problems like the Maximum Independent Set (MIS), such as a larger action space, increased modeling complexity, the existence of multiple equivalent solutions from a graph partitioning perspective, and strong dependencies between the partially colored and uncolored portions of the graph. A detailed discussion of these challenges and how we address them is provided in Appendix C.1.

3 Framework for Vertex Coloring Problem

We now describe our framework for the VCP. Given a graph $G = (V, E)$ with vertex set V and edge set E, the objective is to assign colors to the vertices of a graph with the minimum number of colors, so that no two adjacent vertices share the same color. In our approach to the VCP, we attempt to color the vertices using a multiclass vertex classification strategy with 15 color classes, selecting colors from the ordered set $C = \{1, 2, \ldots, 15\}$ for each vertex. If some vertices remain uncolored due to timestep completion or color insufficiency, they are addressed as described in Section 3.3. The resulting solution can be represented as a

vector $x = [x_i : i \in V] \in (\{*\} \cup C)^V$, where each element x_i either indicates the color assigned to vertex i from the set C or $x_i = *$ denotes that vertex i has not been assigned any color.

3.1 Markov Decision Process for the VCP

We model the VCP as a finite MDP, which terminates when either all vertices are colored or the time limit (episode length) is reached.

The key components of the MDP are:

States: States represent the current configuration of the system. A state is represented by a vertex-state vector $s = [s_i : i \in V] \in (\{*\} \cup C)^{|V|}$, where $s_i \in C$ denotes the color assigned to vertex i, and $s_i = *$ indicates that the vertex is undecided, meaning it is yet to be colored. Initially, all vertices are undecided ($s_i = *$ for all $i \in V$). The process ends when no undecided vertices remain or the time limit is reached.

Actions: Actions represent the decisions the agent makes for the undecided vertices. Let V_* denote the set of undecided vertices. We consider two models for action:

- *Model with Deferral:* In this model, the agent can either defer the decision for an undecided vertex or assign it a color. This is represented by $a_* = [a_i : i \in V_*] \in (\{*\} \cup C)^{|V_*|}$, allowing the agent to delay coloring certain vertices and focus on smaller parts of the graph first.
- *Model without Deferral:* In this model, the agent must assign a color to each undecided vertex without the option to defer. This is represented by $a_* = [a_i : i \in V_*] \in C^{|V_*|}$.

Transitions: Transitions define how the system moves from one state to another after an action is taken. The transition from state $s \to s'$ for action a_* occurs in two phases: the update phase and the clean-up phase.

Update Phase: The action a_*, determined by the policy for the undecided vertices V_*, is applied to create an intermediate state $\hat{s}$. Specifically, $\hat{s}_i = a_i$ if $i \in V_*$, and $\hat{s}_i = s_i$ otherwise.

Clean-up Phase: The clean-up phase ensures the resulting state s' is conflict-free. For each pair of conflicting vertices (i.e., vertices assigned the same color), we consider two rollback models:

- *Hard Rollback Model:* Both conflicting vertices are reset to the undecided state, providing greater flexibility to revisit and resolve conflicts, though this may require more steps.
- *Soft Rollback Model:* Only the vertices involved in the latest action are reset to the undecided state, leaving previous assignments unchanged. This approach resolves conflicts more quickly but offers fewer opportunities to adjust earlier assignments.

Rewards: The immediate reward for a transition $s \to s'$ is a weighted combination of two terms, namely the vertex satisfaction reward and the color usage penalty. Vertex satisfaction reward, representing the increase in the number of assigned vertices, is measured as $Sat(s') - Sat(s)$, where $Sat(s) = \sum_{i \in V} \mathbb{I}(s_i \in C)$, and $\mathbb{I}$ is the indicator function.

For the color usage penalty, we consider two models, both of which serve the same purpose of minimizing color usage but represent the penalty in different ways:

- *Max-color strategy:* Penalizes based on the highest-numbered color assigned from C, defined as:

$$UB(s) = \begin{cases} 0, & \text{if } s_i = *, \forall i \in V, \\ \max\{s_i \mid s_i \in C; \forall i \in V\}, & \text{otherwise.} \end{cases}$$

- *Color-count strategy:* Penalizes based on the total number of distinct colors used, defined as

$$Count(s) = \begin{cases} 0, & \text{if } s_i = *, \forall i \in V, \\ |\{s_i \mid s_i \in C; \forall i \in V\}|, & \text{otherwise.} \end{cases}$$

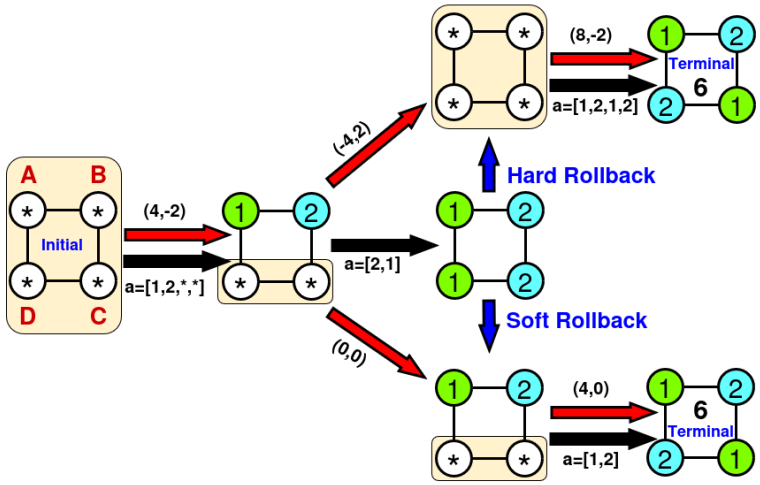


Figure 1: ***Illustration of the MDP for the VCP.*** *The red arrow represents the final transition, the black arrow represents the update phase, and the blue arrow represents the cleanup phase. The sum of elements in the tuples (vertex satisfaction reward, color usage penalty) denotes the immediate reward for a transition, while the numerical value in the terminal state represents the total returns, calculated by summing the rewards of all transitions in the episode. The vertices highlighted in yellow indicate the subgraphs where actions are taken, with the action vector a ordered by vertices A, B, C, and D, maintaining the same vertex order within the subgraphs.*

For a transition $s \to s'$, the immediate reward using the *max-color* strategy is given by $r_{s\to s'} = w_1 \cdot [Sat(s') - Sat(s)] + w_2 \cdot [UB(s) - UB(s')]$, whereas for the *color-count* strategy, the reward is calculated similarly, with the UB function replaced by the $Count$ function.

Since our objective is to ensure that all vertices are colored, we should prioritize increasing the vertex satisfaction reward over reducing the color usage penalty. This can be achieved by ensuring that the weights satisfy $w_1 > w_2$. This choice is crucial because, for example, if w_1 and w_2 are equal, a situation could arise where coloring one more vertex by using one new color results in a zero reward, effectively making it equivalent to taking no action at all. We use the values 2 and 1 for w_1 and w_2, respectively.

Illustrative example of VColRL's underlying MDP: Figure 1 illustrates the vertex coloring process for a graph with four vertices labeled A, B, C, and D. The initial state vector is $[*, *, *, *]$, indicating that no vertices have been assigned colors yet. An action $[1, 2, *, *]$ is taken, assigning colors 1 and 2 to vertices A and B, respectively, resulting in the next state $[1, 2, *, *]$. This transition yields a vertex satisfaction reward of $w_1 \times 2 = 4$, where $w_1 = 2$ is the weight for vertex satisfaction, and 2 vertices are satisfied. The color usage penalty is $w_2 \times -2 = -2$, where $w_2 = 1$ is the weight for color usage, and 2 colors are used. This results in the tuple $(4, -2)$, where 4 corresponds to the weighted vertex satisfaction reward, and -2 corresponds to the weighted color usage penalty. The total transition reward for this step is obtained by summing these values, yielding a reward of 2. Subsequently, action $[2, 1]$ is applied to the undecided vertices C and D, resulting in an intermediate state of $[1, 2, 2, 1]$. This configuration leads to a conflict, requiring the application of rollback strategies during the clean-up phase. Depending on the specific rollback strategy used, the transition will proceed as follows:

In the *hard rollback* strategy, all vertices are reverted to the undecided state, resulting in a state vector $[*, *, *, *]$ and a final tuple of $(-4, 2)$. The reward and penalty are calculated based on reverting all assignments. The action $[1, 2, 1, 2]$ is then applied, leading to the final state $[1, 2, 1, 2]$ with a reward tuple of $(8, -2)$, giving a total episode return (i.e., the cumulative reward in that episode) of 6.

In contrast, the *soft rollback* strategy only reverts the newly assigned vertices, resulting in the state $[1, 2, *, *]$ and a tuple of $(0, 0)$, indicating no additional reward or penalty from the reverted assignments. The action $[1, 2]$ is then applied to the remaining undecided vertices C and D, resulting in the final state $[1, 2, 1, 2]$ with a reward tuple of $(4, 0)$, giving a total return of 6 for the episode.

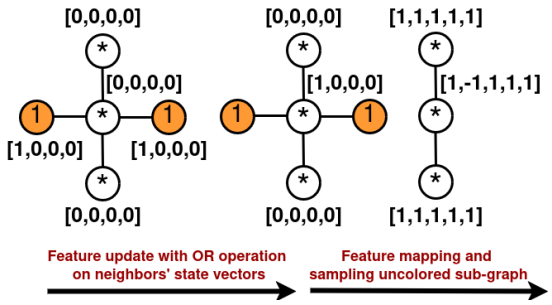


Figure 2: Feature extraction and subgraph sampling.

3.2 Model Training and Hyperparameters

Proximal Policy Optimization algorithm: We use PPO (Schulman et al., 2017) to train the agent to solve the VCP problem. The objective for the actor is expressed as:

$$\mathcal{L}_{\text{actor}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right]$$

where the subscript t denotes the time within the episode (trajectory), $r_t(\theta) = \frac{\pi(a_t|s_t;\theta_{\text{new}})}{\pi(a_t|s_t;\theta_{\text{old}})}$ is the probability ratio of the new policy to the old policy at time t, ϵ is a hyperparameter that defines the clipping range to prevent large policy updates, $\hat{A}_t$ is the advantage which represents the difference between the actual return and the estimated value of the state at time t. In addition, PPO also has a critic loss, and we add entropy regularization to the total objective to encourage exploration. For a detailed explanation of PPO equipped with entropy regularization, refer to Appendix A.3.

Model architecture and input feature: We use the GraphSAGE architecture (Hamilton et al., 2017) for the policy and value networks. This architecture assigns colors to the undecided vertices of the graph by considering their subgraph, and therefore, we design a vertex feature that effectively conveys all the necessary information to the subgraph, enabling it to identify which color assignments could lead to conflicts. Consequently, our vertex feature vector is of the length of $1 + |C|$, where $|C|$ represents the number of colors in the set C. The first element of the feature vector is set to 1 to reflect the vertex's degree after feature aggregation in the GNN (see Appendix A.2). The remaining $|C|$ elements represent the color usage status of the vertex's neighbors: a value of 1 indicates that none of the neighbors use the corresponding color, while -1 indicates that one or more neighbors use the color. To get the common vector representing colors used by the neighbors, we convert each of the neighbor's states into a one-hot vector, with 1 at the position corresponding to the color it uses, and then perform OR operation on the neighbors' one-hot state vectors. Figure 2 shows the feature extraction and subgraph sampling process of VColRL with $|C| = 4$. After sampling the subgraph, we do not perform random sampling within the new neighborhood during message passing, as it may lead to a loss of crucial neighborhood information. For details about the VColRL architecture, GraphSAGE architecture, and PPO implementation, refer to Appendix A.1, Appendix A.2, and Appendix A.3.

Table 1: Hyperparameters used for training.

Hyperparameter	Value
Episode length/ Time limit	32
Replay buffer size	32
Batch size	32
Batch size for gradient step	16
Number of gradient steps per update	4
Critic loss coefficient	0.25
Entropy regularization coefficient	0.01
Learning rate of optimizer	0.0001
Gradient norm	1
Discount factor for PPO	1
Clip value for PPO	0.2

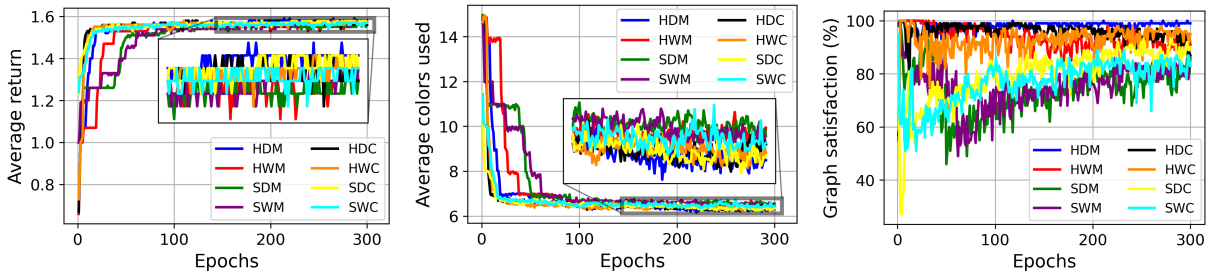
Training details and hyperparameters: We generate a graph dataset containing $15,000$ random Erdős–Rényi (ER (Erdos et al., 1960)) graphs with $50 - 100$ vertices and an edge probability of 0.15. This data set is divided into two parts: the first $14,000$ graphs are used for training, while the last 1000 graphs are used for validation. The dataset contains graphs with chromatic numbers ranging from 4 to 7. Both actor and critic networks consist of 4 hidden layers (number of messaging passing steps), each with a dimension of 128. ReLU is used as the activation function. We train the model for 300 epochs using the ADAM optimizer (Kingma, 2014). The model configurations and hyperparameters are selected empirically with the help of the Optuna (Akiba et al., 2019) framework, as described in Appendix B.1. During the experiments, we observe that all MDP variants converge to the best objective for similar values of hyperparameters. Table 1 presents the final hyperparameters.

3.3 Addressing Incomplete Solutions

The model is designed to output a color assignment for each vertex at each state, which requires learning a probability distribution over a fixed set of colors. We set this color set length to 15. VColRL begins multiclass vertex classification with an initial set of 15 colors. An incomplete solution may occur either when the time limit is reached before all vertices are colored, resulting in fewer than 15 colors being used, or when all 15 colors are utilized but some vertices remain uncolored. In either case, the subgraph of uncolored vertices is extracted, and the multiclass vertex classification is restarted fresh on the subgraph. If, in the previous iteration, k colors were used, the current iteration starts with color $k + 1$. This iterative process continues until all vertices are successfully colored. The final color count is obtained by summing the total number of colors used across all iterations.

4 Performance Evaluation of VColRL

This section first compares various MDP configurations to identify the best-performing MDP for training. Subsequently, we evaluate VColRL (trained using this optimal MDP) against several baselines on synthetic and benchmark graphs to assess its performance in solving the VCP. We use an NVIDIA GeForce RTX 4090 GPU for training models, and a 12th Gen Intel® Core™ i7-12700 processor featuring 20 logical cores with 32 GB RAM for testing models and baselines.



(a) Returns *(b) Colors used* *(c) Graph satisfaction*

Figure 3: ***Performance of VColRL with different MDP configurations.*** *The strategies used in our framework are represented by the following abbreviations:* ***H*** *denotes hard rollback,* ***S*** *denotes soft rollback,* ***D*** *denotes with deferral action,* ***W*** *denotes without deferral action,* ***M*** *denotes max-color reward strategy, and* ***C*** *denotes color-count reward strategy, e.g.,* ***HDM*** *denotes hard rollback with deferral action and max-color reward strategy.*

4.1 Comparison of various MDP Configurations

We analyze the validation results of models based on all the combinations of rollback strategies (*soft rollback* and *hard rollback*), action types (*with deferral* and *without deferral*), and reward strategies (*max-color* and *color-count*), resulting in a total of 8 configurations. The evaluation is conducted on 1000 validation graphs, enabling a thorough assessment to identify the optimal MDP configuration.

Table 2: Summary of Figure 3. ***Max Return*** *indicates the highest average return achieved across validation evaluations,* ***Min Color Used*** *denotes the minimum of the average colors used across validation evaluations, and* ***> 95% Graph Satisfaction*** *refers to the number of validation evaluations in which the model completely colors more than 95% of the graphs.*

Configuration	Max Return	Min Color Used	> 95% Graph Satisfaction
HDM	1.59	6.155	282
HWM	1.57	6.347	99
SDM	1.56	6.445	3
SWM	1.57	6.379	3
HDC	1.58	6.218	195
HWC	1.58	6.247	80
SDC	1.58	6.203	0
SWC	1.57	6.337	0

Figure 3a shows the return (sum of rewards) over epochs. Figure 3b illustrates the average number of colors used, reflecting how color usage evolves during training. Figure 3c presents the graph satisfaction percentage, indicating the proportion of validation graphs where all vertices are satisfied in one MDP trajectory. Together, these figures offer a comprehensive view of the agent's decision-making process, highlighting its impact on color efficiency and graph satisfaction. In addition, Table 2 summarizes the results of these figures.

Figure 3a demonstrates that returns increase for all MDP configurations, indicating that the agent effectively learns to fulfill its objective. Similarly, Figure 3b shows a consistent decrease in the number of colors used across epochs for all configurations, reflecting the agent's ability to minimize color usage. Among these, the HDM (*hard rollback with deferral action and max-color reward strategy*) emerges as the best-performing configuration, achieving higher returns and using fewer colors on average. This is particularly evident between epochs 150–300, where the blue line representing HDM stands out by being above others in returns and below others in average colors used. In Figure 3c, all *hard rollback* configurations initially start with lower graph satisfaction values, gradually increase to 100%, but differ in their post-convergence behavior. While HDM maintains full satisfaction, demonstrating stability, other configurations begin to deteriorate, resulting in instability. In contrast, the *soft rollback* configurations start with high graph satisfaction (near 100%), experience significant drops, and then oscillate before recovering to approximately 90% in 300 epochs, implying slower convergence and longer training times. Not maintaining graph satisfaction indicates that the agent fails to color all the vertices within the allotted time limit. Despite this, Figure 3a shows that returns remain relatively stable for such configurations, suggesting that few vertices per graph are left uncolored, which is undesirable. However, HDM consistently achieves full graph satisfaction along with minimized color usage, demonstrating its ability to color graphs more efficiently within less time. This is also evident from Table 2, where the HDM configuration achieves higher average returns, uses fewer colors, and maintains over 95% graph satisfaction in 282 out of 300 validation runs, which is better than all other configurations. To further support our claim, we run each variant five times and present the results in Appendix C.5. As shown in Figure 7 and Figure 8, the HDM configuration consistently yields more stable policies in terms of both graph satisfaction and color usage compared to the other variants. This result aligns with the objective of emphasizing vertex satisfaction, as discussed in Section 3.1.

HDM emerges as the best configuration due to several key factors. First, the *hard rollback* mechanism in HDM reverts both conflicting vertices to a deferred state, unlike *soft rollback*, which only reverts the newly assigned vertex. Since reinforcement learning relies on trial and error, *hard rollback* offers greater flexibility, enabling the model to explore better solutions more effectively. Additionally, the *with deferral* action strategy allows the agent to break the problem into simpler subproblems by postponing some decisions to later timesteps. This mirrors human problem-solving, where certain vertices are addressed first to simplify the decision process in subsequent stages. Conversely, in the *without deferral* action strategy, the agent is forced to make decisions immediately, regardless of its confidence, which hinders its performance. Lastly, the *max-color* reward strategy treats the color set as ordered, encouraging the agent to assign colors sequentially from the first color up to k, ensuring the solution is unique in terms of the colors used. This contrasts with the *color-count* strategy, where multiple optimal solutions can exist. For instance, if three colors are needed to color a graph, under the color count strategy, both color assignments '1, 2, 3' and '1, 2, 4' will yield a

color usage penalty of 3. However, under the *max-color* reward strategy, the former will yield a color usage penalty of 3, while the latter will incur a penalty of 4, ensuring a unique and contiguous solution in terms of used colors in the trained model thereby reducing randomness in the learning process by providing an organized and effective approach for the VCP.

In summary, HDM is an MDP configuration developed by combining our novel *max-color* reward strategy with the *deferral* action strategy proposed by Ahn et al. (2020). This configuration outperforms others in terms of training time and graph coloring efficiency, making it the optimal choice.

Table 3: ***Performance comparison across the DIMACS and COLOR02 benchmarks [1/2].*** *Each entry in the table contains either six or two values. Entries with six values report the number of colors used (best solution) and execution time (in seconds), followed by the mean and standard deviation of these two quantities across hundred runs. Entries with only two values omit the mean and standard deviation due to the algorithm being either deterministic or the standard deviations being zero. The best-performing algorithms, in terms of minimizing the number of colors, are boldfaced for each graph. Empty entries indicate the algorithm did not run due to RAM limitations.*

Graph Type	Vertices, Edges	Greedy	TabucolMin	VColMIS	Gurobi	FastColor	ReLCol	VColRL
ash331GPIA	662, 4181	10, $7e^{-4}$	**4**, 111.02	6, 0.61 6.06, 0.31, 0.68, 0.12	**4**, 813.35	**4**, 60.00	7, 2549.96	5, 0.29 5.56, 0.94, 0.09, 0.09
ash608GPIA	1216, 7844	9, $1e^{-3}$	**4**, 254.72	6, 0.77 6.08, 0.27, 0.76, 0.07	7, 900.00	**4**, 60.00	7, 15055.91	5, 0.04 5.79, 1.24, 0.11, 0.11
ash958GPIA	1916, 12506	10, $2e^{-3}$	**4**, 555.93	6, 0.88 6.12, 0.32, 0.87, 0.13	10, 900.00	**4**, 60.00 4.52, 0.50, 60.00, 0.04	7, 60791.44	5, 0.06 6.30, 1.31, 0.15, 0.12
1-FullIns_3	30, 100	8, $3e^{-5}$	**4**, 1.06	**4**, 0.38 4.02, 0.20, 0.26, 0.07	**4**, 0.61	**4**, 60.00	**4**, 0.26	**4**, 0.01 4.73, 0.44, 0.01, 0.006
1-FullIns_4	99 , 593	11, $1e^{-4}$	**5**, 2.20	**5**, 0.19 5.05, 0.26, 0.31, 0.08	**5**, 20.78	**5**, 60.00	**5**, 5.02	**5**, 0.03 5.03, 0.17, 0.03, 0.01
1-FullIns_5	282 , 3247	14, $5e^{-4}$	**6**, 8.84	**6**, 0.32 6.04, 0.24, 0.41, 0.11	**6**, 900.00	**6**, 60.00	8, 150.15	**6**, 0.10 6.21, 0.60, 0.11, 0.07
2-FullIns_3	52 , 201	10, $5e^{-5}$	**5**, 0.94	**5**, 0.42 5.01, 0.10, 0.31, 0.07	**5**, 0.72	**5**, 60.00	**5**, 1.02	**5**, 0.04 5.05, 0.26, 0.03, 0.03
2-FullIns_4	212 , 1621	14, $2e^{-4}$	**6**, 2.31	**6**, 0.56 6.05, 0.21, 0.40, 0.10	**6**, 82.75	**6**, 60.00	**6**, 59.80	**6**, 0.49 6.12, 0.49, 0.21, 0.16
2-FullIns_5	852 , 12201	18, $1e^{-3}$	**7**, 54.40	**7**, 0.54 7.95, 0.47, 0.56, 0.13	**7**, 900.00	**7**, 60.00	8, 5181.63	**7**, 0.06 7.96, 1.36, 0.24, 0.10
3-FullIns_3	80 , 346	12, $7e^{-5}$	**6**, 1.04	**6**, 0.52 6.01, 0.10, 0.51, 0.10	**6**, 3.92	**6**, 60.00	**6**, 2.77	**6**, 0.10 6.37, 0.97, 0.13, 0.08
3-FullIns_4	405 , 3524	17, $6e^{-4}$	**7**, 4.46	**7**, 0.72 7.05, 0.21, 0.62, 0.11	**7**, 513.22	**7**, 60.00	8, 524.77	**7**, 0.23 7.33, 0.96, 0.18, 0.08
4-FullIns_3	114 , 541	14, $1e^{-4}$	**7**, 0.97	**7**, 0.58 7.07, 0.25, 0.60, 0.11	**7**, 5.99	**7**, 60.00	**7**, 7.32	**7**, 0.17 8.23, 1.73, 0.50, 0.18
4-FullIns_4	690, 6650	20, $6e^{-4}$	**8**, 15.04	**8**, 0.58 8.11, 0.34, 0.75, 0.11	**8**, 900.00	**8**, 60.00	9, 2549.94	**8**, 0.65 10.56, 1.38, 0.74, 0.19
5-FullIns_3	154, 792	16, $1e^{-4}$	**8**, 1.14	**8**, 1.01 8.02, 0.14, 0.75, 0.10	**8**, 5.86	**8**, 60.00	**8**, 20.51	**8**, 0.29 11.21, 1.73, 0.37, 0.11
1-Insertions_4	67 , 232	**5**, $5e^{-5}$	**5**, 0.62	**5**, 0.29 5.01, 0.10, 0.19, 0.06	**5**, 31.07	**5**, 60.00	**5**, 1.65	**5**, 0.02 5.01, 0.10, 0.02, 0.02
1-Insertions_5	202 , 1227	**6**, $2e^{-4}$	**6**, 1.24	**6**, 0.31 6.01, 0.10, 0.23, 0.06	**6**, 900.00	**6**, 60.00	**6**, 46.70	**6**, 0.10 6.13, 0.44, 0.10, 0.08
1-Insertions_6	607 , 6337	**7**, $9e^{-4}$	**7**, 12.78	**7**, 0.18 7.07, 0.25, 0.30, 0.11	**7**, 900.00	**7**, 60.00	8, 1698.44	**7**, 0.16 7.54, 0.89, 0.19, 0.11
2-Insertions_3	37 , 72	**4**, $2e^{-5}$	**4**, 0.47	**4**, 0.12	**4**, 0.42	**4**, 60.00	**4**, 0.30	**4**, 0.01 4.38, 0.48, 0.01, 0.005
2-Insertions_4	149 , 541	**5**, $1e^{-4}$	**5**, 0.73	**5**, 0.19 5.08 0.27, 0.22, 0.08	**5**, 900.00	**5**, 60.00	**5**, 16.71	**5**, 0.02 5.01, 0.1, 0.01, 0.009
2-Insertions_5	597 , 3936	**6**, $6e^{-4}$	**6**, 6.31	**6**, 0.30 6.90, 0.36, 0.34, 0.11	**6**, 900.00	**6**, 60.00	**6**, 1636.29	**6**, 0.12 6.15, 0.43, 0.10, 0.09
3-Insertions_3	56 , 110	**4**, $4e^{-5}$	**4**, 0.50	**4**, 0.05	**4**, 5.06	**4**, 60.00	**4**, 0.79	**4**, 0.01 4.50, 0.50, 0.01, 0.005
3-Insertions_4	281 , 1046	**5**, $2e^{-4}$	**5**, 1.25	**5**, 0.22 5.09, 0.32, 0.26, 0.10	**5**, 900.00	**5**, 60.00	**5**, 141.07	**5**, 0.03
3-Insertions_5	1406 , 9695	**6**, $1e^{-3}$	**6**, 35.57	**6**, 0.31 6.95, 0.43, 0.50, 0.15	**6**, 900.00	**6**, 60.00	**6**, 19407.82	**6**, 0.16 6.10, 0.41, 0.20, 0.17
4-Insertions_3	79 , 156	**4**, $5e^{-5}$	**4**, 0.52	**4**, 0.30 4.01, 0.10, 0.20, 0.07	**4**, 14.71	**4**, 60.00	**4**, 2.00	**4**, 0.01 4.76, 0.42, 0.01, 0.005
4-Insertions_4	475 , 1795	**5**, $3e^{-4}$	**5**, 2.23	**5**, 0.25 5.10, 0.30, 0.29, 0.12	**5**, 900.00	**5**, 60.00	**5**, 791.63	**5**, 0.02 5.02, 0.20, 0.02, 0.03

4.2 Performance across different graphs

We evaluate the model trained with the HDM configuration, using the hyperparameter settings specified in Table 1, based on the number of colors used and the execution time, i.e., the total time taken by the algorithms until termination. Due to the stochastic nature of the model, for the graphs with more than 10000 vertices, we integrate a diversification search engine into the model. To do this, we batch 100 disconnected instances of the graph to form a single graph, which is then fed into the model. After obtaining the output, we search for the best solution. In such cases, the reported execution time is inclusive of all these processes.

Table 4: ***Performance comparison across the DIMACS and COLOR02 benchmarks [2/2].*** *Each entry in the table contains either six or two values. Entries with six values report the number of colors used (best solution) and execution time (in seconds), followed by the mean and standard deviation of these two quantities across a hundred runs. Entries with only two values omit the mean and standard deviation due to the algorithm being either deterministic or the standard deviations being zero. The best-performing algorithms, in terms of minimizing the number of colors, are boldfaced for each graph. Empty entries indicate the algorithm did not run due to RAM limitations.*

Graph Type	Vertices, Edges	Greedy	TabucolMin	VColMIS	Gurobi	FastColor	ReLCol	VColRL
le450_5a	450 , 5714	14, $7e^{-4}$	**5**, 510.47	9, 1.08 10.82, 0.90, 1.31, 0.35	7, 900.00	**5**, 35.03 7.14, 0.83, 58.98, 5.10	13, 711.41	6, 0.23 7.48, 1.07, 0.33, 0.20
le450_5b	450 , 5734	13, $7e^{-4}$	**5**, 542.17	9, 0.84 11.06, 0.78, 1.37, 0.29	9, 900.00	**5**, 39.58 7.19, 0.81, 59.60, 0.88	13, 729.42	6, 0.08 7.85, 1.45, 0.25, 0.17
le450_5c	450 , 9803	17, $1e^{-3}$	**5**, 863.32	7, 0.37 9.24, 0.95, 0.62, 0.19	7, 900.00	**5**, 9.20 5.3, 0.46, 33.90, 22.69	16, 725.36	**5**, 0.04 6.69, 1.26, 0.15, 0.10
le450_5d	450 , 9757	18, $1e^{-3}$	7, 39.22	7, 0.18 9.16, 0.91, 0.63, 0.20	13, 900.00	**5**, 12.51	17, 726.04	**5**, 0.08 6.47, 1.28, 0.27, 0.20
mug88_1	88 , 146	**4**, $5e^{-5}$	**4**, 0.50	**4**, 0.50 4.13, 0.33, 0.49, 0.04	**4**, 3.02	**4**, 60.00	**4**, 3.56	**4**, 0.03 4.98, 0.20, 0.03, 0.01
mug88_25	88 , 146	**4**, $5e^{-5}$	**4**, 0.52	**4**, 0.59 4.11, 0.31, 0.49, 0.04	**4**, 3.05	**4**, 60.00	**4**, 3.55	**4**, 0.03 4.98, 0.14, 0.03, 0.01
mug100_1	100 , 166	**4**, $6e^{-5}$	**4**, 0.53	**4**, 0.49 4.04, 0.19, 0.49, 0.03	**4**, 0.49	**4**, 60.00	**4**, 5.49	**4**, 0.04 4.99, 0.1, 0.06, 0.02
mug100_25	100 , 166	**4**, $6e^{-5}$	**4**, 0.54	**4**, 0.47 4.10, 0.30, 0.49, 0.04	**4**, 0.47	**4**, 60.00	**4**, 4.94	**4**, 0.02 5.01, 0.22, 0.03, 0.02
myciel3	11 , 20	**4**, $1e^{-5}$	**4**, 0.46	**4**, 0.19	**4**, 0.04	**4**, 60.00	**4**, 0.03	**4**, 0.009 4.23, 0.42, 0.02, 0.01
myciel4	23 , 71	**5**, $2e^{-5}$	**5**, 0.63	**5**, 0.15	**5**, 0.52	**5**, 60.00	**5**, 0.14	**5**, 0.02 5.02, 0.2, 0.04, 0.06
myciel5	47 , 236	**6**, $6e^{-5}$	**6**, 0.84	**6**, 0.14	**6**, 5.74	**6**, 60.00	**6**, 0.74	**6**, 0.07 6.27, 0.70, 0.27, 0.21
myciel6	95 , 755	**7**, $1e^{-4}$	**7**, 1.29	**7**, 0.11 7.24, 0.42, 0.29, 0.08	**7**, 900.00	**7**, 60	**7**, 5.13	**7**, 0.06 7.15, 0.50, 0.26, 0.18
myciel7	191, 2360	**8**, $2e^{-3}$	**8**, 2.87	**8**, 0.14 8.23, 0.42, 0.32, 0.10	**8**, 900.00	**8**, 60.00	**8**, 42.23	**8**, 0.32 9.31, 0.81, 0.32, 0.02
queen5_5	25 , 160	8, $4e^{-5}$	**5**, 5.15	**5**, 0.11 5.91, 1.01, 0.27, 0.13	**5**, 0.11	**5**, 0.001	**5**, 0.16	**5**, 0.03 5.29, 0.59, 0.05, 0.02
queen6_6	36 , 290	11, $6e^{-5}$	**7**, 4.91	8, 0.39 9.20, 0.65, 0.69, 0.15	**7**, 0.39	**7**, 60.00	8, 0.43	**7**, 0.21 10.16, 1.16, 0.33, 0.06
queen7_7	49 , 476	10, $8e^{-5}$	**7**, 14.85	8, 0.53 10.30, 0.79, 0.79, 0.15	**7**, 0.73	**7**, 1.20	9, 0.85	**7**, 0.27 12.97, 1.86, 0.55, 0.18
DSJC125.1	125 , 736	8, $1e^{-4}$	**5**, 12.58	6, 0.57 7.53, 0.62, 0.72, 0.18	**5**, 23.26	**5**, 60.00	7, 11.40	6, 0.10 6.62, 0.48, 0.11, 0.02
will199GPIA	701 , 6772	11, $9e^{-4}$	**7**, 19.16	8, 2.55 10.05, 0.84, 1.52, 0.58	9, 900.00	**7**, 60.00	9, 3288.72	**7**, 0.23 10.10, 1.52, 0.32, 0.05

Baselines: For our experiments, we compare the performance of VColRL against six baselines. The first baseline is a straightforward **Greedy Coloring** approach, which colors graph vertices sequentially while ensuring that no two adjacent vertices share the same color. The second baseline is **TabucolMin**, an enhanced version of the TabuCol algorithm (Hertz & Werra, 1987), incorporating an additional minimization step to refine the coloring. The third baseline is **VColMIS**, which we develop based on the maximum independent set (MIS) problem. It operates as follows: given an input graph, the MIS is identified and colored with color '1,' followed by identifying and coloring the MIS on the remaining subgraph with color '2,' and so on. We use the model by Ahn et al. (2020) for the MIS engine and train it on the same dataset as VColRL, following the hyperparameters recommended by the authors. The fourth baseline is the **Gurobi 12** solver (Gurobi Optimization, 2024), a state-of-the-art optimization solver. The fifth baseline is **FastColor** (Lin et al., 2017), a reduction-based method based on Bounded Independent Set (BIS). Lastly, we include **ReLCol** (Watkins et al., 2023), a heuristic for the VCP based on Deep Q-Networks (DQN) and GNN. We directly use the trained model provided by the authors. Further details about these baselines are provided in Appendix B.3.

Performance on DIMACS, COLOR02, NR and SNAP benchmarks: The performance of various algorithms is evaluated on a subset of the DIMACS and COLOR02 (Trick, 2002-2004), NR (Rossi & Ahmed, 2015), and SNAP (Leskovec & Krevl, 2014) benchmark datasets, focusing on graphs with chromatic numbers in the range of $4-8$, aligning with the $4-7$ range of the training dataset. For Gurobi, the time limit is set to 900 seconds, while for FastColor, the cutoff time is set to 60 seconds, as used in the original paper.

For DIMACS and COLOR02 benchmarks in Table 3 and Table 4, VColRL performs equivalently or better than Greedy, VColMIS, and ReLCol in terms of using fewer colors across all graphs. It outperforms these baselines on $\sim 53\%$, $\sim 23\%$, and $\sim 37\%$ of the graphs, using an average of 6.73, 1.70, and 3.43 fewer colors, respectively. Compared to Gurobi, TabucolMin, and FastColor, it performs equivalently or better on $\sim 95\%$,

$\sim 86\%$, $\sim 86\%$ of the graphs, respectively. VColRL on average uses 3.28 fewer colors than Gurobi and 2 fewer colors than TabucolMin. In terms of execution time, VColRL is $2-1000000\times$, $2-45000\times$, $4-20000\times$, $4-6000\times$, and $1-40\times$ faster than ReLCol, Gurobi, TabucolMin, FastColor, and VColMIS, respectively.

Table 5: ***Performance comparison across the NR and SNAP benchmarks.*** *Each entry in the table contains either six or two values. Entries with six values report the number of colors used (best solution) and execution time (in seconds), followed by the mean and standard deviation of these two quantities across hundred runs. Entries with only two values omit the mean and standard deviation due to the algorithm being either deterministic or the standard deviations being zero. The best-performing algorithms, in terms of minimizing the number of colors, are boldfaced for each graph. Empty entries indicate the algorithm did not run due to RAM limitations.*

Graph Type	Vertices, Edges	Greedy	TabucolMin	VColMIS	Gurobi	FastColor	ReLCol	VColRL
ia-reality	6809, 7680	**5**, $2e^{-4}$	14, 30.08	**5**, 0.53 5.74, 0.44, 0.50, 0.08	**5**, 23.26	**5**, 0.21	-	**5**, 0.09 5.30, 0.7, 0.13, 0.11
ia-fb-messages	1266, 6541	13, $1e^{-3}$	8, 35.02	10, 1.21 11.73, 0.64, 1.34, 0.14	11, 900.00	**6**, 60.00	12, 13648.69	9, 0.72 11.60, 0.80, 0.75, 0.09
p2p-Gnutella04	10879, 39994	8, $9e^{-3}$	6, 1188.80	8, 25.57 8.97, 0.41, 25.82, 0.36	14, 900.00	**5**, 60.00 5.3, 0.46, 60.00, 0.21	-	**5**, 16.06 6.95, 0.43, 15.93, 0.21
p2p-Gnutella24	26518, 65369	9, $2e^{-2}$	6, 3640.00	8, 43.42 8.50, 0.52, 43.66, 0.42	15, 900.00	**5**, 60.00	-	**5**, 25.17 5.06, 0.23, 25.27, 0.21
p2p-Gnutella25	22687, 54705	8, $1e^{-2}$	6, 2229.00	8, 35.39 8.17, 0.37, 35.38, 0.30	15, 900.00	**5**, 60.00	-	**5**, 22.12
p2p-Gnutella30	36682, 88328	8, $2e^{-2}$	6, 7319.00	8, 54.08 8.93, 0.40, 53.36, 0.53	10, 900.00	**5**, 60.00	-	**5**, 31.49 5.26, 0.44, 30.97, 0.25
p2p-Gnutella31	62586, 147892	8, $4e^{-2}$	6, 29700.00	9, 95.18 9.08, 0.27, 97.29, 2.51	15, 900.00	**5**, 60.00	-	**5**, 49.05 5.02, 0.14, 49.13, 0.47
rt-retweet	96, 117	5, $5e^{-5}$	**4**, 0.86	**4**, 0.32 4.21, 0.40, 0.35, 0.08	**4**, 0.19	**4**, 0.004	**4**, 3.31	**4**, 0.01 4.98, 0.14, 0.01, 0.004
rt-twitter-copen	761, 1029	7, $4e^{-5}$	5, 3.16	6, 0.70 7.63, 0.56, 0.80, 0.12	**4**, 27.12	**4**, 0.03	7, 2075.86	5, 0.08 5.40, 0.63, 0.07, 0.06
soc-dolphins	62, 159	7, $5e^{-6}$	**5**, 1.57	**5**, 0.44 6.05, 0.29, 0.56, 0.21	**5**, 0.31	**5**, 0.01	**5**, 1.39	**5**, 0.04 5.11, 0.34, 0.03, 0.02
soc-karate	34, 78	6, $4e^{-5}$	**5**, 1.03	**5**, 0.36 5.20, 0.40, 0.40, 0.10	**5**, 0.008	**5**, 0.001	**5**, 0.31	**5**, 0.01 5.03, 0.22, 0.06, 0.08
soc-wiki-vote	889, 2914	10, $6e^{-4}$	**7**, 23.88	11, 1.25 12.27, 0.72, 1.37, 0.17	**7**, 95.08	**7**, 0.42	12, 4533.15	**7**, 0.40 11.46, 1.25, 0.74, 0.13
bio-yeast	1458, 1948	**6**, $7e^{-4}$	**6**, 2.90	**6**, 0.85 6.13, 0.33, 0.85, 0.06	**6**, 106.32	**6**, 0.07	**6**, 12778.59	**6**, 0.24 7.51, 1.73, 0.25, 0.12
inf-power	4941, 6594	**6**, $2e^{-3}$	**6**, 28.42	**6**, 1.06 6.82, 0.51, 1.09, 0.15	**6**, 471.27	**6**, 0.22	-	**6**, 0.31 6.35, 0.89, 0.33, 0.19
tech-p2p-gnutella	62561, 147878	8, $4e^{-2}$	6, 29297.00	9, 101.23 9.02, 0.14, 98.27, 1.11	15, 900.00	**5**, 60.00 5.02, 0.14, 60.00, 2.25	-	**5**, 49.00 5.02, 0.14, 49.28, 0.19

For SNAP and NR benchmarks in Table 5, VColRL performs equivalently or better than Greedy, VColMIS, and ReLCol in terms of using fewer colors across all graphs. VColRL outperforms these baselines on $\sim 80\%$, $\sim 60\%$, and $\sim 42\%$ of the graphs, using an average of 2.66, 2.88, and 3.33 fewer colors, respectively. Compared to Gurobi, TabucolMin, and FastColor, VColRL performs equivalently or better on $\sim 93\%$, $\sim 93\%$, $\sim 87\%$ of the graphs, respectively. VcolRL outerforms Gurobi and TabucolMin on $\sim 47\%$ graphs. VColRL on average uses 8 fewer colors than Gurobi and 2.14 fewer colors than TabucolMin. In terms of execution time, VColRL is $30-50000\times$, $1-1000\times$, $10-600\times$, and $1-36\times$ faster than ReLCol, Gurobi, TabucolMin, and VColMIS, respectively. Compared to FastColor, VColRL is $1-80\times$ faster on 60% graphs, while for the remaining graphs, FastColor is $1-10\times$ faster.

Table 6: ***Performance comparison across different synthetic graph types and vertex ranges.*** *Each entry in the table has two values: the average number of colors used, and the average execution time in seconds. The objectives of the best-performing methods in terms of minimizing color usage are boldfaced. In ER graphs, p is the probability of edge creation; in BA, n is the number of edges to attach from a new vertex to existing vertices; whereas in WS, n is the number of nearest neighbors to which each vertex is joined within a ring topology, and p is the probability of rewiring each edge.*

Graph Type	Vertex Range	Greedy	TabucolMin	VColMIS	Gurobi	FastColor	ReLCol	VColRL
ER (p=0.15)	50-100	7.648, $8e^{-5}$	**5.252**, 5.55	7.216, 0.57	5.322, 5.02	5.264, 53.05	6.294, 2.83	5.592, 0.36
ER (p=0.125)	100-150	9.330, $1e^{-4}$	**6.240**, 10.87	8.688, 0.74	6.674, 10.00	6.492, 60.00	9.056, 11.90	6.760, 0.65
ER (p=0.10)	150-200	10.054, $2e^{-4}$	**6.554**, 15.53	9.274, 0.86	7.340, 10.00	6.998, 60.00	10.140, 33.67	7.528, 0.80
BA (n=6)	50-100	6.682, $8e^{-5}$	6.110, 3.39	9.054, 0.48	**6.008**, 4.06	**6.008**, 21.53	7.160, 2.88	6.168, 0.46
BA (n=6)	100-150	6.866, $1e^{-4}$	6.238, 4.55	9.960, 0.57	**6.028**, 8.41	**6.028**, 24.65	9.054, 11.55	6.686, 0.51
BA (n=6)	150-200	6.920, $1e^{-4}$	6.412, 6.38	10.442, 0.62	**6.030**, 9.94	**6.030**, 26.55	9.858, 32.99	6.898, 0.70
WS (n=15, p=0.30)	50-100	8.778, $1e^{-4}$	**6.574**, 7.63	8.604, 0.67	6.838, 7.52	6.678, 31.78	7.986, 2.82	6.996, 0.54
WS (n=15, p=0.30)	100-150	8.968, $2e^{-4}$	**6.426**, 10.11	8.716, 0.72	7.004, 10.00	6.964, 37.47	9.206, 12.06	7.008, 0.70
WS (n=15, p=0.30)	150-200	9.038, $2e^{-4}$	**6.490**, 11.74	8.804, 0.90	7.022, 10.00	7.006, 35.64	9.512, 33.69	7.014, 0.82

Performance on synthetic random graphs: We record the performance of various algorithms on different types of graphs in Table 6, comprising ER (Erdos et al., 1960), BA (Albert & Barabási, 2002), WS (Watts & Strogatz, 1998) graphs. We evaluate 500 random graphs for each row in the table. We set a time limit of 10 seconds for Gurobi, as it would otherwise continue searching for optimal solutions for an extended period, making it impractical to evaluate a plethora of graphs within a reasonable timeframe. The cutoff timer for FastColor is set to 60 seconds.

We observe that VColRL outperforms Greedy, VColMIS, and RelCol in terms of using fewer colors across all graph types and ranges, while using on average 1.51, 2.23, and 1.95 fewer colors, respectively. It achieves competitive performance compared to Gurobi, TabuColMin, and FastColor in terms of color usage, using less than one additional color on average. However, VColRL offers a significant advantage in execution time, being on average $\sim 14\times$ faster than Gurobi, $\sim 13\times$ faster than TabucolMin, $\sim 23\times$ faster than ReLCol, $1.5\times$ faster than VColMIS, and $\sim 60\times$ faster than FastColor, thus demonstrating a good balance of efficiency and performance.

Generalization capability and scalability: From Table 3, Table 4, and Table 5, we observe that VColRL outperforms or matches the best-performing baseline on 50 out of 58 tested graph instances ($\sim 86\%$ graphs) in terms of color usage. On the graphs where VColRL slightly underperforms, it requires only 1.125 additional colors on average compared to the best-performing baseline. This small performance difference highlights the strong generalization ability of VColRL across diverse graph types and sizes. Although the model is trained exclusively on a dataset containing ER graphs, it performs well on test graphs drawn from a variety of unseen distributions. To further support our generalization claim, we conduct an additional experiment in which the model is trained on a mixed dataset comprising ER, BA, and WS graphs. We observe that both the ER-trained and mixed-trained models perform comparably, indicating that VColRL possesses strong generalization capability even without explicit exposure to the target graph distributions during training. For details, refer to Appendix C.3. Furthermore, VColRL exhibits a significant speed advantage over all baselines, except Greedy, which is the most basic heuristic with poor performance in terms of color usage. For some instances, it is $\sim 45000\times$, $\sim 20000\times$, $\sim 6000\times$, and $\sim 40\times$ faster than Gurobi, TabucolMin, FastColor, and VColMIS, respectively. Compared to ReLCol, which is also an RL-based baseline that uses GNN, VColRL is $120000 - 1000000\times$ faster on certain graphs such as `3-Insertions_5` and `ash958GPIA`. Despite both methods being RL-based and GNN-driven, this performance gap stems from architectural differences. ReLCol employs a complex and computationally expensive model that processes both vertex and edge features and evaluates the entire graph at each MDP step, incurring substantial time and computational cost. In contrast, VColRL adopts a reduction-based neural model that processes only vertex features and focuses on subgraphs of uncolored vertices during MDP steps. This reduction is enabled by a carefully designed feature vector outlined in Section 3.2 that effectively conveys information about the colored portion of the graph to the sampled subgraph, making VColRL a scalable solution.

5 Conclusion and Future Work

This paper presents VColRL, a deep reinforcement learning framework for solving the VCP. VColRL is based on a novel MDP formulation with a reward mechanism that minimizes the highest-numbered color used. Through extensive experimentation, we identify the HDM configuration as the most effective for this task. Empirical results on a wide range of synthetic and benchmark graph instances show that VColRL consistently outperforms or matches Greedy, VColMIS, and ReLCol in terms of color usage. It also performs competitively with Gurobi, TabuColMin, and FastColor, requiring only about one additional color on average in cases where it slightly underperforms. In terms of runtime, VColRL achieves significant improvements, being approximately $1000000\times$ faster than ReLCol, $45000\times$ faster than Gurobi, $20000\times$ faster than TabuColMin, $6000\times$ faster than FastColor, and $40\times$ faster than VColMIS on certain graph instances. These results demonstrate that VColRL offers a strong balance of solution quality and computational efficiency, making it a scalable and generalizable approach for solving the VCP.

Potential future direction involves using imitation learning (Hussein et al., 2017) and integrating advanced search algorithms, leveraging the solution provided by VColRL as a starting point. For more details, refer to Appendix C.2.

References

Shamim Ahmed. Applications of graph coloring in modern computer science. *International Journal of Computer and Information Technology*, 3(2):1–7, 2012.

Sungsoo Ahn, Younggyo Seo, and Jinwoo Shin. Learning what to defer for maximum independent sets. In *International conference on machine learning*, pp. 134–144. PMLR, 2020.

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

Saif Al-Sultan, Moath M Al-Doori, Ali H Al-Bayatti, and Hussien Zedan. A comprehensive survey on vehicular ad hoc network. *Journal of network and computer applications*, 37:380–392, 2014.

Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

Murat Aslan and Nurdan Akhan Baykan. A performance comparison of graph coloring algorithms. *International Journal of Intelligent Systems and Applications in Engineering*, 4(Special Issue-1):1–7, 2016.

George D Birkhoff. A determinant formula for the number of ways of coloring a map. *Annals of Mathematics*, 14(1/4):42–46, 1912.

A. Burkov. *The Hundred-Page Machine Learning Book*. Andriy Burkov, 2019. ISBN 9781999579517. URL `https://books.google.co.in/books?id=0jbxwQEACAAJ`.

Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

Chase Cummins and Richard Veras. Reinforcement learning for graph coloring: Understanding the power and limits of non-label invariant representations. *arXiv preprint arXiv:2401.12470*, 2024.

Dibyendu Das, Shahid Asghar Ahmad, and Kumar Venkataramanan. Deep learning-based hybrid graph-coloring algorithm for register allocation. *arXiv preprint arXiv:1912.03700*, 2019.

Alane Marie de Lima and Renato Carmo. Exact algorithms for the graph coloring problem. *Revista de Informática Teórica e Aplicada*, 25(4):57–73, 2018.

Tansel Dokeroglu and Ender Sevinc. Memetic teaching–learning-based optimization algorithms for large graph coloring problems. *Engineering Applications of Artificial Intelligence*, 102:104282, 2021.

Hao Dong, Hao Dong, Zihan Ding, Shanghang Zhang, and T Chang. *Deep Reinforcement Learning*. Springer, 2020.

Paul Erdos, Alfréd Rényi, et al. On the evolution of random graphs. *Publ. math. inst. hung. acad. sci*, 5(1): 17–60, 1960.

Damien Ernst and Arthur Louette. Introduction to reinforcement learning. *Feuerriegel, S., Hartmann, J., Janiesch, C., and Zschech, P.(2024). Generative ai. Business & Information Systems Engineering*, 66(1): 111–126, 2024.

Thomas A Feo, Mauricio GC Resende, and Stuart H Smith. A greedy randomized adaptive search procedure for maximum independent set. *Operations Research*, 42(5):860–878, 1994.

Michael R Garey and David S Johnson. The complexity of near-optimal graph coloring. *Journal of the ACM (JACM)*, 23(1):43–49, 1976.

Lukas Gianinazzi, Maximilian Fries, Nikoli Dryden, Tal Ben-Nun, Maciej Besta, and Torsten Hoefler. Learning combinatorial node labeling algorithms. *arXiv preprint arXiv:2106.03594*, 2021.

Jean-Louis Goffin and Jean-Philippe Vial. Convex nondifferentiable optimization: A survey focused on the analytic center cutting plane method. *Optimization methods and software*, 17(5):805–867, 2002.

Gurobi Optimization. Gurobi optimizer reference manual, 2024. URL `https://www.gurobi.com`. Version 12.0, Available at `https://www.gurobi.com`.

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

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

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

Jeroen Hoebeke, Ingrid Moerman, Bart Dhoedt, and Piet Demeester. An overview of mobile ad hoc networks: applications and challenges. *Journal-Communications Network*, 3(3):60–66, 2004.

Jiayi Huang, Mostofa Patwary, and Gregory Diamos. Coloring big graphs with alphagozero. *arXiv preprint arXiv:1902.10162*, 2019.

Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

IBM. Ibm ilog cplex optimization studio, 2024. URL `https://www.ibm.com/products/ilog-cplex-optimization-studio`.

Ali Zeeshan Ijaz, Raja Hashim Ali, Nisar Ali, Talha Laique, and Talha Ali Khan. Solving graph coloring problem via graph neural network (gnn). In *2022 17th International Conference on Emerging Technologies (ICET)*, pp. 178–183. IEEE, 2022.

Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(6):4909–4926, 2021.

Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–6, 2015.

M. Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more*. Packt Publishing, 2020. ISBN 9781838820046. URL `https://books.google.co.in/books?id=O0vODwAAQBAJ`.

Yin Tat Lee, Aaron Sidford, and Sam Chiu-wai Wong. A faster cutting plane method and its implications for combinatorial and convex optimization. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pp. 1049–1065. IEEE, 2015.

Henrique Lemos, Marcelo Prates, Pedro Avelar, and Luis Lamb. Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 879–885. IEEE, 2019.

Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

Jinkun Lin, Shaowei Cai, Chuan Luo, and Kaile Su. A reduction based method for coloring very large graphs. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 517–523, 2017. doi: 10.24963/ijcai.2017/73. URL `https://doi.org/10.24963/ijcai.2017/73`.

Arno Luppold, Dominic Oehlert, and Heiko Falk. Evaluating the performance of solvers for integer-linear programming, 2018. URL http://tubdok.tub.tuhh.de/handle/11420/1842.

Raymond E Miller and David E Muller. A problem of maximum consistent subsets. Technical report, IBM Research Report RC-240, JT Watson Research Center, Yorktown Heights, NY, 1960.

Eduardo F Morales, Rafael Murrieta-Cid, Israel Becerra, and Marco A Esquivel-Basaldua. A survey on deep learning and deep reinforcement learning in robotics with a tutorial on deep reinforcement learning. *Intelligent Service Robotics*, 14(5):773–805, 2021.

David R Morrison, Sheldon H Jacobson, Jason J Sauppe, and Edward C Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19: 79–102, 2016.

Taha Mostafaie, Farzin Modarres Khiyabani, and Nima Jafari Navimipour. A systematic study on meta-heuristic approaches for solving the graph coloring problem. *Computers & Operations Research*, 120: 104850, 2020.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL https://networkrepository.com.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

Michael Trick. Color02/03/04 benchmark datasets. https://mat.tepper.cmu.edu/COLOR02/, 2002-2004.

A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

Bernhard Friedrich Voigt. Der handlungsreisende, wie er sein soll und was er zu thun hat, um aufträge zu erhalten und eines glücklichen erfolgs in seinen geschäften gewiss zu sein. *Commis-Voageur, Ilmenau*, pp. 69–72, 1831.

George Watkins, Giovanni Montana, and Juergen Branke. Generating a graph colouring heuristic with deep q-learning and graph neural networks. In *International Conference on Learning and Intelligent Optimization*, pp. 491–505. Springer, 2023.

Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684): 440–442, 1998.

A ADDITIONAL DETAILS ABOUT VColRL FRAMEWORK

A.1 VColRL Architecture

Figure 4 illustrates the architecture of VColRL, which consists of an agent and an environment. The agent interacts with the environment by exploring different actions and stores key information, including the state, action, reward, and episode termination status, in a replay buffer. The returns and advantages (defined in Appendix A.3) are calculated using data from the replay buffer, which are then utilized by the optimizers as feedback to minimize the total objective function defined in Appendix A.3.

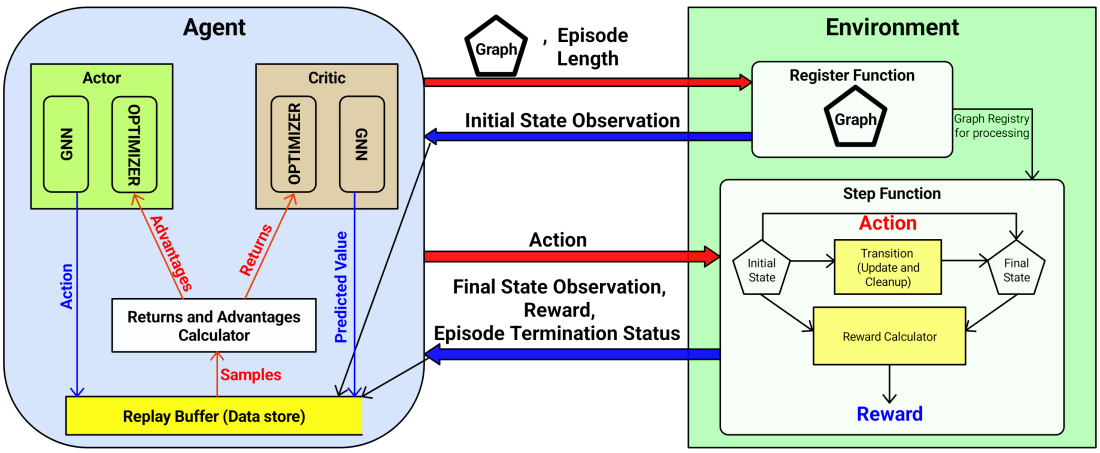


Figure 4: Architecture of VColRL

A.2 GraphSAGE Architecture

The GNN component in Figure 4 follows the GraphSAGE architecture (Hamilton et al., 2017), illustrated in Figure 5, where the vertex feature vectors are represented using solid arrows, while the flow of information is depicted with regular arrows. It has three modules: *Sampling and Normalization*, *Feature Aggregation and Concatenation*, and *Transformation*.

In the *Sampling and Normalization* module, first, the subgraph containing uncolored vertices is sampled, and then the vertex features are normalized by multiplying each feature vector by the inverse square root of the vertices' degree, thereby stabilizing the learning process. After the normalization, each vertex samples its entire neighborhood to aggregate the vertex features of neighbors to be able to capture the information about its neighborhood. GraphSAGE typically allows for sampling a subset of neighboring vertices; however, in our approach, we sample the entire neighborhood. This ensures that all relevant vertices are considered for efficient processing, which is essential for accurate feature aggregation.

In the *Feature Aggregation and Concatenation* module, the features of the sampled vertices are aggregated and normalized again. The aggregated features are then concatenated with the original vertex features, resulting in a feature vector that is twice the length of the input vector. This helps to retain the information about the vertex and its neighborhood, providing a comprehensive representation for further processing.

In the *Transformation* module, the concatenated and aggregated features (represented as a matrix with the number of rows equals the number of vertices in the graph) are multiplied by a weight matrix, followed by the addition of a bias matrix. Both these matrices are learnable components of our model. An activation function is then applied to produce the final output, enabling the model to capture complex relationships in the data. The final output feature vectors then become the input for the next layer, allowing the model to propagate information through multiple layers.

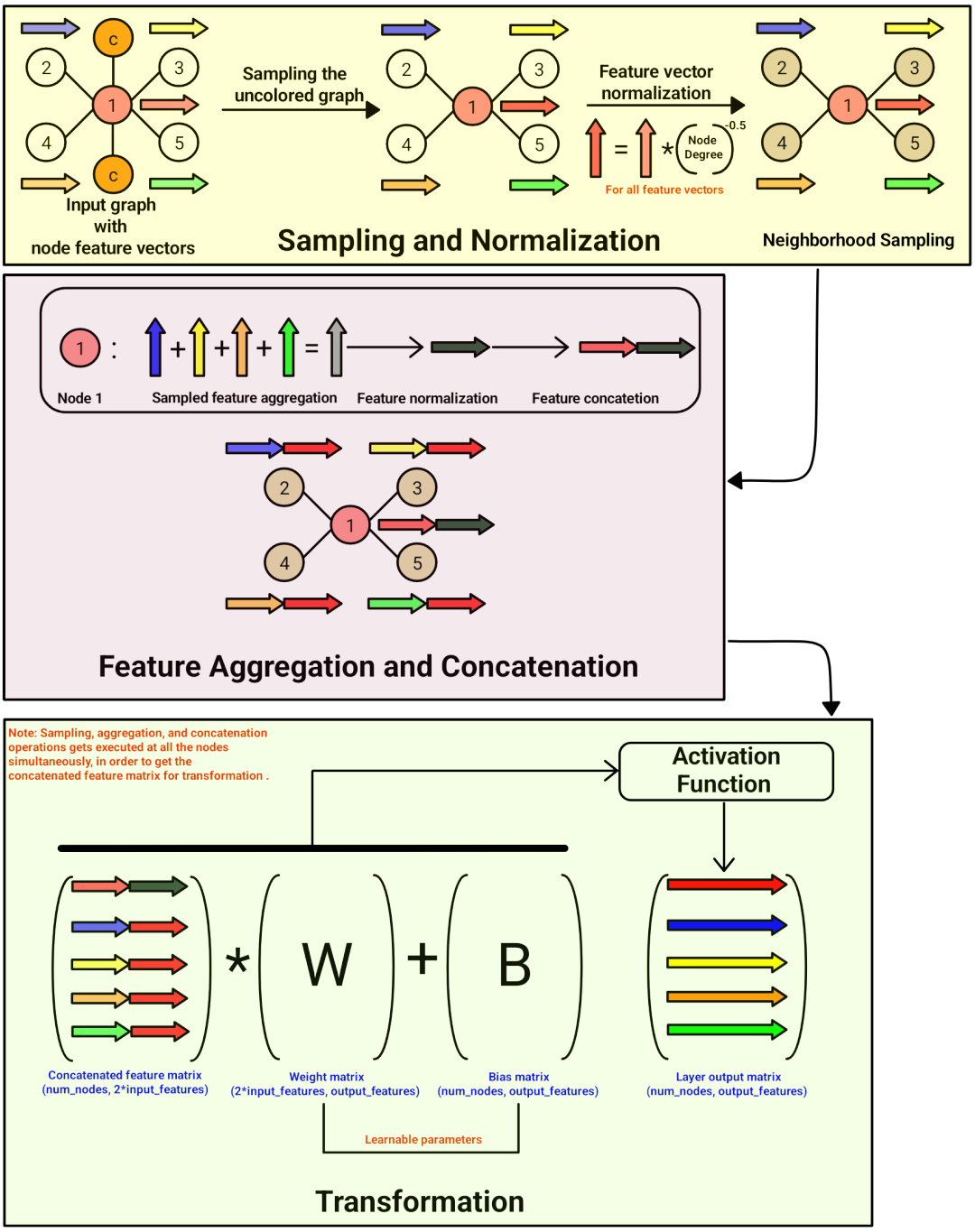


Figure 5: GraphSAGE architecture

A.3 Proximal Policy Optimization with entropy regularization

The optimizer component in Figure 4 uses the Proximal Policy Optimization algorithm (PPO, (Schulman et al., 2017)) to train the agent to solve the VCP problem. PPO is an actor-critic-based reinforcement learning algorithm that employs two parameterized networks: the policy network $\pi(a_t \mid s_t; \theta)$ and the value

network $V(s_t;\omega)$, where θ and ω are the parameters of the function estimators. In our case, these are Graph Neural Networks with GraphSAGE (Hamilton et al., 2017) architecture as shown in Figure 5.

The policy network $\pi(a_t \mid s_t;\theta)$ provides the probabilities of selecting an action a_t, given a state s_t at time t. The action is determined by sampling from this probability distribution.

On the other hand, the value network $V(s_t;\omega)$ estimates the expected return $\mathbb{E}_\pi[G_t \mid s_t,\omega]$ from the current state s_t, where G_t is defined as the discounted sum of future rewards:

$$G_t = \sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k}$$

Here, r_{t+k} denotes the reward received at time $t+k$, and γ is the discount factor that balances the importance of immediate versus future rewards.

Additionally, PPO uses the advantage function to improve the policy. The advantage function $\hat{A}_t$ is defined as the difference between the actual return from the episode and the estimated value of the state predicted by the old episode-generating policy:

$$\hat{A}_t = G_t - V(s_t;\omega_{old})$$

The objective function of PPO involves maximizing a surrogate loss function to improve the policy while ensuring stability. The objective function can be expressed as:

$$\mathcal{L}_{\text{actor}}(\theta_{new}) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right]$$

where $r_t(\theta) = \frac{\pi(a_t|s_t;\theta_{\text{new}})}{\pi(a_t|s_t;\theta_{\text{old}})}$ is the probability ratio of the new policy (the policy being optimized in the *gradient step*) to the old episode-generating policy, and ϵ is a hyperparameter that defines the clipping range to prevent large policy updates.

In addition to the policy loss, we have critic loss and entropy regularization to guide the training process. The critic loss is responsible for training the value network $V(s_t;\omega)$ by minimizing the difference between the predicted value from the value network of the new policy (the policy being optimized in the *gradient step*) and the actual return from the episode. This is done using a squared error loss between the value estimate and the discounted return G_t. The critic loss function is given by:

$$\mathcal{L}_{\text{critic}}(\omega_{new}) = \mathbb{E}_t\left[\left(V(s_t;\omega_{new}) - G_t\right)^2\right]$$

The entropy regularization term is added to encourage exploration and prevent premature convergence to suboptimal policies. The entropy of the policy $\pi(a_t \mid s_t;\theta_{new})$ measures the uncertainty of the action selection and is defined as:

$$\mathcal{H}(\pi(a_t \mid s_t;\theta_{new})) = -\sum_{a_t} \pi(a_t \mid s_t;\theta_{new}) \log \pi(a_t \mid s_t;\theta_{new})$$

For using an optimizer that works by minimizing an objective function, we must adjust the signs of the actor loss and the entropy term accordingly. The actor loss should be negated because we want to maximize the policy objective, while the critic loss remains as it is, as we want to minimize the value estimation error. The entropy term is subtracted to maximize exploration by encouraging higher entropy.

Thus, the total minimization objective function for PPO, combining the actor loss, critic loss, and entropy regularization, is given by:

$$\mathcal{L}_{\text{total}}(\theta_{new}, \omega_{new}) = -\mathcal{L}_{\text{actor}}(\theta_{new}) + c_1\mathcal{L}_{\text{critic}}(\omega_{new}) - c_2\mathcal{H}(\pi(a_t \mid s_t; \theta_{new}))$$

where c_1, $c_2 \in \mathbb{R}^+$ are hyperparameters that control the balance between actor loss, critic loss, and entropy regularization.

In summary, the actor learns to adjust the policy to increase the probability of selecting actions in a state for which the advantage is maximized, thereby promoting actions that yield higher returns. The critic loss helps the agent predict the returns from states, while the entropy regularization encourages the exploration of action space, leading to more robust policies.

B Additional Experimental Details

B.1 Hyperparameter Tuning

The hyperparameters *Discount factor for PPO* and *Clip value for PPO* are set to 1 and 0.2, respectively. The *Discount factor for PPO* is set to 1 to ensure that rewards from all steps are equally weighted as required for the VCP, while the *Clip value for PPO* is set to 0.2, as it is commonly used in several implementations, including Stable Baselines3's (Raffin et al., 2021) PPO. We use the Optuna framework (Akiba et al., 2019) with its Tree-structured Parzen Estimator (TPE) sampler and Median Pruner to tune the remaining hyperparameters. The Optuna experiment iteratively samples hyperparameters using the sampler and trains the model. To save time, unpromising trials are pruned by the pruner before completing all epochs. Figure 6 illustrates an example of the Optuna experiment, where the number of epochs is set to 70.

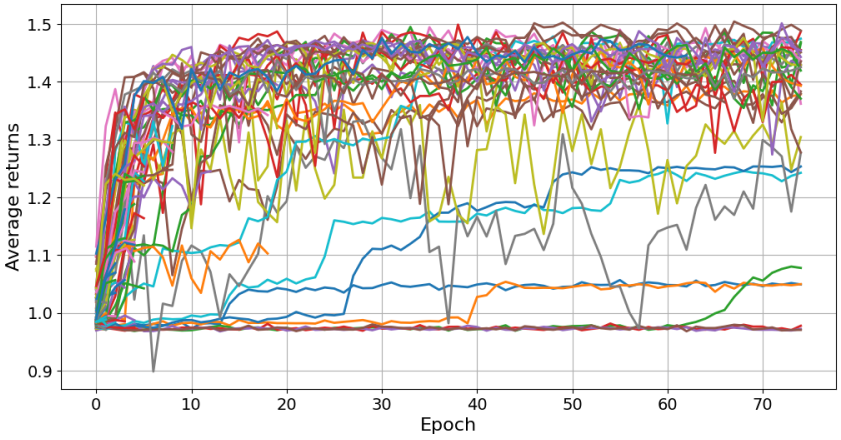


Figure 6: ***Optuna experiment.*** *Each curve represents a trial in the experiment.*

We begin the tuning process by first finalizing the *Hidden dimension* and *Number of hidden layers*. We explore the combinations of *Hidden dimension* $= \{64, 128, 256, 512\}$ and *Number of hidden layers* $= \{3, 4, 5\}$ sampled by the TPE sampler while setting the other hyperparameters to the values suggested by Ahn et al. (2020) for the MIS problem. Compared to the final selected configuration of *Hidden dimension* and *Number of hidden layers*, all other combinations either give an equivalent or worse performance in terms of maximizing the reward. Balancing performance and complexity, we finalize the architecture with a *Hidden dimension* of 128 and *Number of hidden layers* set to 4. After finalizing the model architecture, we run the tuning experiment again with the remaining hyperparameters. We find that most hyperparameters have minimal impact on performance, except for *Learning rate of optimizer*, *Critic loss coefficient*, and *Entropy regularization coefficient*. Optuna's hyperparameter importance analysis tool highlights these three as having the most significant effect on performance. Consequently, we fix the remaining hyperparameters to the values suggested by Ahn et al. (2020), except for the *Gradient norm*, which we set to 1 to prevent the gradient

step from becoming too small. To arrive at the final hyperparameter setting present in Table 1, we conduct the final round of optimization involving the key hyperparameters *Learning rate of optimizer*, *Critic loss coefficient*, and *Entropy regularization coefficient*.

The values and ranges chosen for the Optuna experiments are as follows: *Critic loss coefficient* $\in [0.1, 0.9]$, *Entropy regularization coefficient* $\in [0, 0.5]$, *Gradient norm* $\in \{0.5, 1, 3, 5\}$, *Learning rate of optimizer* $\in [10^{-5}, 10^{-3}]$, *Hidden dimension* $\in \{64, 128, 256, 512\}$, *Number of hidden layers* $\in \{2, 3, 4, 5\}$, *Number of gradient steps per update* $\in \{2, 4, 8\}$, *Batch size for gradient step* $\in \{4, 8, 16\}$, *Rollout batch size* $\in \{16, 32, 64\}$.

B.2 Normalization of Rewards

Since our immediate reward is the weighted sum of the vertex satisfaction reward and the color usage penalty, we normalize each component by dividing it by a relevant factor before computing the sum. Specifically, the vertex satisfaction reward is divided by the number of vertices, and the color usage penalty is divided by the number of colors in set C. This ensures that both components are scaled appropriately and that neither dominates the reward function. While an additional normalization step could be applied after combining the two components, we do not adopt this in our approach, as empirical observations showed that the model exhibited more stable training without it.

B.3 Implementation of baselines

B.3.1 Greedy Coloring

Algorithm 1 Greedy Coloring

```
V ← Number of vertices in Graph
result ← array of length V initialized with -1
result[0] ← 0
available ← array of length V initialized with True
max_color ← 0
for each vertex u ranging from 1 to V − 1 do
  for each neighbor i of vertex u in Graph do
    if result[i] ≠ −1 then
      available[result[i]] ← False
    end if
  end for
  cr ← 0
  while cr < V do
    if available[cr] == True then
      break
    end if
    cr ← cr + 1
  end while
  result[u] ← cr
  if cr > max_color then
    max_color ← cr
  end if
  available ← array of length V initialized with True //resetting available colors
end for
return max_color +1
```

This baseline assigns colors to vertices in a sequential manner, ensuring that no two adjacent vertices share the same color. The algorithm works by iterating through each vertex and selecting the smallest available color that has not been assigned to its neighboring vertices. The pseudocode for this is available in Algorithm 1.

B.3.2 TabucolMin

This algorithm is an iterative metaheuristic designed to minimize the number of colors required to properly color a graph, leveraging the Tabucol algorithm as a subroutine. Hertz & Werra (1987) contains the pseudocode and details about the Tabucol algorithm. TabucolMin begins with an initial color count, typically set to 15, and attempts to color the graph. If the graph can be successfully colored with the current number of colors, the algorithm reduces the color count by one and retries. Conversely, if the color count cannot be reduced further and no solution has yet been found, the algorithm increases the color count by 15 to expand the search space and retries, whereas if the color count cannot be reduced further and a solution has been found, the process stops. Crucially, the reported runtime accounts only for the search space where a solution was successfully found, excluding any computational effort expended in unsuccessful attempts, such as the initial trials with fewer colors. This ensures that the timing reflects only the effective search for the minimal coloring solution.

B.3.3 VColMIS

VColMIS is based on the Maximum Independent Set (MIS) strategy. In this approach, the vertices in the MIS set are colored with the same color. A subgraph is then created from the remaining uncolored vertices, and the process is repeated by finding a new MIS and assigning another color. This process continues iteratively until all vertices in the graph are colored. For the MIS part, we train the RL-based model by Ahn et al. (2020) on the same dataset used to train our model using the hyperparameters as suggested by the authors. This model takes a graph as input and outputs its maximum independent set.

B.3.4 Gurobi 12 Solver

Given a color set C and graph $G = (V, W)$, we use the Gurobi 12 Optimizer (Gurobi Optimization, 2024) to solve the following integer linear programming model for the vertex coloring problem.

$$\begin{aligned}
\text{Minimize} \quad & \sum_{c=1}^{|C|} z_c \\
\text{subject to} \quad & x_{v,c} + x_{u,c} \leq z_c, \quad \forall (u,v) \in W, \quad \forall c \in C \\
& \sum_{c=1}^{|C|} x_{v,c} = 1, \quad \forall v \in V \\
& x_{v,c} \in \{0,1\}, \quad z_c \in \{0,1\}, \quad \forall v \in V, \quad \forall c \in C
\end{aligned}$$

Here, $x_{v,c}$ is a binary variable that indicates whether vertex v is assigned color c, and z_c is a binary variable that indicates whether color c is used in the solution.

The algorithm starts with an initial color set size of 15 and iteratively increases the size by 15 if the solver fails to find a feasible solution. The reported runtime includes only the computational effort spent in the search space where a solution is successfully found, excluding any time spent on earlier, unsuccessful attempts with smaller color sets.

B.3.5 FastColor

FastColor (Lin et al., 2017) is a reduction-based method for solving the VCP. It is based on the concept of a bounded independent set. The algorithm recursively removes an l-degree bounded independent set from the graph, where l is a lower bound on the chromatic number. A key property of this approach is that any optimal coloring of the reduced graph can be extended to an optimal coloring of the original graph by coloring the removed vertices using the construction rule proposed by the authors. We implemented the algorithm in Python based on the pseudocode provided in the paper. While the results reported in the original paper are based on a C++ implementation, we chose to use Python to ensure consistency and fair comparison with our other baselines and models, which are all implemented in Python.

B.3.6 ReLCol

ReLCol (Watkins et al., 2023) is an RL model that leverages Deep Q-Networks (DQN) and Graph Neural Networks (GNN) to solve the VCP. The model selects the next vertex to be colored and searches for the smallest valid color that can be assigned while ensuring proper coloring. This process continues iteratively until all vertices are colored.

B.4 Cutoff timer tuning of baselines

In this section, we focus on tuning the cutoff time for the baseline algorithms. Among them, FastColor and Gurobi allow setting a cutoff time.

We begin by running FastColor with a cutoff time equal to VColRL's runtime and find that it underperforms on 10% of the benchmark graphs. On these graphs, VColRL uses an average of 5.33 colors, while FastColor uses 8.83 colors, which is approximately 1.65× more, or 66% additional colors. To determine how much time FastColor requires to match VColRL's performance, we gradually increase its cutoff time on each graph using multiples of VColRL's runtime (e.g., 5×, 10×, and so on) until it achieves comparable color usage. We observe that, depending on the graph, FastColor requires between 5× and 160× more time than VColRL. This highlights the significant runtime advantage of our approach.

Specific results are provided in Table 7: the graphs `ash608GPIA`, `ash958GPI`, `le450_5c`, `le450_5d`, `le450_5b`, and `le450_5a` take 0.20, 0.34, 3.50, 6.78, 14.40, and 32.41 seconds respectively to match VColRL's color usage. Since FastColor is a search-based algorithm, these times may vary slightly across runs. Based on these observations, we could select 30 seconds as a fixed cutoff time for FastColor if the goal is to match VColRL's performance. However, we use a 60-second cutoff in our experiments, since FastColor achieves slightly better coloring on some graphs at this setting. Notably, this 60-second cutoff is consistent with the original FastColor paper. Even with a 30-second cutoff, VColRL remains faster.

For Gurobi, when the cutoff time is set equal to VColRL's runtime, it underperforms on approximately 95% of the benchmark graphs. On these graphs, VColRL uses an average of 5.53 colors, while Gurobi uses 14.70, which is about 2.65× more, or 165% additional colors. Even with a 60-second cutoff, Gurobi still underperforms on 31% of the graphs, using 12.27 colors on average compared to VColRL's 5.72, which is approximately 2.12× more, or 114% additional colors. To evaluate Gurobi's best-case performance, we set a high cutoff time of 900 seconds.

Table 7: Tuning the cutoff timer for FastColor (F). The numerical values in the column headers represent multiples of VColRL's runtime.

Graph Type	VColRL	F (1×)	F (5×)	F (10×)	F (20×)	F (40×)	(F 80×)	F (160×)
ash608GPIA	**5, 0.04**	7, 0.04	**5, 0.20**	-	-	-	-	-
ash958GPIA	**5, 0.06**	7, 0.06	**5, 0.34**	-	-	-	-	-
le450_5a	**6, 0.23**	9, 0.23	8, 1.15	8, 2.32	7, 4.63	7, 9.23	7, 18.43	**6, 32.41**
le450_5b	**6, 0.08**	9, 0.08	9, 0.42	9, 0.82	9, 1.62	9, 3.22	7, 6.41	**6, 14.40**
le450_5c	**5, 0.04**	10, 0.04	6, 0.20	6, 0.43	6, 0.80	6, 1.60	**5, 3.50**	-
le450_5d	**5, 0.08**	11, 0.08	7, 0.40	6, 0.85	6, 1.58	6, 3.32	**5, 6.78**	-

C Discussion and Additional Experiments

C.1 Addressing unique challenges faced by VCP in comparison to other combinatorial optimization problems

We outline three key differences that make VCP more challenging for reinforcement learning compared to the MIS problem, and how our approach addresses them:

1. **Larger Action Space and MDP Modeling Complexity:** Unlike MIS, where each vertex has a binary decision (include/exclude), VCP requires assigning one of many color labels per vertex,

leading to a significantly larger action space. This complexity necessitates careful design of the MDP. In our work, we systematically study different combinations of actions, transitions, and reward strategies to identify the configuration that enables stable training and strong performance.

2. **Equivalent Solutions:** Modeling vertex coloring as a classification problem is challenging due to label ambiguity. In MIS, each vertex receives a binary label with a fixed interpretation, indicating whether it belongs to the independent set. In contrast, in vertex coloring, the color labels are arbitrary identifiers without inherent meaning, as long as adjacent vertices receive different labels. For example, consider assigning colors to four vertices using labels from $\{1, 2, 3, 4\}$. The assignments $[1, 2, 3, 1]$ and $[1, 2, 4, 1]$ define the same partitioning of vertices into color classes but use different label values. While structurally equivalent, they appear different to the agent, introducing ambiguity during learning. To mitigate this, we employ a *max-color reward* that penalizes the use of higher-indexed colors, encouraging contiguous and consistent labelings. This helps reduce ambiguity arising from equivalent labelings and provides a more stable learning signal

3. **Stronger Residual Dependencies in Partially Solved Graphs:** In MIS, once a vertex is included, its neighbors are excluded, allowing the agent to proceed cleanly on the remaining subgraph. In VCP, assigning a color restricts neighbor options but does not uniquely determine them. Thus, learning cannot rely on the uncolored subgraph alone, and it must incorporate information from already-colored neighbors. Our framework handles this by designing vertex features that encode neighborhood color usage, enabling informed and context-aware decision making.

C.2 Future Work

A promising direction for future work is to incorporate Imitation Learning (Hussein et al., 2017) to accelerate and stabilize training. Algorithms like Fastcolor generate high-quality solutions that can be used to train models to mimic expert behavior, potentially leading to a lightweight training process. Additionally, the current method relies on a greedy search during inference. A promising extension would be to incorporate the learned policy as a prior within a more sophisticated search framework. For example, Ahn et al. (2020) improves their solution by using the learned output as a starting point and applying a 2-improvement local search algorithm (Feo et al., 1994), which iteratively removes one vertex and adds two others to increase the size of the independent set, continuing until no further improvement is possible. In a similar manner, a search method tailored for the vertex coloring problem can be applied to the solution generated by VColRL, which could further enhance the coloring quality and overall performance.

Table 8: Comparison between the VColRL model trained on the ER graph dataset and a mix of the ER, BA, and WS datasets.

Graph Type	Vertex Range	Trained on ER Dataset	Trained on Mix Dataset
ER	50-100	5.592, 0.36	5.754, 0.22
ER	100-150	6.760, 0.65	6.918, 0.37
ER	150-200	7.528, 0.80	7.526, 0.50
BA	50-100	6.168, 0.46	6.136, 0.28
BA	100-150	6.686, 0.51	6.534, 0.35
BA	150-200	6.898, 0.70	6.888, 0.43
WS	50-100	6.996, 0.54	7.014, 0.29
WS	100-150	7.008, 0.70	7.024, 0.38
WS	150-200	7.014, 0.82	7.082, 0.49

C.3 Training with mixed dataset for testing generalizability

In the numerical experiments conducted so far, we use a model trained exclusively on ER graphs with 50–100 vertices and observe that it performs well across a wide range of graph types and sizes, despite not being explicitly trained on those distributions. To further investigate the impact of training data diversity, we

train a new model on a mixed dataset comprising ER, BA, and WS graphs. We evaluate this model on the same set of test graphs used in Table 6 and compare its performance with that of the model trained solely on ER graphs. The results are presented in Table Table 8. We observe a slight improvement on ER graphs in the 150–200 vertex range and across all BA graph ranges. However, performance on other graph types and sizes declines slightly. Overall, the two models perform comparably, and the observed fluctuations can be attributed to the stochastic nature of the VColRL framework. This experiment strengthens our generalization claim: VColRL trained solely on ER graphs performs on par with a model trained on a more diverse dataset, demonstrating its ability to generalize effectively across a variety of unseen graph distributions without requiring explicit exposure during training.

C.4 Sequential Models

The underlying MDP in VColRL assigns colors to vertices in bulk and performs rollbacks for vertices involved in conflicts. To isolate the impact of bulk versus sequential assignment, we compare VColRL with two sequential variants: one with deferral (SD, Sequential Defer) and one without deferral (SW, Sequential Without-Defer). These sequential models follow a greedy-style approach where, at each step, the actor randomly selects a vertex to update. The results are presented in Table 9. VColRL uses an average of 5.60 colors, whereas SD and SW use 6.55 and 7.06 colors, respectively. In terms of execution time, VColRL is $2 - 500\times$ faster than both SD and SW due to its bulk assignment strategy. Between the two sequential models, SD is typically $1 - 3\times$ faster.

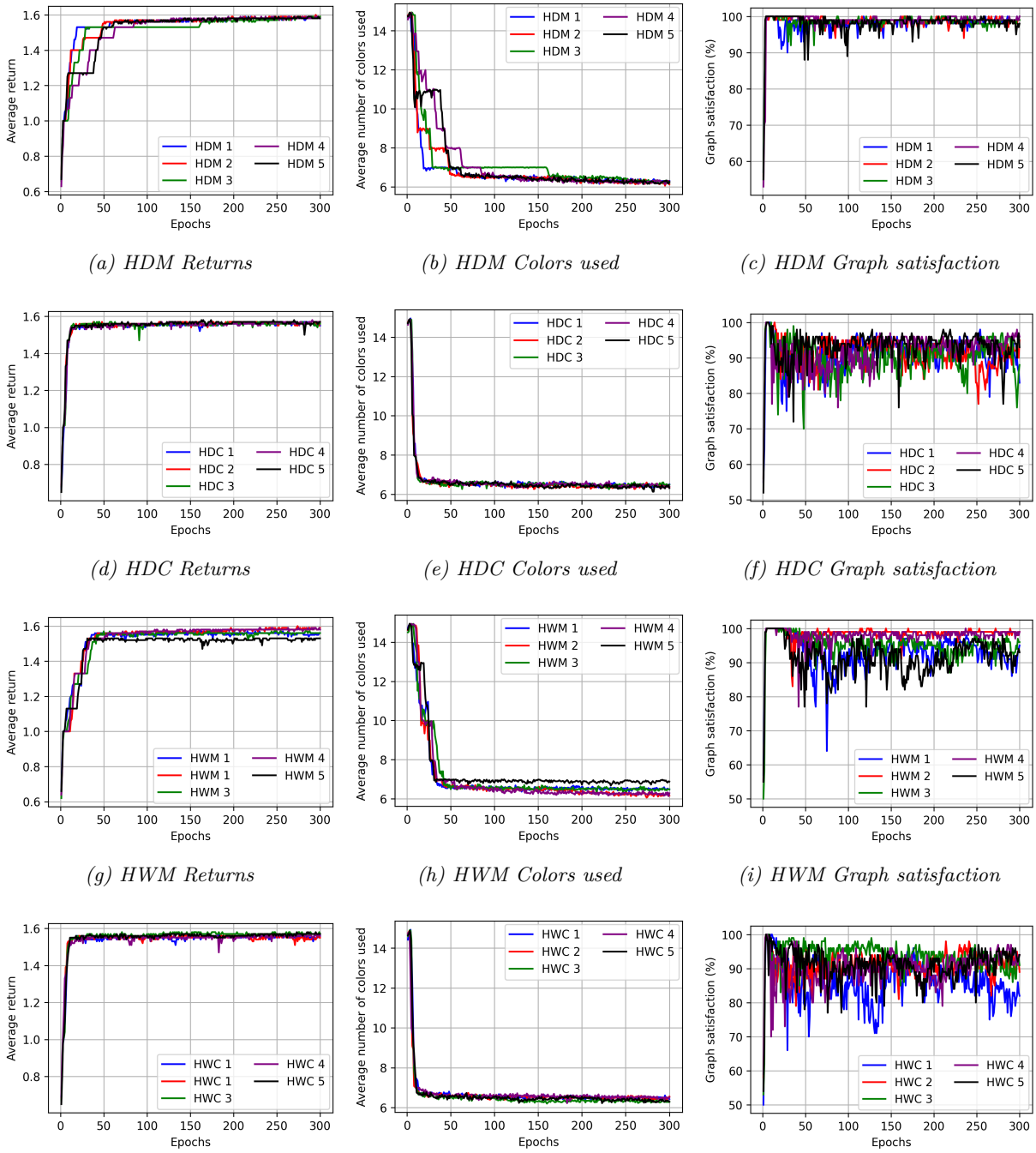
Table 9: Comparison between VColRL and sequential models with defer (SD) and without-defer actions (SW) in terms of best solutions on DIMACS and COLOR02 benchmarks. The best performing models are boldfaced.

Graph Type	SD	SW	VCoLRL	Graph Type	SD	SW	VCoLRL
ash331GPIA	6, 4.08	7, 7.74	**5**, 0.29	3-Insertions_5	**6**, 9.69	8, 22.22	**6**, 0.16
ash608GPIA	6, 13.69	7, 31.72	**5**, 0.04	4-Insertions_3	**4**, 0.38	**4**, 1.74	**4**, 0.01
ash958GPIA	8, 29.24	7, 27.30	**5**, 0.06	4-Insertions_4	**5**, 3.66	6, 3.67	**5**, 0.02
1-FullIns_3	**4**, 0.10	**4**, 0.29	**4**, 0.01	le450_5a	14, 10.33	13, 7.14	**6**, 0.23
1-FullIns_4	**5**, 0.36	**5**, 0.71	**5**, 0.03	le450_5b	13, 10.68	12, 8.79	**6**, 0.08
1-FullIns_5	**6**, 1.67	**6**, 4.03	**6**, 0.10	le450_5c	9, 6.24	12, 8.43	**5**, 0.04
2-FullIns_3	**5**, 0.23	**5**, 0.15	**5**, 0.04	le450_5d	10, 8.15	10, 5.37	**5**, 0.08
2-FullIns_4	**6**, 0.94	**6**, 1.69	**6**, 0.49	mug88_1	**4**, 0.27	6, 0.89	**4**, 0.03
2-FullIns_5	**7**, 3.93	9, 14.87	**7**, 0.06	mug88_25	**4**, 0.59	5, 0.93	**4**, 0.03
3-FullIns_3	**6**, 0.35	**6**, 0.77	**6**, 0.10	mug100_1	**4**, 0.72	6, 1.04	**4**, 0.04
3-FullIns_4	**7**, 4.58	**7**, 4.50	**7**, 0.23	mug100_25	**4**, 0.69	6, 1.04	**4**, 0.02
4-FullIns_3	**7**, 2.44	**7**, 1.09	**7**, 0.17	myciel3	**4**, 0.13	**4**, 0.10	**4**, 0.009
4-FullIns_4	**8**, 8.34	10, 22.25	**8**, 0.65	myciel4	**5**, 0.16	**5**, 0.22	**5**, 0.02
5-FullIns_3	9, 1.66	9, 1.68	**8**, 0.29	myciel5	**6**, 0.39	**6**, 0.29	**6**, 0.07
1-Insertions_4	**5**, 0.20	**5**, 0.66	**5**, 0.02	myciel6	**7**, 2.17	**7**, 1.04	**7**, 0.06
1-Insertions_5	**6**, 0.89	7, 2.17	**6**, 0.10	myciel7	**8**, 4.66	11, 3.67	**8**, 0.32
1-Insertions_6	8, 7.14	9, 8.93	**7**, 0.16	queen5_5	**5**, 0.29	**5**, 0.07	**5**, 0.03
2-Insertions_3	**4**, 0.10	**4**, 0.34	**4**, 0.01	queen6_6	8, 0.48	9, 0.42	**7**, 0.21
2-Insertions_4	**5**, 1.01	6, 1.50	**5**, 0.02	queen7_7	10, 0.92	9. 0.60	**7**, 0.27
2-Insertions_5	**6**, 3.84	7, 7.89	**6**, 0.12	DSJC125.1	8, 1.35	7, 1.26	**6**, 0.10
3-Insertions_3	**4**, 0.33	**4**, 0.54	**4**, 0.01	will199GPIA	11, 10.56	11, 9.02	**7**, 0.23
3-Insertions_4	**5**, 1.99	**5**, 2.54	**5**, 0.03				

C.5 Multiple Runs of MDP configurations

In Section 4.1, we compared various MDP configurations and concluded that HDM outperforms all other configurations. To further verify the claim, we train each variant five times and report the performance in Figure 7 and Figure 8. While all variants achieve similar average returns and color usage, the HDM configuration consistently yields higher graph satisfaction and more stable policies across runs. This is

especially important in the vertex coloring problem, where ensuring that all vertices are properly colored while also minimizing color usage is paramount. However, coloring all vertices takes precedence over reducing the number of colors used. As discussed in Section 3.1, our reward design reflects this by assigning greater weight to vertex satisfaction. Notably, the other variants, despite achieving comparable average returns and color usage to HDM, leave a few vertices uncolored when it comes to graph satisfaction, which is an undesirable outcome. These results underscore the effectiveness of HDM in producing robust policies that better meet the core objectives of the task.



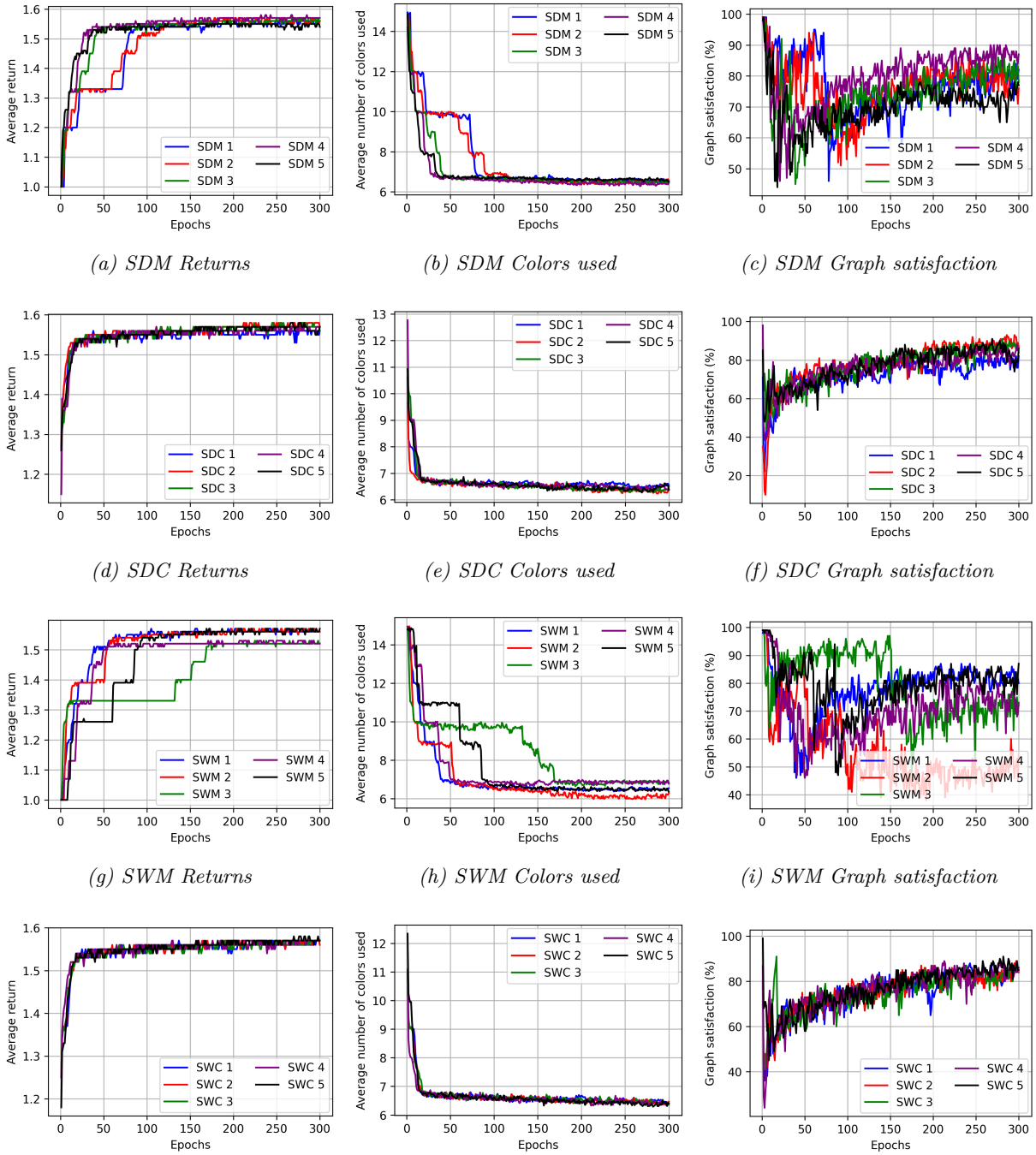
(a) HDM Returns *(b) HDM Colors used* *(c) HDM Graph satisfaction*

(d) HDC Returns *(e) HDC Colors used* *(f) HDC Graph satisfaction*

(g) HWM Returns *(h) HWM Colors used* *(i) HWM Graph satisfaction*

(j) HWC Returns *(k) HWC Colors used* *(l) HWC Graph satisfaction*

Figure 7: ***Performance of VColRL across different configurations of hard rollback.***



(a) SDM Returns *(b) SDM Colors used* *(c) SDM Graph satisfaction*

(d) SDC Returns *(e) SDC Colors used* *(f) SDC Graph satisfaction*

(g) SWM Returns *(h) SWM Colors used* *(i) SWM Graph satisfaction*

(j) SWC Returns *(k) SWC Colors used* *(l) SWC Graph satisfaction*

Figure 8: ***Performance of VColRL across different configurations of soft rollback.***