Stop thinking hard: Proximal Policy Optimization is everything you need for Combinatorial Optimization

Nguyen Ngoc Tu[*]
Department of Computer Science
International University
Ho Chi Minh, Vietnam
`ITCSIU23043@student.hcmiu.edu.vn`

Nguyen Quang Truc[*]
Department of Computer Science
International University
Ho Chi Minh, Vietnam
`ITCSIU23041@student.hcmiu.edu.vn`

Abstract

Combinatorial optimization problems, which are categorized as NP-Hard, arise frequently in real-world applications such as logistics, scheduling, and resource allocation. While exact algorithms guarantee optimal solutions, their exponential computational complexity renders them impractical for large-scale instances. Approximation algorithms offer a viable alternative by providing near-optimal solutions within reasonable time bounds; however, they often require problem-specific heuristics and domain expertise. In this paper, we investigate the application of deep reinforcement learning to address NP-Hard combinatorial optimization problems. Specifically, we employ Proximal Policy Optimization (PPO), a state-of-the-art policy gradient algorithm, to learn generalizable solution strategies. Our approach formulates each combinatorial problem as a sequential decision-making process, where an agent learns to construct solutions step-by-step through interaction with a problem-specific environment. We evaluate our method on three classical NP-Hard problems: the 0/1 Knapsack Problem, the Travelling Salesman Problem (TSP), and the Graph Coloring Problem. To enhance training stability and solution quality, we incorporate curriculum learning, progressively training the agent on problem instances of increasing difficulty. Experimental results demonstrate that our PPO-based approach can effectively learn competitive solution policies, achieving near-optimal performance while maintaining low inference time, thus offering a promising data-driven alternative to traditional algorithmic approaches for combinatorial optimization. Our implementation is available at https://github.com/nguyentuss/IT159IU-Artificial-Intelligence.

1 Introduction

Combinatorial optimization (CO) problems are fundamental to computer science and operations research, with widespread applications in logistics, scheduling, resource allocation, network design and artificial intelligence [15].
Traditional approaches to solving NP-Hard problems fall into two categories: *exact methods* and *approximation methods*. Exact algorithms, such as branch-and-bound, dynamic programming, and integer linear programming, guarantee optimal solutions but suffer from prohibitive computational complexity for large-scale instances [17]. Approximation algorithms and heuristics, including greedy methods, local search, and metaheuristics (e.g., genetic algorithms [1], simulated annealing [9]), provide near-optimal solutions in reasonable time but often require extensive domain expertise and problem-specific tuning [8].
In recent years, deep reinforcement learning (DRL) has emerged as a promising paradigm for tackling combinatorial optimization problems [7]. Unlike traditional methods, DRL agents learn

[*]Equal contribution

solution strategies directly from data through trial-and-error interaction with the problem environment, potentially discovering novel heuristics without explicit human guidance. Among various DRL algorithms, **Proximal Policy Optimization (PPO)** has gained significant attention due to its stability, sample efficiency, and ease of implementation [19]. PPO belongs to the family of policy gradient methods and employs a clipped surrogate objective to prevent destructive policy updates, making it well-suited for complex sequential decision-making tasks.
This study investigates the application of PPO to solve three classical NP-Hard combinatorial optimization problems: the **0/1 Knapsack Problem**, the **Travelling Salesman Problem (TSP)**, and the **Graph Coloring Problem**. We formulate each problem as a Markov Decision Process (MDP) and train neural network-based policies to construct solutions through sequential decision-making. Additionally, we employ **curriculum learning** to progressively train agents on problem instances of increasing difficulty, enhancing training stability and solution quality [5], also we will provide the empirical experiment using PPO that faster than *exact method* but still nearly optimal solution.

2 Related Work

The paradigm of Neural Combinatorial Optimization (NCO) has shifted from early supervised approaches [23] to Deep Reinforcement Learning (DRL), driven by the desire to solve NP-Hard problems without labeled optima. Although foundational works established the viability of constructive heuristics using Policy Gradients [4] and Attention mechanisms [13], the recent literature reveals a trend toward increasingly specialized and computationally intensive architectures.

2.1 Architectural Complexity vs. Generalization

Recent literature in Neural Combinatorial Optimization often tends toward highly specialized architectures to squeeze out marginal performance gains. We contrast these complex approaches with our streamlined PPO implementation in three core domains.

0/1 Knapsack Problem. In the domain of resource allocation, recent work such as Hubbs et al. [11] have demonstrated PPO's utility in stochastic environments, yet they often struggle to compete with classical heuristics on large-scale deterministic instances without auxiliary search mechanisms. Other deep learning approaches frequently rely on hybridizing RL with beam search or greedy decoders to ensure capacity constraints are met. In contrast, our work investigates the efficacy of a "vanilla" PPO agent using a lightweight Multi-Layer Perceptron (MLP) architecture, relying strictly on validity masking rather than complex decoder strategies to maintain solution feasibility.

Traveling Salesman Problem (TSP). The TSP serves as the primary testbed for architectural innovation in NCO. State-of-the-art methods, such as the Attention Model (AM) [13] and recent improvements by Yimer et al. [25], often introduce intricate components like adaptive temperature scheduling, specialized normalization layers, or multi-agent coordination [21] to minimize optimality gaps. These additions significantly increase hyperparameter sensitivity. We diverge from this trend by employing a standard Transformer encoder-decoder backbone without problem-specific architectural engineering, testing the hypothesis that curriculum learning alone is sufficient to bridge the gap to optimality.

Graph Coloring Problem. The strict constraints of graph coloring traditionally drive researchers toward heavy architectural engineering. Approaches like VColRL [2] integrate PPO with complex Message-Passing Neural Networks (MPNNs) and "rollback" mechanisms to recover from invalid coloring states. While effective, such reliance on deep iterative message passing increases computational overhead and training instability. Our approach simplifies this by utilizing a standard Graph Neural Network (GNN)[27] combined with a rigorous valid-action mask, eliminating the need for backtracking dynamics or soft-penalty shaping while ensuring all generated colorings are legally valid by construction.

2.2 MDP Formulations and Constraint Handling

Neural Combinatorial Optimization (NCO) approaches generally fall into two paradigms: *improvement heuristics*, which iteratively refine a complete solution, or *constructive heuristics*, which build solutions sequentially [13]. A critical bifurcation in this literature lies in the handling of hard constraints. To ensure solution validity (e.g., satisfying knapsack capacity or graph coloring rules), prevalent methods often resort to "soft" penalty shaping—adding negative terms to the reward function—or complex transition dynamics like "rollback" to recover from invalid states [2].

However, these strategies introduce significant optimization challenges. Relying on penalty shaping confounds the learning objective, forcing the agent to simultaneously learn *feasibility* (what is valid) and *optimality* (what is good), often leading to slow convergence or collapse into safe but suboptimal local minima. Similarly, rollback mechanisms increase computational overhead and complicate the underlying Markov process. We address these limitations by strictly enforcing constraints through *valid action masking* within the policy logits. By mechanically setting the probabilities of invalid transitions to zero, we guarantee that the agent only samples valid states by construction. This effectively decouples feasibility from optimality, pruning the exploration space, and allowing the policy to focus exclusively on maximizing the objective function.

2.3 Curriculum Learning in NCO

Scaling DRL agents to large combinatorial instances remains a persistent challenge due to the sparsity of rewards and the combinatorial explosion of the state space. The dominant trend in recent literature addresses this scaling problem via architectural innovation, introducing heavy inductive biases—such as specialized Attention mechanisms or deep Graph Neural Networks—intended to generalize across problem sizes.

Despite these architectural advances, training remains often unstable. Expecting an agent to learn complex, long-horizon strategies from scratch on difficult instances (e.g., high-chromatic number graphs) frequently results in high-variance gradients and a failure to escape initial random policies. We posit that this focus on architectural engineering overlooks the importance of the data distribution itself. Instead of complicating the model, we stabilize the optimization process via *Curriculum Learning* [5]. By training the agent on instances of progressively increasing difficulty, we smooth the optimization landscape, allowing the agent to transfer learned primitives from simple to complex tasks. This data-centric approach enables a standard, lightweight PPO architecture to achieve performance parity with more complex models that lack such a curriculum.

3 A Background from Policy Gradient to PPO

Reinforcement learning aims to train an agent to make sequential decisions that maximize cumulative rewards. Policy gradient methods achieve this by directly optimizing a parameterized policy—a probability distribution over actions given a state. The agent collects experience by interacting with the environment, then uses this experience to update its policy in a direction that increases expected rewards (see Appendix B for the formal policy gradient theorem).
A key challenge with policy gradient methods is training stability. Large policy updates can lead to performance collapse, where the agent's behavior degrades catastrophically. Trust Region Policy Optimization (TRPO) [18] addressed this by constraining how much the policy can change in each update, but its implementation requires complex second-order optimization (formal definition are provided in Appendix C.1).
PPO offers an elegant solution: instead of a hard constraint, it uses a *clipped surrogate objective* that naturally limits policy changes. The core idea is simple—when the new policy deviates too far from the old policy (measured by the ratio of action probabilities), the objective function is clipped to remove the incentive for further deviation. This allows the use of standard first-order optimization methods like Adam while maintaining training stability.

3.1 Key Components

PPO combines three essential components:

1. **Clipped Policy Objective.** The algorithm computes the ratio between the new and old policy probabilities for each action. When this ratio exceeds a threshold (typically 1 ± 0.2), the objective is clipped, preventing excessively large updates. This mechanism is particularly important when an action has positive advantage (the action is better than average), as it prevents the policy from becoming overconfident about that action. The formal definition and mathematical are provided in Appendix C.2.
2. **Value Function.** A separate neural network estimates the expected future reward from each state. This value function serves two purposes: it provides a baseline to reduce variance in policy gradient estimates, and it is trained alongside the policy to improve predictions over time. The formal definition of value functions and the Bellman equations are provided in Appendix A.3.
3. **Generalized Advantage Estimation (GAE).** To determine whether an action was good or bad, PPO uses the advantage function—the difference between the value of taking a specific action and the average value of all actions. GAE provides a practical way to estimate advantages by combining information from multiple future time steps, balancing the trade-off between bias and variance. The mathematical formulation of GAE is detailed in Appendix D.

3.2 Why PPO for Combinatorial Optimization?

PPO is particularly well-suited for combinatorial optimization problems for several reasons:

- **Discrete Action Spaces.** Combinatorial problems naturally involve discrete decisions (e.g., accept or reject an item, select a city, choose a color). PPO handles discrete action spaces effectively through categorical policy distributions.
- **Sequential Decision Making.** Each problem can be decomposed into a sequence of decisions, forming a natural MDP structure (see Appendix A for MDP preliminaries). The agent learns to construct solutions step-by-step.
- **Constraint Handling.** PPO can incorporate feasibility constraints through action masking, ensuring that the agent only considers valid actions at each step.
- **Generalization.** Once trained, PPO policies can generalize to new problem instances, providing fast inference compared to traditional optimization algorithms that must solve each instance from scratch.

The complete mathematical derivation of PPO, including the clipped objective function, value function loss, and entropy regularization, is provided in Appendix C.

4 Methodology

In this section, we presents the application of Proximal Policy Optimization (PPO) to solve the three NP-Hard combinatorial optimization problems. For each problem, we describe the MDP formulation, neural network architecture, and training procedure.

4.1 0/1 Knapsack Problem

4.1.1. Problem Statement

Given a set of n items, each with a non-negative weight w_i and a value v_i, and a knapsack with maximum capacity $C_{\max}$, the objective is to select a binary selection vector $\mathbf{x}^* \in \{0,1\}^N$ that maximizes the total value without exceeding the capacity constraint:

$$\mathbf{x}^* = \arg \max_{\mathbf{x}\in\{0,1\}^N} \sum_{i=1}^{N} v_i x_i \quad \text{subject to} \quad \sum_{i=1}^{N} w_i x_i \leq C_{\max}$$

4.1.2. MDP Formulation

We formulate the Knapsack problem as a sequential decision process where the agent considers each item in order and decides whether to include it. At time step $t \in \{0, 1, \ldots, N-1\}$, the state is defined as:

$$s_t = (\mathbf{v}, \mathbf{w}, \mathbf{x}_t, W_t, V_t, C_{\max})$$

where $\mathbf{v} \in \mathbb{R}^N_{>0}$ is the item value vector, $\mathbf{w} \in \mathbb{R}^N_{>0}$ is the item weight vector, $\mathbf{x}_t \in \{-1, 0, 1\}^N$ is the decision vector (1: selected, -1: rejected, 0: undecided), $W_t = \sum_{i:x_t^{(i)}=1} w_i$ is the accumulated weight, and $V_t = \sum_{i:x_t^{(i)}=1} v_i$ is the accumulated value.

4.1.3. Action space

For sequential item processing, the action space is binary:

$$\mathcal{A}(s_t) = \{0, 1\}$$

where $a_t = 1$ indicates selecting item t and $a_t = 0$ indicates rejecting item t. To ensure feasibility, we apply a mask that forces $a_t = 0$ when $W_t + w_t > C_{\max}$.

4.1.4. Transition Function

The transition function deterministically updates the state based on the action:

$$x_{t+1}^{(i)} = \begin{cases} 2a_t - 1 & \text{if } i = t \\ x_t^{(i)} & \text{otherwise} \end{cases}$$

$$W_{t+1} = W_t + a_t \cdot w_t, \quad V_{t+1} = V_t + a_t \cdot v_t$$

The episode terminates after all N items have been processed. The value $2a_t - 1$ conditioned $i = t$ that map from $a_t \in \{0, 1\}$ into $x_{t+1}^{(i)} \in \{-1, 1\}$.

4.1.5. Reward Function

The reward function provides immediate feedback based on the value-to-weight trade-off:

$$r_t = a_t \cdot \left(\alpha \cdot \frac{v_t}{V_{\text{sum}}} - \beta \cdot \frac{O_t}{C_{\max}} \right)$$

where $V_{\text{sum}} = \sum_{i=1}^N v_i$ is the total value of all items, $O_t = \max(0, W_t + w_t - C_{\max})$ is the capacity overflow, and $\alpha, \beta > 0$ are weighting coefficients. The first term rewards selecting valuable items, while the second term penalizes capacity violations.

4.2 Travelling Salesman Problem (TSP)

4.2.1. Problem Definition

The Traveling Salesman Problem (TSP) is a classical combinatorial optimization problem. Given a set of N cities with coordinates $\mathbf{P} = \{p_1, p_2, \ldots, p_N\}$ where each $p_i \in \mathbb{R}^2$, the objective is to find a permutation $\tau = (\tau_1, \tau_2, \ldots, \tau_N)$ of cities that minimizes the total tour length:

$$L(\tau) = \sum_{i=1}^{N-1} d(p_{\tau_i}, p_{\tau_{i+1}}) + d(p_{\tau_N}, p_{\tau_1})$$

where $d(p_i, p_j) = \|p_i - p_j\|_2$ denotes the Euclidean distance between cities i and j. The tour must visit each city exactly once and return to the starting city.

4.2.2. MDP Formulation

We formulate the TSP as a finite-horizon MDP where the agent constructs a tour by sequentially selecting unvisited cities. At time step $t \in \{0, 1, \ldots, N-1\}$, the state is defined as:

$$s_t = (\mathbf{P}, \mathbf{v}_t, c_t, c_0)$$

where $\mathbf{P} \in \mathbb{R}^{N \times 2}$ is the city coordinate matrix, $\mathbf{v}_t \in \{0,1\}^N$ is the visitation indicator vector with $v_t^{(i)} = 1$ if city i has been visited by time t, c_t is the index of the current city, and c_0 is the index of the starting city (depot).

4.2.3. Action Space

The action space at state s_t consists of all unvisited cities:

$$\mathcal{A}(s_t) = \{i \in \{1, \ldots, N\} : v_t^{(i)} = 0\}$$

At each step, the agent selects an action $a_t \in \mathcal{A}(s_t)$ representing the next city to visit. The action space shrinks as more cities are visited, with exactly one valid action remaining at the final step.

4.2.4. Transition Function

The transition function is deterministic. Given state s_t and action a_t, the next state s_{t+1} is computed as:

$$v_{t+1}^{(i)} = \begin{cases} 1 & \text{if } i = a_t \\ v_t^{(i)} & \text{otherwise} \end{cases}, \quad c_{t+1} = a_t$$

The episode terminates when all cities have been visited, i.e., when $\sum_{i=1}^{N} v_t^{(i)} = N$.

4.2.5. Reward Function

We employ a dense reward function to provide step-wise feedback. At each step t, the reward is defined as:

$$r_t = -\frac{d(p_{c_t}, p_{a_t})}{D_{\text{avg}}}$$

where $D_{\text{avg}} = \frac{2}{N(N-1)} \sum_{i=1}^{N} \sum_{j=i+1}^{N} d(p_i, p_j)$ is the average pairwise distance. At the terminal step $t = N-1$, the reward includes the return distance to the depot:

$$r_{N-1} = -\frac{d(p_{c_{N-1}}, p_{a_{N-1}}) + d(p_{a_{N-1}}, p_{c_0})}{D_{\text{avg}}}$$

The normalization by D_{avg} ensures scale-invariance across problem instances.

4.3 Graph Coloring Problem

4.3.1. Problem Definition

The Graph Coloring Problem requires assigning colors to vertices of a graph such that no adjacent vertices share the same color. Given an undirected graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, N\}$ and edge set E, and a set of K colors $\mathcal{C} = \{1, 2, \ldots, K\}$, the objective is to find a color assignment $\mathbf{c} : V \to \mathcal{C}$ satisfying:

$$\forall (i, j) \in E : c_i \neq c_j$$

The chromatic number χ(G) denotes the minimum number of colors required for a valid coloring.

4.3.2. MDP Formulation

We formulate graph coloring as a sequential vertex coloring process. At time step $t \in \{0, 1, \ldots, N-1\}$, the state is defined as:

$$s_t = (\mathbf{A}, \mathbf{c}_t)$$

where $\mathbf{A} \in \{0,1\}^{N \times N}$ is the adjacency matrix with $A_{ij} = 1$ if $(i,j) \in E$, and $\mathbf{c}_t \in \{0, 1, \ldots, K\}^N$ is the color assignment vector where $c_t^{(i)} = 0$ indicates that vertex i is uncolored.

4.3.3. Action Space

For coloring vertex t, the full action space is:

$$\mathcal{A}(s_t) = \{1, 2, \ldots, K\}$$

The valid action space excludes colors already used by neighbors:

$$\mathcal{A}_{\text{valid}}(s_t) = \{k \in \mathcal{C} : \forall j \in \mathcal{N}(t), c_t^{(j)} \neq k\}$$

where $\mathcal{N}(t) = \{j \in V : A_{tj} = 1\}$ is the neighborhood of vertex t.

4.3.4. Transition Function

The transition function deterministically updates the color assignment:

$$c_{t+1}^{(i)} = \begin{cases} a_t & \text{if } i = t \\ c_t^{(i)} & \text{otherwise} \end{cases}$$

The episode terminates when all N vertices have been colored.

4.3.5. Reward Function

The reward function penalizes conflicts and encourages using fewer colors:

$$r_t = -\alpha \cdot \Delta C_t - \beta \cdot \mathbb{I}[a_t \notin \mathcal{C}_{\text{used}}(s_t)]$$

where $\Delta C_t = |\{j \in \mathcal{N}(t) : c_t^{(j)} = a_t\}|$ is the number of new conflicts created, $\mathcal{C}_{\text{used}}(s_t) = \{c_t^{(i)} : c_t^{(i)} \neq 0\}$ is the set of colors already used, and $\alpha, \beta > 0$ are weighting coefficients. The first term penalizes conflicts, while the second term discourages introducing new colors.

5 Implementation and Results

5.1 Implementation

In this section, we describes the implementation details of the PPO-based solution for the three combinatorial optimization problems. Please refer to Appendix E for the implementation details.

5.1.1. TSP Policy Network

For the Traveling Salesman Problem, we employ a Transformer-based encoder-decoder architecture [22]. The encoder processes city coordinates through multiple self-attention layers to capture global dependencies between all cities. Each city i is first embedded as:

$$\mathbf{h}_i^{(0)} = \mathbf{W}_{\text{emb}} \cdot p_i + \mathbf{b}_{\text{emb}}$$

where $p_i \in \mathbb{R}^2$ represents the city coordinates. The encoder applies L Transformer layers with multi-head attention:

$$\mathbf{H}^{(l+1)} = \text{LayerNorm}(\mathbf{H}^{(l)} + \text{MHA}(\mathbf{H}^{(l)}))$$

The decoder computes a context vector by concatenating the mean node embedding, current city embedding, and depot embedding:

$$\mathbf{h}_{\text{ctx}} = [\bar{\mathbf{h}} \| \mathbf{h}_{c_t}^{(L)} \| \mathbf{h}_{c_0}^{(L)}]$$

Action probabilities are computed using attention between the context and city embeddings, with masking applied to visited cities:

$$\pi_\theta(a_t = i \mid s_t) = \frac{\exp(u_i)}{\sum_{j: v_t^{(j)}=0} \exp(u_j)}$$

where $u_i = -\infty$ for visited cities.

5.1.2. Knapsack Policy Network

For the Knapsack Problem, we use a Multi-Layer Perceptron (MLP) architecture. Each item is represented by normalized features:

$$\mathbf{f}_i = \left[\frac{v_i}{\bar{v}}, \frac{w_i}{\bar{w}}, \frac{v_i / w_i}{\overline{(v/w)}}, \frac{w_i}{C_{\max}} \right]$$

Context features capture the current state:

$$\mathbf{f}_{\text{ctx}} = \left[\frac{W_t}{C_{\max}}, \frac{C_{\max} - W_t}{C_{\max}}, \frac{V_t}{V_{\text{sum}}}, \frac{t}{N} \right]$$

The policy network outputs the probability of selecting the current item:

$$\pi_\theta(a_t = 1 \mid s_t) = \sigma(\text{MLP}(\mathbf{f}_t \| \mathbf{f}_{\text{ctx}}))$$

A feasibility mask is applied to ensure the agent cannot select items that would exceed capacity.

5.1.3. Graph Coloring Policy Network

For Graph Coloring, we employ a Graph Neural Network (GNN) with message passing [27]. Node features include degree, current color (one-hot encoded), and blocked colors from neighbors:

$$\mathbf{f}_i = \left[\frac{\deg(i)}{\Delta(G)}, \mathbf{e}_{c_t^{(i)}}, \mathbb{I}[i = t], \mathbf{b}_i \right]$$

Message passing for L layers:

$$\begin{aligned} \mathbf{m}_i^{(l)} &= \text{AGG}_{j \in \mathcal{N}(i)}(\text{MLP}_{\text{msg}}(\mathbf{h}_j^{(l)})) \\ \mathbf{h}_i^{(l+1)} &= \text{LayerNorm}(\mathbf{h}_i^{(l)} + \text{MLP}_{\text{upd}}([\mathbf{h}_i^{(l)} \| \mathbf{m}_i^{(l)}])) \end{aligned}$$

Each color k is embedded, and action probabilities are computed via attention:

$$\pi_\theta(a_t = k \mid s_t) = \frac{\exp(u_k)}{\sum_{k'=1}^{K} \exp(u_{k'})}$$

with $u_k = -\infty$ for colors used by neighbors.

5.2 Curriculum Learning

To enhance training stability and convergence speed, we employ a curriculum learning strategy for the TSP and Graph Coloring tasks. This approach mimics human learning by exposing the model to progressively harder problem instances, allowing it to learn simple heuristics before tackling complex structural constraints. The training process is organized into distinct stages of increasing difficulty:

TSP. The complexity of the TSP is scaled by the number of cities in the tour. Training commences with trivial instances containing 5 cities, allowing the agent to easily learn valid tour construction. As the model stabilizes, the problem size is incrementally increased up to 20 cities. This gradual expansion prevents the agent from converging to poor local optima early in the training process.

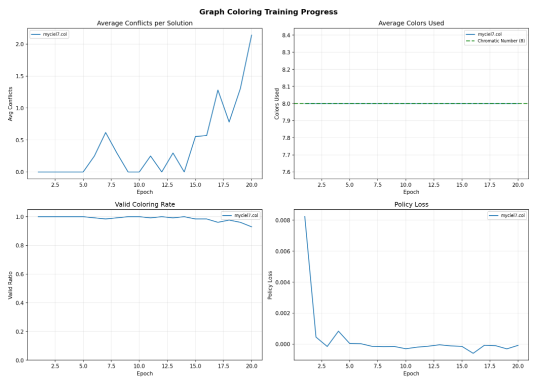

Figure 1: Graph coloring in myciel7 data

Knapsack. We opted to exclude curriculum learning for the Knapsack problem. Preliminary experiments indicated that the curriculum approach was counterproductive for this specific task; the loss function exhibited rapid vanishing, leading to premature convergence where the model failed to learn meaningful policies. Consequently, the Knapsack model is trained directly on the target distribution without staged difficulty.

Graph Coloring. The difficulty for Graph Coloring is modulated using the Mycielski construction method [26]. The curriculum progresses through a sequence of Mycielski graphs with increasing chromatic numbers, starting from simple graphs requiring only 3 colors and advancing to complex instances requiring up to 8 colors. This ensures the model learns to handle increasingly dense constraints and higher-order dependencies.

At each stage, the model is initialized with weights from the previous stage and trained for a fixed number of epochs before advancing to the next difficulty level.

5.3 Experiments

5.3.1. Experimental setup

Traveling Salesman Problem. We use a collection of Euclidean TSP instances with varying sizes. Each instance consists of 2D city coordinates sampled from different distributions. Table II summarizes the TSP instances.
0-1 Knapsack Problem. We use benchmark instances from the classical knapsack problem literature. Each instance specifies item weights, profits, and knapsack capacity. Table I summarizes the knapsack instances.
Graph Coloring. We use Mycielski graphs [26], a family of triangle-free graphs with increasing chromatic numbers. These graphs are challenging because they have clique number 2 but require progressively more colors. Table III summarizes the graph coloring instances.
All the optimal solution are compute using the *exact method*. Please refer to Table IV in the Appendix for the hyperparameter settings.

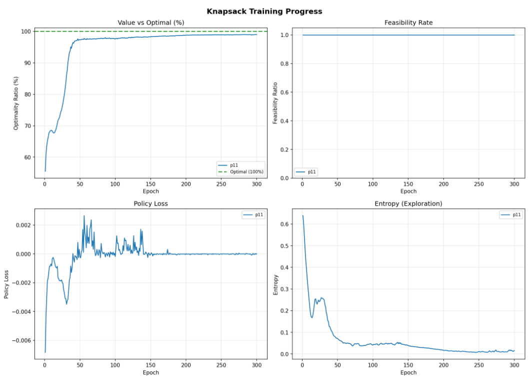

Figure 2: 0-1 Knapsack in p11 data

5.3.2. Experimental Results

We evaluated the performance of our model on three combinatorial optimization tasks—Graph Coloring, 0-1 Knapsack, and the Traveling Salesman Problem (TSP)—using an Apple M4 Pro chip. We present the training curves for the largest data point in each task in Figures 1, 2, and 3. Additional plots and details are provided in Appendix F.

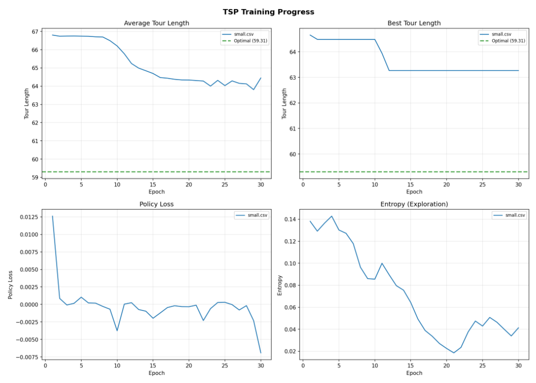

Figure 3: TSP in small data

Graph Coloring and 0-1 Knapsack. For both the Graph Coloring (Figure 1) and 0-1 Knapsack (Figure 2) tasks, the model demonstrated strong initial performance, starting with solutions very close to the optimal values. In the Graph Coloring task, the model exhibited strict adherence to constraints. As shown in the *Average Colors Used* plot (Figure 1, top-right), the curve remains perfectly flat at 8.0, matching the ground-truth Chromatic Number (indicated by the dashed green line). This confirms that the model consistently generated valid 8-color solutions throughout the entire training process. In the 0-1 Knapsack problem, the model similarly achieved values nearest to 100% of the optimal solution. However, we observed that the *Policy Loss* was notably noisy at the start of training, suggesting initial instability before the agent converged to a stable policy.

Traveling Salesman Problem (TSP). The results for the TSP task (Figure 3) presented a different challenge. While the *Average Tour Length* (top-left) showed a steady downward trend, decreasing from 67 to 62, the model failed to fully reach the global optimum of 59.31 (dashed green line). Similarly, the *Best Tour Length* (top-right) rapidly improved from 64.3 to 63.2 but then plateaued, indicating that the model struggled to bridge the final gap to the theoretical optimum. The *Entropy* plot shows a significant decrease over time, confirming that the agent successfully transitioned from random exploration to exploiting the deterministic paths it had learned, though it settled into a local optimum.

5.3.3. Time Efficiency

In this section, we compare the computational efficiency of our PPO-based approach against exact algorithms for each combinatorial optimization problem. All experiments were conducted on a CPU (we using M4 Pro chip) to ensure fair comparison.

0-1 Knapsack. We compare our method against the standard Dynamic Programming (DP) algorithm with time complexity $O(n \times C_{\max})$, where n is the number of items and $C_{\max}$ is the maximum capacity [10]. As shown in Table I, the results reveal a clear scaling advantage for our approach. For small instances (p01–p07) with limited capacity values, the DP method achieves optimal solutions in under 2ms. However, for large instances (p08–p11) where capacity values exceed 6 million, the DP algorithm becomes prohibitively slow, requiring 20–49 seconds per instance due to the pseudo-polynomial complexity. In contrast, our PPO approach maintains consistent inference times of 5–8ms regardless of capacity size, achieving speedups of **3,826$\times$ to 6,565$\times$** over DP while maintaining an average optimality gap of only 1.49%.

Table I: Knapsack Problem Results

Instance	n	PPO(our)	Optimal	Gap (%)	PPO (ms)	DP (ms)	Speedup
p01	10	309	309	0.00	25.42	0.17	0.01x
p02	5	51	51	0.00	0.90	0.02	0.02x
p03	6	146	150	2.67	1.18	0.15	0.13x
p04	7	107	107	0.00	1.18	0.04	0.03x
p05	8	858	900	4.67	1.51	0.09	0.06x
p06	7	1735	1735	0.00	1.13	0.12	0.11x
p07	15	1429	1458	1.99	2.32	1.43	0.62x
p08	24	12 946 247	13 549 094	4.45	5.25	20 098.44	3,826.7x
p09	30	23 170 552	23 480 482	1.32	4.92	32 295.28	6,565.3x
p10	35	28 270 052	28 353 895	0.30	7.33	36 771.91	5,015.2x
p11	40	30 290 680	30 607 878	1.04	7.99	49 069.86	6,139.1x
Average	-	-	-	**1.49**	**5.38**	-	-

Traveling Salesman Problem (TSP). We compare our method against the Held-Karp algorithm [3, 24], a dynamic programming approach with time complexity $O(n^2 \times 2^n)$ and space complexity $O(n \times 2^n)$. This exponential complexity makes exact solutions computationally infeasible for instances with more than 20 cities. As shown in Table II, for the 10-city instance (tiny), both

Table II: Traveling Salesman Problem Results

Dataset	n	PPO(our)	Optimal	Gap (%)	PPO (ms)	Exact (ms)
tiny	10	12.5170	12.5170	0.00	8.36	3.63
small	30	66.9289	59.3055	12.85	27.96	7.79
small-1	20	15.0891	14.3839	4.90	17.21	10974.83
small-2	25	19.2255	16.4314	17.00	23.81	4.72
Average				**8.69**	**19.34**	**2747.74**

methods achieve comparable performance. However, for the 20-city instance (small-1), Held-Karp requires **10,974ms** while our PPO method completes in just **17.21ms**—a speedup of over **637**×. For larger instances (25–30 cities) where Held-Karp becomes infeasible, we use the 2-opt heuristic as a baseline. Our PPO approach maintains an average inference time of 19.34ms across all instances with an average gap of 8.69% from optimal.

Graph Coloring. Since the graph coloring problem is NP-complete, no polynomial-time exact algorithm is known. We compare against the greedy coloring heuristic with complexity $O(n^2)$, where $n = |V|$ is the number of vertices. As shown in Table III, our PPO approach matches the chromatic number χ(G) for 5 out of 6 Mycielski graphs (myciel3–myciel7), achieving an average gap of only 0.17 colors. The inference time scales with graph size, ranging from 2.62ms for myciel2 ($|V| = 5$) to 59.45ms for myciel7 ($|V| = 191$). While the greedy algorithm is faster in raw execution time, our learned policy achieves optimal colorings on challenging graphs where greedy heuristics typically struggle.

Table III: Graph Coloring Results

Graph	$\lvert V\rvert$	$\lvert E\rvert$	PPO(our)	χ(G)	Gap	PPO (ms)
myciel2	5	5	3	2	1	2.62
myciel3	11	20	4	4	0	5.01
myciel4	23	71	5	5	0	11.60
myciel5	47	236	6	6	0	27.27
myciel6	95	755	7	7	0	84.41
myciel7	191	2360	8	8	0	59.45
Average					**0.17**	**31.73**

6 Conclusion

In this work, we presented a Proximal Policy Optimization (PPO) based framework for solving combinatorial optimization problems, demonstrating its effectiveness on three classical NP-hard problems: the 0-1 Knapsack Problem, the Traveling Salesman Problem (TSP), and the Graph Coloring Problem.

6.1 Summary of Contributions

Our main contributions are as follows:

1. **Unified RL Framework.** We developed a unified reinforcement learning framework that formulates each combinatorial problem as a Markov Decision Process (MDP), enabling a consistent approach to learning constructive heuristics across different problem domains.
2. **Problem-Specific Neural Architectures.** We designed tailored neural network architectures for each problem: a Transformer-based encoder-decoder for TSP to capture global city dependencies, an MLP-based policy for Knapsack with item-level feature engineering, and a Graph Neural Network (GNN) for Graph Coloring to leverage graph structure through message passing.
3. **Near-Optimal Solutions with Significant Speedup.** Our experimental results demonstrate that PPO achieves near-optimal solutions with an average optimality gap of 1.49% for Knapsack, 8.69% for TSP, and 0.17 colors for Graph Coloring. Crucially, for large-scale instances, our approach achieves speedups of 3,000× to 7,000× over exact dynamic programming algorithms while maintaining solution quality.
4. **Scalability.** Unlike exact algorithms whose complexity grows exponentially (Held-Karp) or pseudo-polynomially (Knapsack DP), our learned policies provide constant-time inference $O(n)$ per decision step, making them suitable for real-time applications.

6.2 Limitations

Despite the promising results, our approach has several limitations. First, the trained models are specific to the problem size and structure seen during training, requiring retraining for significantly different instances. Second, the stochastic nature of the policy leads to variance in solution quality across runs. Third, for very small instances, exact algorithms remain more efficient and guarantee optimal solutions.

6.3 Future Work

Several promising directions exist for extending this work:

1. **Curriculum Learning Ablation Studies.** We plan to conduct comprehensive ablation studies on curriculum learning strategies, investigating how progressive difficulty scheduling affects both learning efficiency and final solution quality. This includes studying the optimal number of curriculum stages, the choice of difficulty metrics (problem size, constraint tightness, graph density), and transfer learning capabilities across difficulty levels.
2. **Generative Flow Networks (GFlowNets).** A significant limitation of PPO is that it learns a single policy that produces one solution trajectory. We propose exploring GFlowNets [6] as an alternative paradigm that learns to sample diverse, high-quality solutions proportionally to their reward. This would enable:
   - **Multi-Modal Solution Discovery:** Finding multiple near-optimal solutions rather than converging to a single mode
   - **Improved Exploration:** GFlowNets' ability to generate diverse trajectories may help escape local optima
   - **Posterior Sampling:** Generating solutions proportional to $\exp(R(x)/T)$ for temperature-controlled exploration
3. **Hybrid Approaches.** Combining learned heuristics with local search methods (e.g., 2-opt for TSP, tabu search) could improve solution quality while preserving computational efficiency.
4. **Larger-Scale Benchmarks.** Evaluating on industry-standard benchmarks such as TSPLIB[14, 28], DIMACS graph coloring instances[16, 12], and harder Knapsack variants would provide stronger validation of the approach's practical applicability.
5. **Multi-Objective Optimization.** Extending the framework to handle multiple competing objectives, such as simultaneously minimizing cost and maximizing fairness in routing problems.

6.4 Concluding Remarks

This work demonstrates that deep reinforcement learning, specifically PPO, offers a viable and practical approach to combinatorial optimization. By learning problem-specific heuristics from experience, our method bridges the gap between the guaranteed optimality of exact algorithms and the speed of hand-crafted heuristics. As combinatorial problems continue to grow in scale and complexity in real-world applications, learned optimization methods represent a promising direction for achieving both efficiency and solution quality.

References

[1] Tanweer Alam, Shamimul Qamar, Amit Dixit, and Mohamed Benaida. Genetic algorithm: Reviews, implementations, and applications. *International Journal of Engineering Pedagogy*, 10(6):57–77, 2020.

[2] Abhinav Anand, Subrahmanya S. Peruru, and Amitangshu Pal. Vcolrl: Learn to solve the vertex coloring problem using reinforcement learning. *Transactions on Machine Learning Research*, 2025.

[3] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the Association for Computing Machinery*, 9(1):61–63, 1962.

[4] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *International Conference on Learning Representations*, 2017.

[5] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th International Conference on Machine Learning*, pages 41–48, 2009.

[6] Yoshua Bengio, Salem Lahlou, Tristan Deleu, Edward J. Hu, Mo Tiwari, and Emmanuel Bengio. Gflownet foundations. *arXiv preprint arXiv:2111.09266*, 2021.

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

[8] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[9] Sergio Caracciolo, Alexander K. Hartmann, Scott Kirkpatrick, and Martin Weigel. Simulated annealing, optimization, searching for ground states. *arXiv preprint arXiv:2301.00683*, 2023.

[10] e-maxx.ru team. Knapsack problem. https://cp-algorithms.com/dynamic_programming/knapsack.html, 2014. Accessed: 2025-12-10.

[11] Christian D. Hubbs, Hector D. Perez, Owais Sarwar, Nikolaos V. Sahinidis, Ignacio E. Grossmann, and John M. Wassick. Or-gym: A reinforcement learning library for operations research problems. *arXiv preprint arXiv:2008.06319*, 2020.

[12] David S. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Providence, RI, 1996.

[13] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

[14] mastqe. TSPLIB: Repository of symmetric TSP data. https://github.com/mastqe/tsplib, 2017. Accessed: 2025-12-12.

[15] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.

[16] Daniel C. Porumbel. DIMACS graphs: Benchmark instances and best upper bounds. https://cedric.cnam.fr/~porumbed/graphs/, 2011. Accessed: 2025-12-10.

[17] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003.

[18] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 1889–1897, 2015.

[19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[20] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, volume 12, 2000.

[21] Ali A. R. Taresh et al. Solving the traveling salesman problem with drones using ppo and deep rl. *Journal of Information Systems Engineering & Management*, 10(1), 2025.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[23] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, volume 28, 2015.

[24] Wikipedia contributors. Held–karp algorithm. https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm, 2025. Accessed: 2025-12-10.

[25] Hailemicael L. Yimer, Pei Yang, and Letu Qingge. An improved actor-critic architecture with ppo for the traveling salesman problem. *Expert Systems with Applications*, 298, 2025.

[26] Emre Yolcu, Xinyu Wu, and Marijn J. H. Heule. Mycielski graphs and PR proofs. In *23rd International Conference on Theory and Applications of Satisfiability Testing*, pages 202–210, 2020.

[27] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[28] Ziya07. TSPLIB dataset: Traveling salesman problem. https://www.kaggle.com/datasets/ziya07/traveling-salesman-problem-tsplib-dataset, 2024. Accessed: 2025-12-10.

A Preliminaries

A.1 Markov Decision Process

We formulate combinatorial optimization problems as finite-horizon Markov Decision Processes (MDPs). An MDP is defined by the tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, T)$, where:

- $\mathcal{S}$ denotes the state space,
- $\mathcal{A}$ denotes the action space,
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ denotes the transition probability function,
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ denotes the reward function,
- $\gamma \in [0, 1]$ denotes the discount factor,
- $T \in \mathbb{N}$ denotes the horizon (episode length).

A trajectory τ is a sequence of states and actions:

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T)$$

where $r_t = \mathcal{R}(s_t, a_t)$ denotes the reward received at time step t.

A.2 Policy

A stochastic policy $\pi_\theta : \mathcal{S} \times \mathcal{A} \to [0, 1]$ defines a probability distribution over actions conditioned on states, parameterized by $\theta \in \mathbb{R}^d$:

$$\pi_\theta(a \mid s) = \mathbb{P}(a_t = a \mid s_t = s; \theta)$$

The policy satisfies the probability axioms:

$$\sum_{a \in \mathcal{A}(s)} \pi_\theta(a \mid s) = 1, \quad \forall s \in \mathcal{S}$$

where $\mathcal{A}(s) \subseteq \mathcal{A}$ denotes the set of valid actions in state s.

A.3 Value Functions

Definition A.1 (State Value Function). The state value function $V^\pi : \mathcal{S} \to \mathbb{R}$ under policy π is defined as the expected cumulative discounted reward starting from state s:

$$V^\pi(s) = \mathbb{E}_{\tau \sim \pi}\left[\sum_{t=0}^{T-1} \gamma^t r_t \,\middle|\, s_0 = s\right]$$

where the expectation is taken over trajectories τ generated by following policy π.

Definition A.2 (Action Value Function). The action value function $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ under policy π is defined as the expected cumulative discounted reward starting from state s, taking action a, and thereafter following policy π:

$$Q^\pi(s, a) = \mathbb{E}_{\tau \sim \pi}\left[\sum_{t=0}^{T-1} \gamma^t r_t \,\middle|\, s_0 = s, a_0 = a\right]$$

Definition A.3 (Advantage Function). The advantage function $A^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ quantifies the relative benefit of taking action a in state s compared to the average action under policy π:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$$

Remark A.1. The advantage function satisfies $\mathbb{E}_{a \sim \pi(\cdot|s)}[A^\pi(s, a)] = 0$ for all states $s \in \mathcal{S}$.

A.4 Bellman Equations

The value functions satisfy the Bellman equations:

Bellman Equation for V^π:

$$V^\pi(s) = \mathbb{E}_{a \sim \pi(\cdot|s)} \left[\mathcal{R}(s, a) + \gamma \mathbb{E}_{s' \sim \mathcal{P}(\cdot|s,a)}[V^\pi(s')] \right]$$

Bellman Equation for Q^π:

$$Q^\pi(s, a) = \mathcal{R}(s, a) + \gamma \mathbb{E}_{s' \sim \mathcal{P}(\cdot|s,a)} \left[\mathbb{E}_{a' \sim \pi(\cdot|s')}[Q^\pi(s', a')] \right]$$

B Policy Gradient Methods

B.1 Objective Function

The objective of reinforcement learning is to find a policy that maximizes the expected cumulative reward:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T-1} \gamma^t r_t \right] = \mathbb{E}_{s_0 \sim \rho_0}[V^{\pi_\theta}(s_0)]$$

where ρ_0 denotes the initial state distribution.

B.2 Policy Gradient Theorem

Theorem B.1 (Policy Gradient Theorem). The gradient of the objective function with respect to the policy parameters is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot A^{\pi_\theta}(s_t, a_t) \right]$$

Proof. See Sutton et al. (2000) [20] for the complete derivation.

B.3 REINFORCE Estimator

The policy gradient can be estimated using Monte Carlo sampling:

$$\hat{g} = \frac{1}{|\mathcal{B}|} \sum_{\tau \in \mathcal{B}} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot \hat{A}_t$$

where $\mathcal{B}$ denotes a batch of trajectories and $\hat{A}_t$ denotes an estimator of the advantage function.

C Proximal Policy Optimization

C.1 Trust Region Methods

Trust region methods constrain the policy update to prevent destructively large changes. The Trust Region Policy Optimization (TRPO) objective is:

$$\max_\theta \quad \mathbb{E}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t \right]$$
$$\text{subject to} \quad \mathbb{E}_t \left[D_{\text{KL}}(\pi_{\theta_{\text{old}}}(\cdot \mid s_t) \,\|\, \pi_\theta(\cdot \mid s_t)) \right] \leq \delta$$

where D_{KL} denotes the Kullback-Leibler divergence.

C.2 PPO Clipped Objective

Proximal Policy Optimization (PPO) replaces the hard constraint with a clipped objective function.

Definition C.1 (Probability Ratio). The probability ratio between the current and old policies is defined as:

$$\rho_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

Definition C.2 (Clipped Surrogate Objective). The PPO clipped objective is defined as:

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

where $\epsilon \in (0,1)$ is the clipping hyperparameter and the clip function is defined as:

$$\text{clip}(x,a,b) = \max(a, \min(x,b))$$

Proposition C.1. The clipped objective provides a lower bound on the unclipped objective when $\hat{A}_t > 0$ and the ratio ρ_t exceeds $1+\epsilon$, and similarly when $\hat{A}_t < 0$ and ρ_t falls below $1-\epsilon$.

Proof.
Consider two cases:

- *Case 1:* $\hat{A}_t \geq 0$ (advantageous action).
  - If $\rho_t \leq 1+\epsilon$: $\min(\rho_t\hat{A}_t, (1+\epsilon)\hat{A}_t) = \rho_t\hat{A}_t$
  - If $\rho_t > 1+\epsilon$: $\min(\rho_t\hat{A}_t, (1+\epsilon)\hat{A}_t) = (1+\epsilon)\hat{A}_t < \rho_t\hat{A}_t$
- *Case 2:* $\hat{A}_t < 0$ (disadvantageous action).
  - If $\rho_t \geq 1-\epsilon$: $\min(\rho_t\hat{A}_t, (1-\epsilon)\hat{A}_t) = \rho_t\hat{A}_t$
  - If $\rho_t < 1-\epsilon$: $\min(\rho_t\hat{A}_t, (1-\epsilon)\hat{A}_t) = (1-\epsilon)\hat{A}_t < \rho_t\hat{A}_t$

Thus, the clipped objective removes incentives for moving the ratio outside $[1-\epsilon, 1+\epsilon]$.

C.3 Value Function Loss

The value function is trained to minimize the mean squared error between predicted values and empirical returns:

$$\mathcal{L}^{\text{VF}}(\phi) = \mathbb{E}_t\left[\left(V_\phi(s_t) - \hat{R}_t\right)^2\right]$$

where $\hat{R}_t$ denotes the target return and ϕ denotes the value function parameters.

Definition C.3 (Discounted Return). The discounted return from time step t is defined as:

$$\hat{R}_t = \sum_{k=0}^{T-1-t} \gamma^k r_{t+k}$$

C.4 Entropy Regularization

To encourage exploration and prevent premature convergence, an entropy bonus is added to the objective.

Definition C.4 (Policy Entropy). The entropy of the policy distribution at state s is:

$$\mathcal{H}[\pi_\theta(\cdot \mid s)] = -\sum_{a \in \mathcal{A}(s)} \pi_\theta(a \mid s) \log \pi_\theta(a \mid s)$$

The entropy loss is defined as:

$$\mathcal{L}^{\text{ENT}}(\theta) = -\mathbb{E}_t\left[\mathcal{H}[\pi_\theta(\cdot \mid s_t)]\right]$$

C.5 Combined PPO Objective

The complete PPO objective function is:

$$\mathcal{L}^{\text{PPO}}(\theta, \phi) = -\mathcal{L}^{\text{CLIP}}(\theta) + c_1 \mathcal{L}^{\text{VF}}(\phi) + c_2 \mathcal{L}^{\text{ENT}}(\theta)$$

where $c_1, c_2 > 0$ are weighting coefficients.

Remark C.1. The negative sign before $\mathcal{L}^{\text{CLIP}}$ converts the maximization problem to minimization. Similarly, the positive sign before $\mathcal{L}^{\text{ENT}}$ encourages higher entropy (since $\mathcal{L}^{\text{ENT}}$ is the negative entropy).

D Generalized Advantage Estimation

D.1 Temporal Difference Residual

Definition D.1 (TD Residual). The temporal difference residual at time step t is defined as:

$$\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$$

where $V_\phi(s_T) = 0$ for terminal states.

Proposition D.1. The TD residual is an unbiased estimator of the advantage function when the value function is exact, i.e., $V_\phi = V^\pi$:

$$\mathbb{E}[\delta_t \mid s_t, a_t] = A^\pi(s_t, a_t)$$

D.2 GAE Definition

Definition D.2 (Generalized Advantage Estimation). The GAE estimator with parameters (γ, λ) is defined as:

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \sum_{k=0}^{T-1-t} (\gamma\lambda)^k \delta_{t+k}$$

Expanding this expression:

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \cdots + (\gamma\lambda)^{T-1-t}\delta_{T-1}$$

D.3 Recursive Formulation

Proposition D.2.
The GAE estimator satisfies the following recursive relationship:

$$\hat{A}_t = \delta_t + \gamma\lambda \hat{A}_{t+1}$$

with boundary condition $\hat{A}_T = 0$.

Proof.
By definition:

$$\hat{A}_t = \sum_{k=0}^{T-1-t} (\gamma\lambda)^k \delta_{t+k} = \delta_t + \gamma\lambda \sum_{k=0}^{T-2-t} (\gamma\lambda)^k \delta_{t+1+k} = \delta_t + \gamma\lambda \hat{A}_{t+1}$$

D.4 Bias-Variance Trade-off

Proposition D.3. The GAE parameter λ controls the bias-variance trade-off:

- When $\lambda = 0$: $\hat{A}_t^{\text{GAE}(\gamma,0)} = \delta_t$ (one-step TD, low variance, high bias)
- When $\lambda = 1$: $\hat{A}_t^{\text{GAE}(\gamma,1)} = \sum_{k=0}^{T-1-t} \gamma^k r_{t+k} - V_\phi(s_t)$ (Monte Carlo, high variance, low bias)

Proof.
For $\lambda = 0$:

$$\hat{A}_t^{\text{GAE}(\gamma,0)} = \sum_{k=0}^{T-1-t} 0^k \delta_{t+k} = \delta_t$$

For $\lambda = 1$:

$$\hat{A}_t^{\text{GAE}(\gamma,1)} = \sum_{k=0}^{T-1-t} \gamma^k \delta_{t+k} = \sum_{k=0}^{T-1-t} \gamma^k (r_{t+k} + \gamma V_\phi(s_{t+k+1}) - V_\phi(s_{t+k}))$$

The telescoping sum yields:

$$\hat{A}_t^{\text{GAE}(\gamma,1)} = \sum_{k=0}^{T-1-t} \gamma^k r_{t+k} + \gamma^{T-t} V_\phi(s_T) - V_\phi(s_t) = \sum_{k=0}^{T-1-t} \gamma^k r_{t+k} - V_\phi(s_t)$$

since $V_\phi(s_T) = 0$.

D.5 Advantage Normalization

To reduce variance and stabilize training, advantages are normalized across each batch:

$$\hat{A}_t \leftarrow \frac{\hat{A}_t - \mu_{\hat{A}}}{\sigma_{\hat{A}} + \varepsilon}$$

where $\mu_{\hat{A}} = \frac{1}{|\mathcal{B}|} \sum_{t \in \mathcal{B}} \hat{A}_t$, $\sigma_{\hat{A}} = \sqrt{\frac{1}{|\mathcal{B}|} \sum_{t \in \mathcal{B}} (\hat{A}_t - \mu_{\hat{A}})^2}$, and $\varepsilon > 0$ is a small constant for numerical stability.

E Implementation Details

The implementation is developed in Python using PyTorch. We using single node A100 for training. Key implementation choices include:

- **Batched Environments:** Multiple copies of each problem instance are solved in parallel to improve sample efficiency.
- **Deterministic Transitions:** All environment transitions are deterministic, simplifying the MDP formulation.
- **Feasibility Masking:** Invalid actions are masked at the policy level to ensure all generated solutions satisfy problem constraints.
- **Advantage Normalization:** Advantages are normalized per batch to stabilize training.
- **Gradient Clipping:** Gradient norms are clipped to 0.5 to prevent training instabilities.

All experiments are conducted on static benchmark datasets to enable fair comparison across training runs and evaluation against known optimal solutions. We list the training hyperparameters in Table IV.

Table IV: Hyperparameters used for training

Parameter	**TSP**	**Knapsack**	**Graph Coloring**
Learning rate	10^{-4}	3×10^{-4}	3×10^{-4}
Batch size	128	128	128
Discount factor γ	1.0	1.0	1.0
GAE parameter λ	0.95	0.95	0.95
Clip parameter ϵ	0.2	0.2	0.2
Embedding dimension	128	64	64
Hidden dimension	-	128	128
Number of layers	3	-	3
Attention heads	8	-	-
PPO epochs per iteration	4	4	4
Training epochs	500	500	500

F Additional Experimental Results

F.1 Curriculum learning on Graph coloring

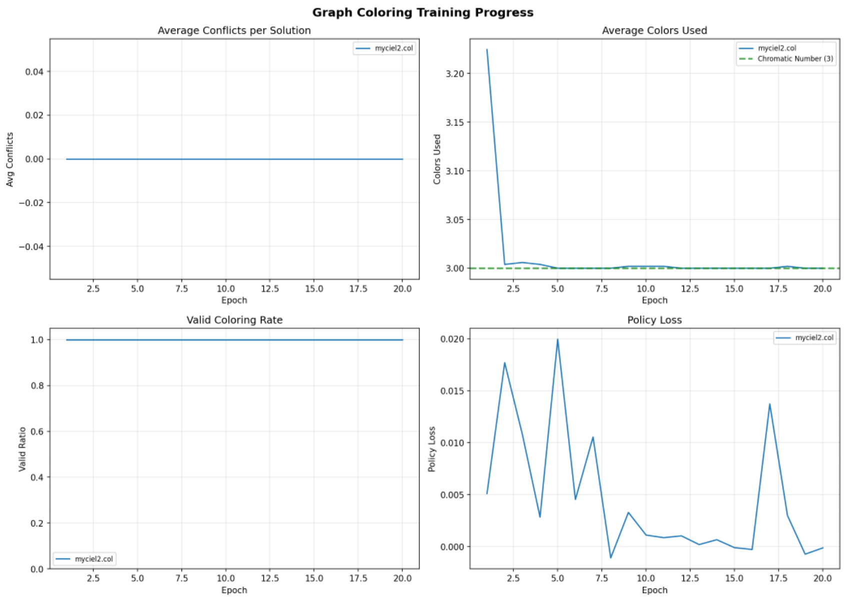

Figure 4: Graph coloring in curriculum myciel2 data

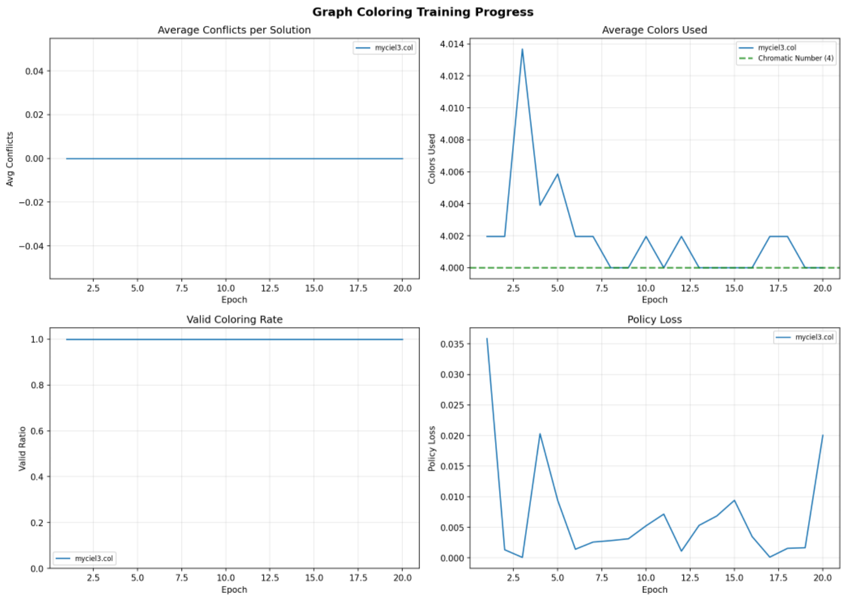

Figure 5: Graph coloring in curriculum myciel3 data

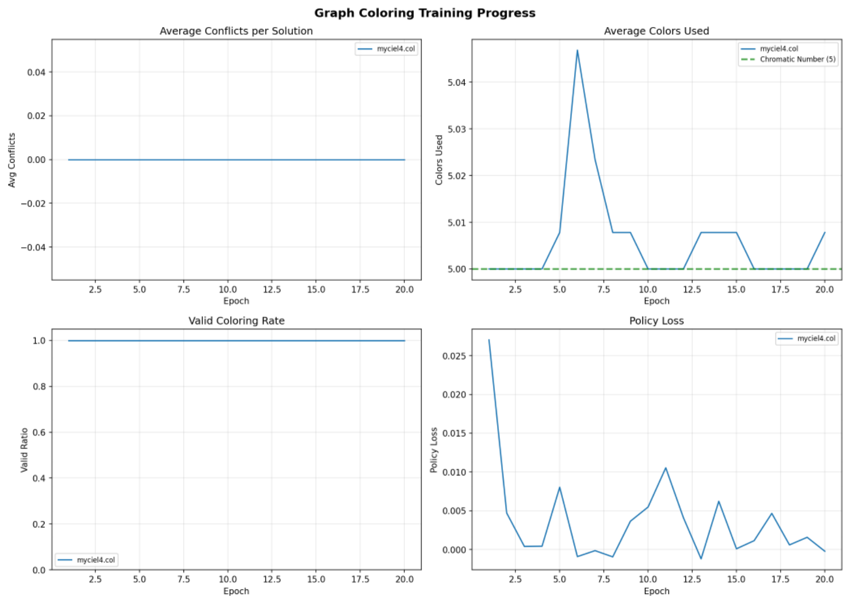

Figure 6: Graph coloring in curriculum myciel4 data

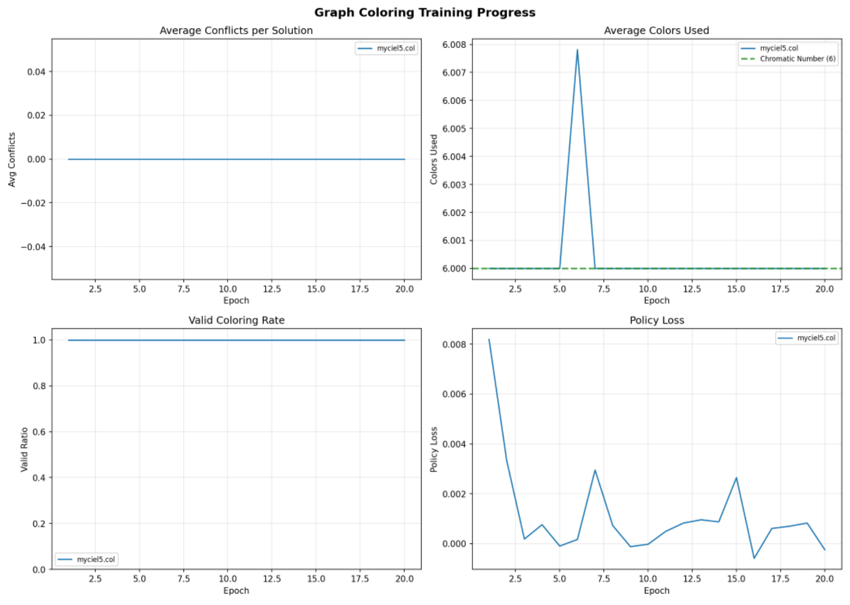

Figure 7: Graph coloring in curriculum myciel5 data

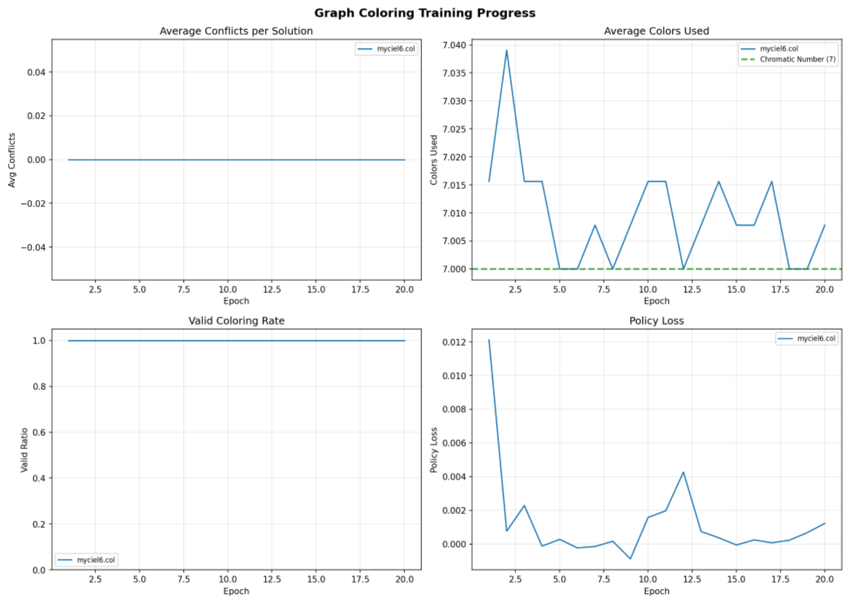

Figure 8: Graph coloring in curriculum myciel6 data

F.2 Knapsack on single Training

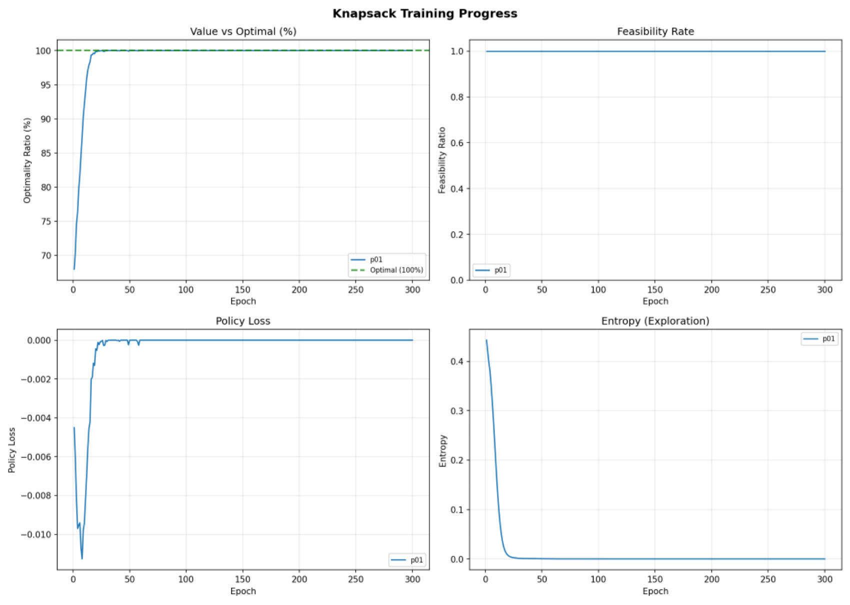

Figure 9: Knapsack problem in p01 single data

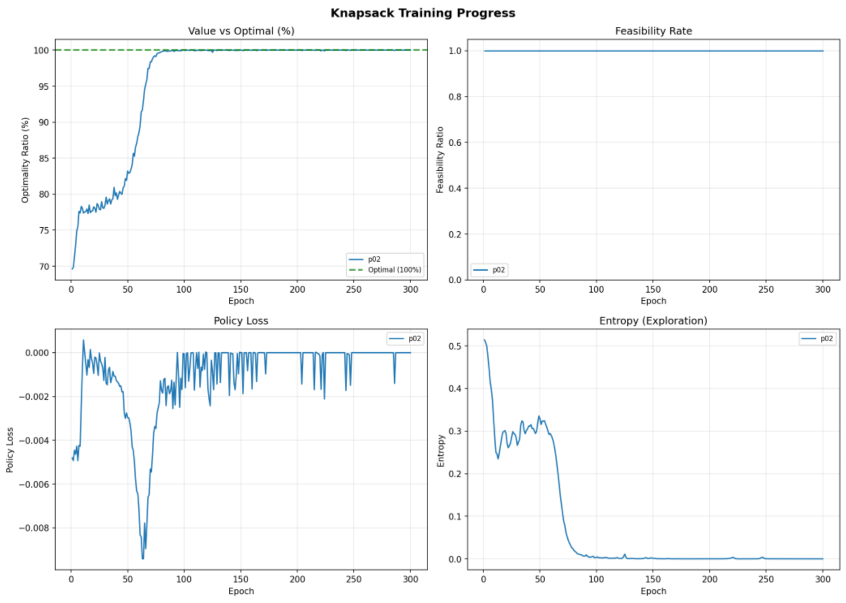

Figure 10: Knapsack problem in p02 single data

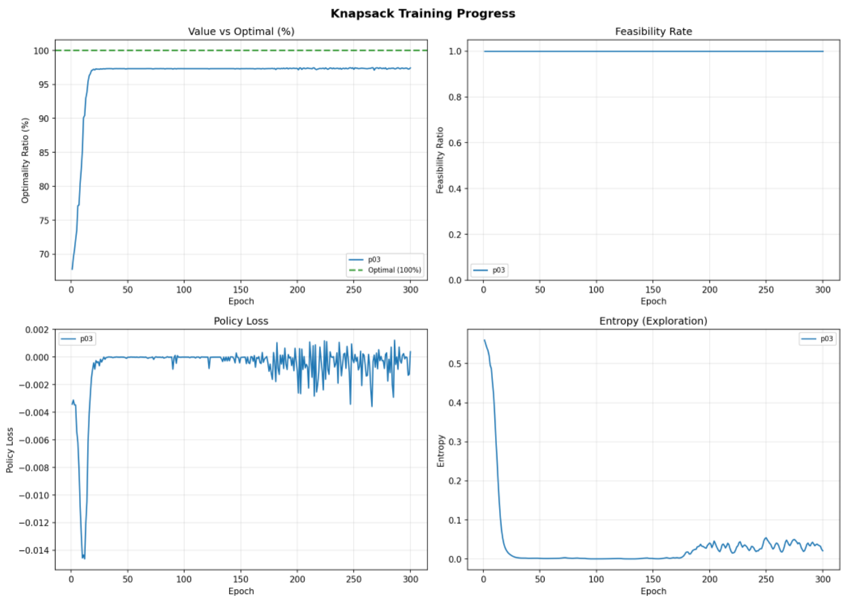

Figure 11: Knapsack problem in p03 single data

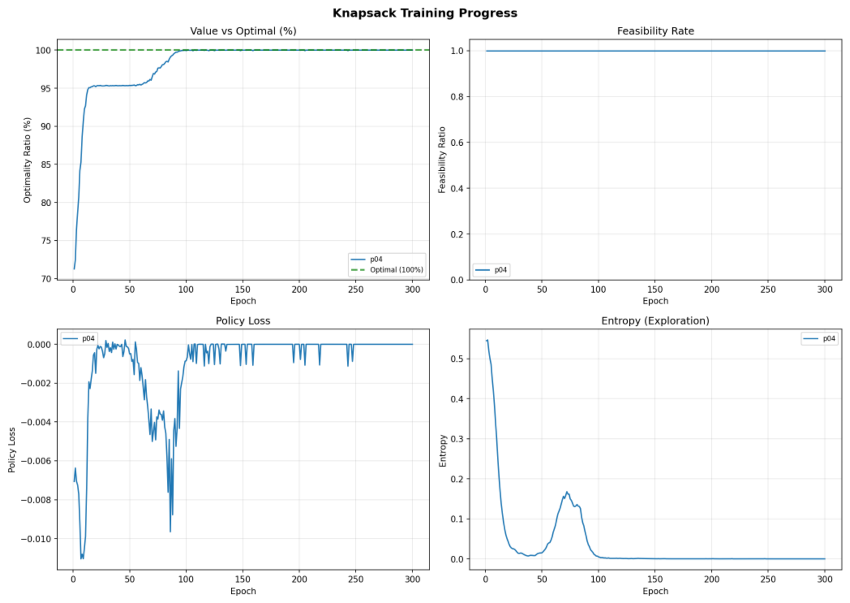

Figure 12: Knapsack problem in p04 single data

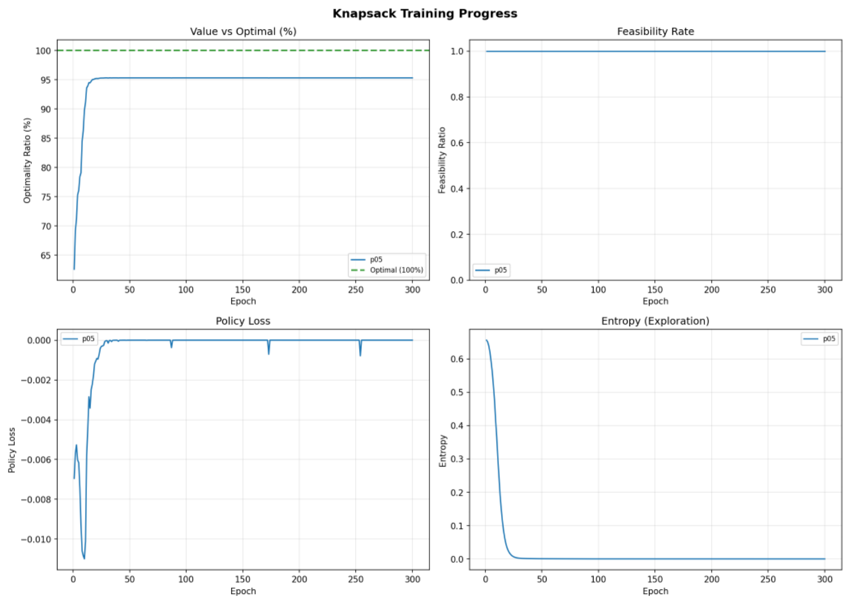

Figure 13: Knapsack problem in p05 single data

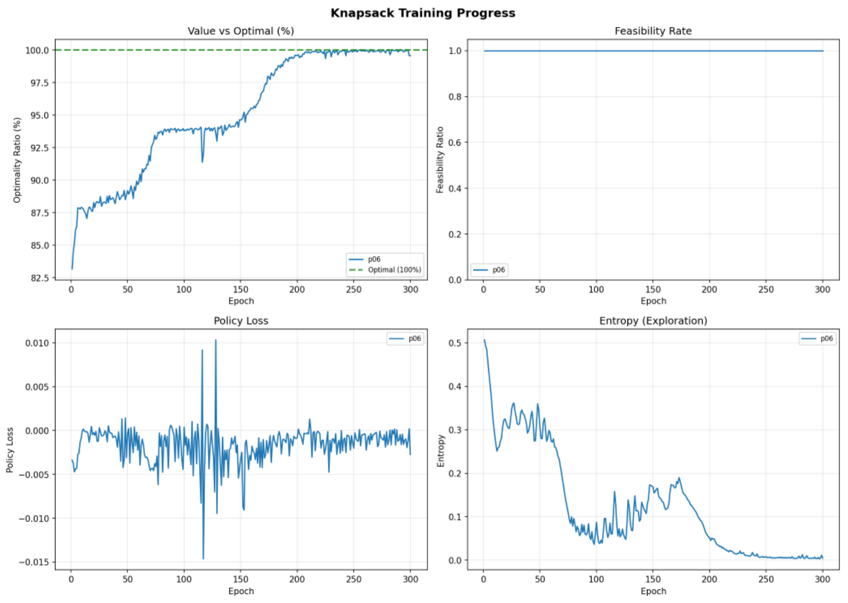

Figure 14: Knapsack problem in p06 single data

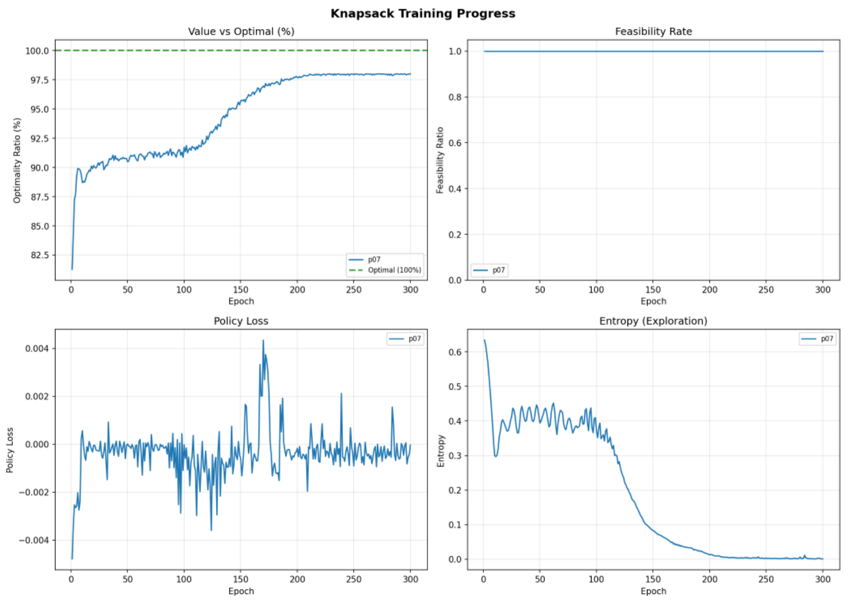

Figure 15: Knapsack problem in p07 single data

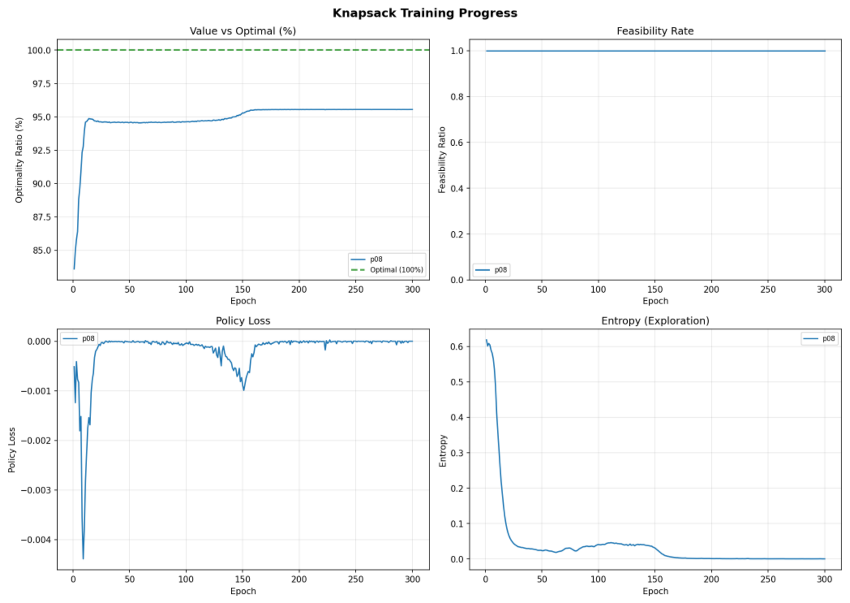

Figure 16: Knapsack problem in p08 single data

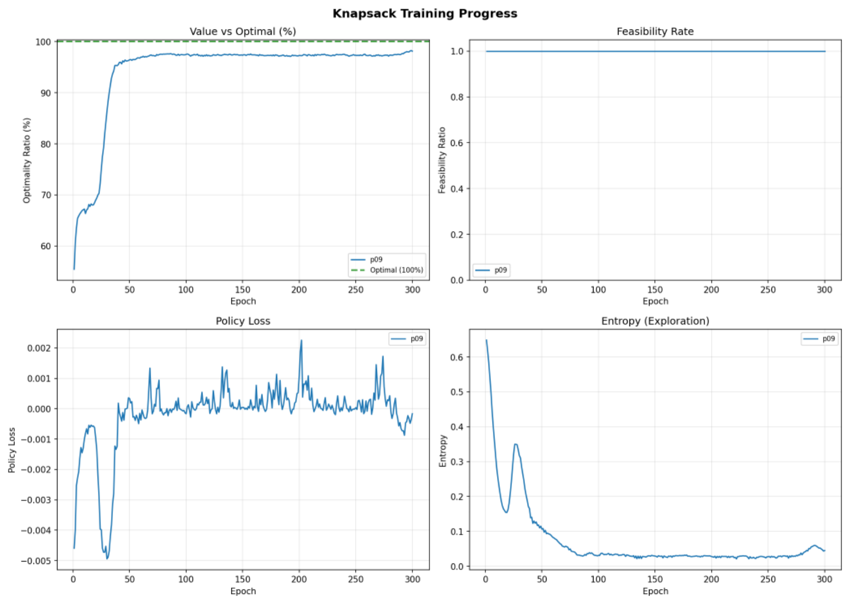

Figure 17: Knapsack problem in p09 single data

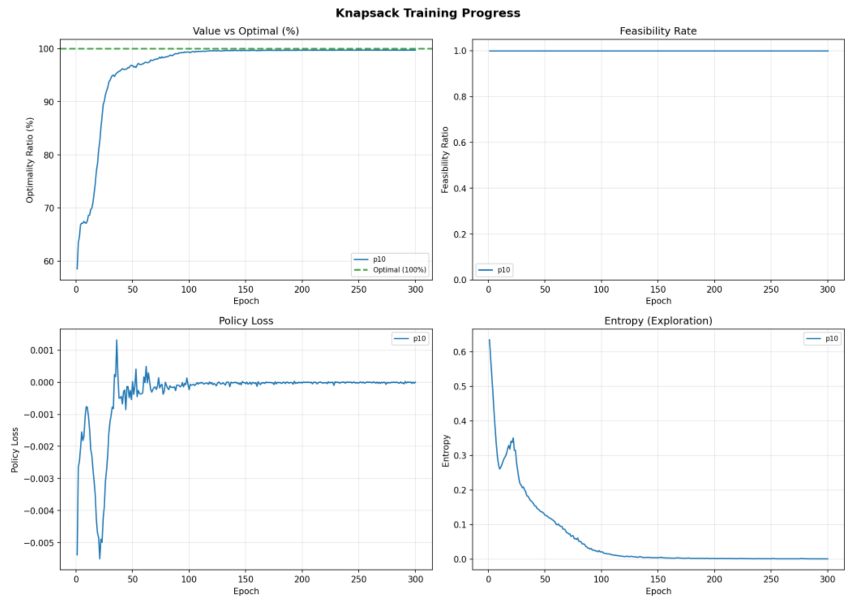

Figure 18: Knapsack problem in p10 single data

F.3 TSP on Curriculum learning

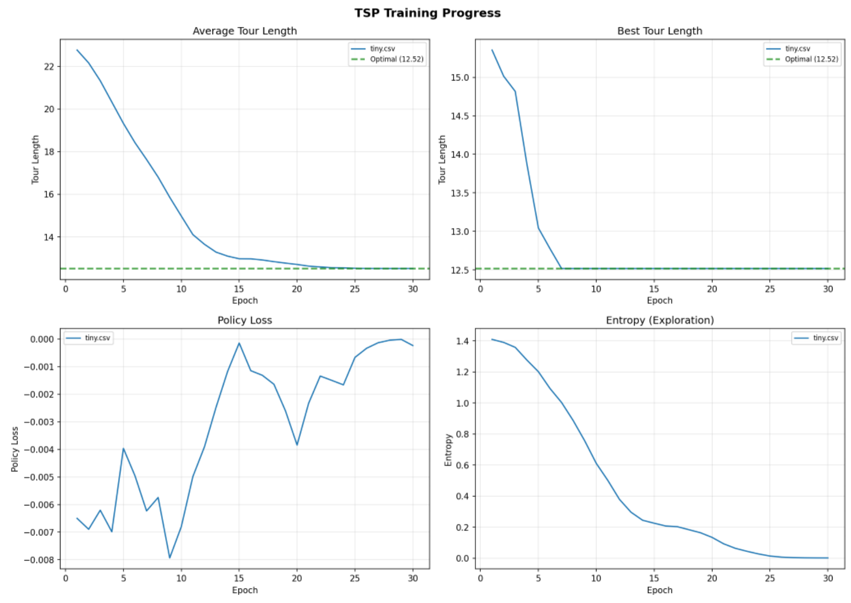

Figure 19: Traveling salesperson problem in curriculum tiny data

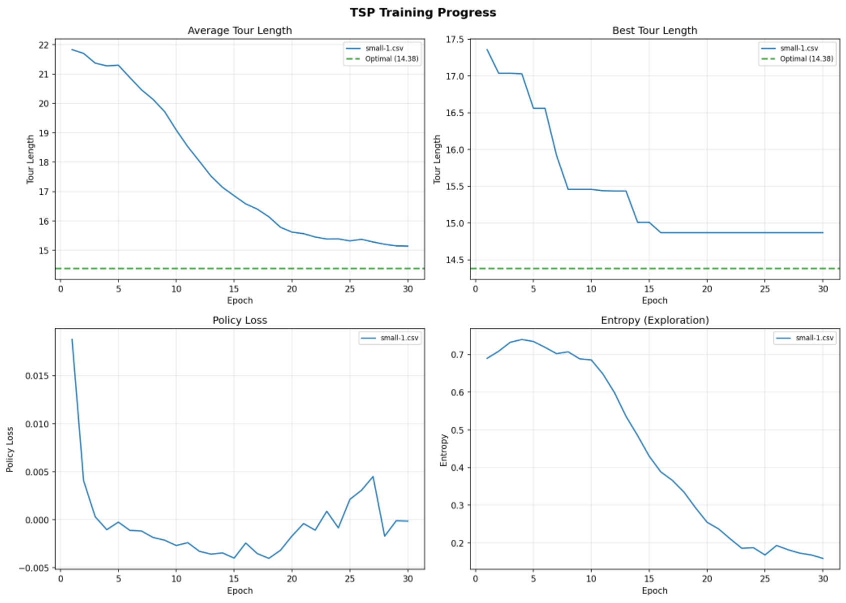

Figure 20: Traveling salesperson problem in curriculum small-1 data

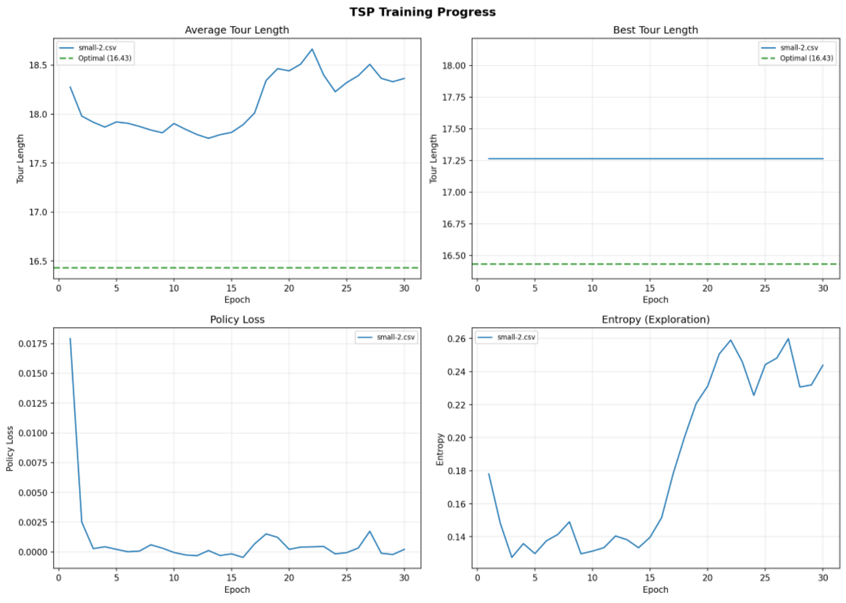

Figure 21: Traveling salesperson problem in curriculum small-2 data