ReFill: Reinforcement Learning for Fill-In Minimization

Elfarouk Harb
University of Illinois Urbana-Champaign
eyfmharb@gmail.com

Ho Shan Lam
The Trade Desk
sharonhslhk@gmail.com

Abstract

Efficiently solving sparse linear systems $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a large, sparse, symmetric positive semi-definite matrix, is a core challenge in scientific computing, machine learning, and optimization. A major bottleneck in Gaussian elimination for these systems is *fill-in*—the creation of non-zero entries that increase memory and computational cost. Minimizing fill-in is NP-hard, and existing heuristics like Minimum Degree and Nested Dissection offer limited adaptability across diverse problem instances.

We introduce *ReFill*, a reinforcement learning framework enhanced by Graph Neural Networks (GNNs) to learn adaptive ordering strategies for fill-in minimization. ReFill trains a GNN-based heuristic to predict efficient elimination orders, outperforming traditional heuristics by dynamically adapting to the structure of input matrices. Experiments demonstrate that ReFill outperforms strong heuristics in reducing fill-in, highlighting the untapped potential of learning-based methods for this well-studied classical problem.

1 Introduction and Background

In this paper, we consider the fundamental problem of solving the system of equations $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a $n \times n$ symmetric positive semi-definite matrix. Such sparse linear systems show up naturally in scientific computing, machine learning, theoretical computer science, and scalable optimization [12, 8, 5, 7, 17, 42, 40, 41]. These systems are instrumental in numerous domains ranging from finite element analysis to large graph-based models in machine learning. Typically, Gaussian elimination serves as a primary solution method. However, the efficiency of Gaussian elimination is highly sensitive to matrix ordering, as different variable elimination orderings introduce varying levels of *fill-in*—the additional non-zero entries created during variable elimination, which directly impact memory usage and runtime. If the matrix $\mathbf{A}$ is dense, then Gaussian Elimination requires $\Theta(n^3)$ time. However, if $\mathbf{A}$ is sparse, we might be able to save time by avoiding explicit manipulation of zeros or fill-in. Reducing fill-in, therefore, is critical for optimizing resource use and enabling scalable computation in high-dimensional settings.

One way to minimize fill-in during Gaussian elimination is to reorder the rows and columns of $\mathbf{A}$ to solve an equivalent system $\mathbf{PAP}^T$, where $\mathbf{P}$ denotes a permutation matrix. See Figure 1 for an example where different variable elimination order in Gaussian elimination leads to fewer fill-in. This approach, introduced in the seminal works by Tarjan and Rose [40, 41, 42], reinterprets fill-in into a graph-theoretic perspective, thereby providing structural insights into sparse matrices. Specifically, we will recap *elimination orders* and *fill-in minimization* within the context of graph theory.

Let us consider a symmetric positive semi-definite system $\mathbf{Ax} = \mathbf{b}$, where the sparse structure of $\mathbf{A}$ can be represented by an undirected graph $G(\mathbf{A})$. In this graph, each vertex corresponds to a row (or column) in $\mathbf{A}$,

$$(a)\quad \begin{bmatrix} 8 & 3 & 0 & 4 & 1 \\ 3 & 4 & 0 & 0 & 0 \\ 0 & 0 & 3 & 2 & 0 \\ 4 & 0 & 2 & 6 & 0 \\ 1 & 0 & 0 & 0 & 2 \end{bmatrix} \qquad (b)\quad \begin{bmatrix} 4 & 0 & 0 & 0 & 3 \\ 0 & 3 & 2 & 0 & 0 \\ 0 & 2 & 6 & 0 & 4 \\ 0 & 0 & 0 & 2 & 1 \\ 3 & 0 & 4 & 1 & 8 \end{bmatrix}$$

$$\downarrow \qquad\qquad\qquad\qquad \downarrow$$

$$\begin{bmatrix} 1 & \frac{3}{8} & 0 & \frac{1}{2} & \frac{1}{8} \\ 0 & 1 & 0 & -\frac{12}{23} & -\frac{3}{23} \\ 0 & 0 & 1 & \frac{2}{3} & 0 \\ 0 & 0 & 0 & 1 & -\frac{24}{65} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 & \frac{3}{4} \\ 0 & 1 & \frac{2}{3} & 0 & 0 \\ 0 & 0 & 1 & 0 & \frac{6}{7} \\ 0 & 0 & 0 & 1 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 1: **(a)** A symmetric positive-definite matrix $\mathbf{A}$ and its factorization after Gaussian elimination with a fixed variable elimination order. This ordering introduces three fill-in entries at positions $A_{2,4}$, $A_{2,5}$, and $A_{4,5}$, which were initially zero but became non-zero during elimination. **(b)** The same matrix after applying a permutation $\pi = (5, 1, 2, 3, 4)$ via permutation matrix $\mathbf{P}_\pi$, which reorders the rows and columns of $\mathbf{A}$ by moving the first row to the last row, and the first column to the last position resulting in matrix $\mathbf{A'}$. This reordering leads to a matrix where Gaussian elimination introduces no fill-in, resulting in lower memory usage and faster computation.

with an edge $ij, i \neq j$ present if $\mathbf{A}_{i,j} \neq 0$. The process of Gaussian elimination on $\mathbf{A}$ can then be interpreted in terms of *eliminating* vertices in $G(\mathbf{A})$. Specifically, eliminating a vertex i in G' entails connecting all neighbors of i (denoted $\mathcal{N}_{G'}(i)$) to form a clique, followed by the deletion of i from G'. This in turn adds "fill-in" edges, neighbors of i that were previously not connected.

Each permutation matrix $\mathbf{P}$ determines a specific ordering π of the vertices. Executing Gaussian elimination according to this ordering $\pi = (\pi(1), \ldots, \pi(n))$ means sequentially eliminating variables in the specified order. The edges that emerge from this process are collectively known as the *fill-in*. See Figure 2 for an illustration.

To formalize this process, consider the iterative graph sequence associated with an ordering π:

1. Set $G^{(0)} \leftarrow G(\mathbf{A})$.
2. For each $i = 1$ to n:
   - Define $\mathcal{P}_\pi(i) = \{(v, w) : v, w \in \mathcal{N}_{G^{(i-1)}}(\pi(i)), v \neq w, (v, w) \notin E(G^{i-1})\}$ as the fill-in edges that will be added by eliminating $\pi(i)$.
   - Eliminate $\pi(i)$ from $G^{(i-1)}$, resulting in the updated graph $G^{(i)} = G^{(i-1)} \cup \mathcal{P}_\pi(i) \setminus \{\pi(i)\}$.

The edges $E_\pi = \cup_i \mathcal{P}_\pi(i)$ thus represent the fill-in that occurs when performing Gaussian elimination in the order π.

The following theorem offers a characterization of the fill-in associated with any ordering π.

Theorem 1.1 ([42])**.** *Given an ordering π and assuming no "lucky cancellations"[1], an edge $ij \in E_\pi$ exists if*

[1]Lucky cancellations happens for some matrices $\mathbf{A}$ where some non-zero elements incidentally get cancelled to zeros when eliminating a variable. In practice, this is unlikely to happen, so one often ignores their effect.

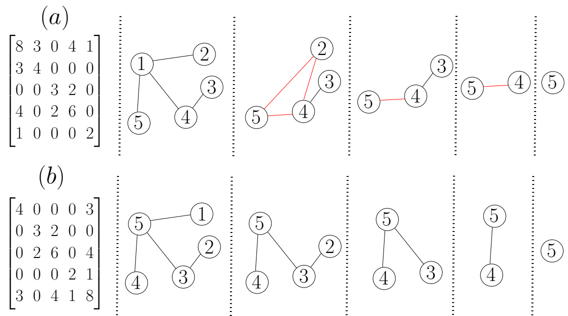

Figure 2: **(a)** The matrix $\mathbf{A}$ from Figure 1 and its corresponding graph representation $G(\mathbf{A})$. Vertices are eliminated in the order $1, 2, 3, 4, 5$. Eliminating vertex 1 forces its neighbors $(2, 4, 5)$ to form a clique, introducing three fill-in edges $(2, 4), (2, 5), (4, 5)$ (shown in red) before vertex 1 is removed. These fill-in edges directly correspond to the non-zero entries added during Gaussian elimination in Figure 1. **(b)** The permuted matrix $\mathbf{A}'$ and its associated graph $G(\mathbf{A}')$, obtained by reordering the rows and columns of $\mathbf{A}$ by moving the first row to be the last row, and the first column to be the last column (i.e. according to the permutation $\pi = (51234)$). Although the graph is structurally identical to $G(\mathbf{A})$, the new vertex ordering eliminates all variables without introducing any fill-in edges, showing how proper ordering prevents unnecessary fill-in.

and only if $i \neq j$ and there exists a path $v_0 = i, v_1, \ldots, v_k, v_{k+1} = j$ in $G(\mathbf{A})$ satisfying

$$\max_{1 \leq r \leq k} \pi(v_r) < \min(\pi(i), \pi(j)).$$

Unfortunately, while we can characterize the fill-in for any given ordering π, finding an optimal ordering π^* that minimizes the fill-in is known to be NP-Complete [55]. Moreover, several inapproximation results are known. For example in [9], the existence of polynomial time approximation schemes for this problem is ruled out, assuming $\mathbf{P} \neq \mathbf{NP}$, and the existence of a $2^{O(n^{1-\delta})}$-time approximation schemes for any positive δ is also ruled out, assuming the Exponential Time Hypothesis. Hence apriori, the problem might seems hopeless to tackle. Despite the theoretical computational difficulty of this problem, several heuristics, such as the Minimum Degree, Minimum Fill-In, and Nested Dissection heuristics, have been proposed and extensively studied in the literature [31, 17, 41, 42]. These heuristics remain widely used in practice for fill-in minimization and are surprisingly effective in practice, often being within a few percentages off from the optimal fill-in order.

Minimum Degree Heuristic, MDH This heuristic is a natural greedy algorithm, initially introduced by Markowitz [31], and was popularized by Rose in their PhD thesis. The algorithm is simple; it creates the elimination order π dynamically. It starts with $G^{(0)} \leftarrow G(\mathbf{A})$. In iteration $i \geq 1$, the algorithm picks the vertex $\pi(i)$ with minimum degree (ties broken arbitrarily) in $G^{(i-1)}$:

$$\pi(i) = \arg\min_{u \in G^{(i-1)}} \deg_{G^{(i-1)}}(u)$$

The algorithm then eliminates $\pi(i)$ (by making its neighbors a clique, adding potentially some fill-in edges $\mathcal{P}_\pi(i)$, then removing $\pi(i)$). Finally, it sets $G^{(i)} \leftarrow G^{(i-1)} \cup \mathcal{P}_\pi(i) \setminus \{\pi(i)\}$ and goes to iteration $i + 1$.

The algorithm can be implemented in $O(mn)$ time where m, n are the number of edges and vertices respectively [13]. Moreover, Cummings, Fahrbach, and Fatehpuria [13] showed that under the strong exponential time hypothesis, no $O(nm^{1-\epsilon})$ time algorithm exists for any $\epsilon > 0$.

The Minimum degree heuristic has given rise to *hundreds* of research papers on improving the running time of its practical implementations [2], and it is widely used and implemented in practice.

Minimum Fill-In Heuristic, MFillH This heuristic is also another natural greedy algorithm that also creates its elimination order π dynamically. It starts with $G^{(0)} \leftarrow G(\mathbf{A})$. In iteration $i \geq 1$, it picks the vertex $\pi(i)$ that minimizes the fill-in in $G^{(i-1)}$ of eliminating $\pi(i)$:

$$\begin{aligned} \textsc{FillIn}(u) = \Big| \big\{ (v,w) \mid v, w \in \mathcal{N}_{G^{(i-1)}}(u), \\ (v,w) \notin E\big(G^{(i-1)}\big) \big\} \Big| \\ \pi(i) = \underset{u \in G^{(i-1)}}{\arg\min}\, \textsc{FillIn}(u). \end{aligned}$$

The algorithm then eliminates $\pi(i)$. Finally, it sets $G^{(i)} \leftarrow G^{(i-1)} \cup \mathcal{P}_{\pi}(i) \setminus \{\pi(i)\}$ and goes to iteration $i+1$.

Importance of Tie-Breaking. The heuristics discussed above handle ties in an arbitrary manner; specifically, when multiple vertices have the same minimum degree or fill-in value, any one of them may be selected for elimination. Some research has focused on developing more sophisticated tie-breaking rules to enhance the effectiveness of these heuristics. For example, some methods combine multiple heuristics, using one heuristic to resolve ties determined by another, which can sometimes result in improved fill-in but does not consistently guarantee better fill-in [2, 28]. Additionally, the *Multiple Minimum Degree Heuristic* (MMDH) has been introduced, which involves removing all vertices that share the minimum degree simultaneously [28]. This approach strikes a balance between the aggressive removal of numerous vertices in a single step and the potential suboptimal fill-in introduced by making sequential, single-vertex decisions. By deleting multiple vertices at once, MMDH can reduce the overall computational complexity while maintaining a reasonable level of fill-in.

Nested Dissection Nested dissection is a powerful method that leverages graph separators to recursively decompose a graph into smaller, more manageable subproblems [17, 41, 42]. By identifying and removing a small "separator" set of vertices (or edges), the graph is partitioned into subgraphs that can be processed independently, after which the solutions are combined. This approach is especially efficient for classes of graphs where small separators are guaranteed to exist, such as planar graphs, thanks to well-known separator theorems.

Beyond its extensive use in practice for tasks like sparse matrix factorization, nested dissection also shines in theoretical settings; notably, the fastest known algorithm for maximum matching in planar graphs is built upon the framework of nested dissection [35], underscoring its fundamental importance in both theory and application.

How Do The Heuristics Compare? In practice, the minimum degree heuristic (MDH) often runs faster and is simpler to implement, since it only needs to identify the current vertex with the smallest degree at each step. By contrast, the minimum fill-in heuristic (MFillH) invests more computation to estimate which vertex's elimination would incur the fewest additional edges. While MFillH sometimes achieves smaller overall fill-in, this more sophisticated approach does not guarantee a strictly better ordering on every graph, and it can be significantly more expensive to run. Nested dissection, on the other hand, can deliver very effective results when a small separator is available or can be computed. However, finding such separators is not always straightforward for arbitrary graphs. Consequently, MDH tends to be the go-to choice for speed and ease of implementation, MFillH can yield sparser factorizations in select cases (albeit at greater cost), and nested dissection stands out when its underlying structural assumptions, such as the presence of small separators, are well satisfied. Regardless, all three heuristics usually get almost optimal fill-in on most graphs.

Motivation and Contributions Many real-world applications require solving linear systems repeatedly with identical sparsity patterns, or sparsity patterns drawn from the same distribution, such as in PDE simulations and real-time control, where even a slight improvement in fill-in reduction translates into considerable time and memory savings across *multiple solves*. Traditional heuristics like Minimum Degree work well for single-shot scenarios but fail to adapt to recurring solves, leaving room for improvement, particularly when the matrix structure is complex or irregular. For these multi-solve settings, existing methods cannot dynamically refine their strategies. Nested dissection offers strong performance for grid-like problems but relies on domain-specific insights about separators, making it unsuitable for general matrix classes.

Surprisingly, despite decades of research on fill-in minimization, no existing heuristic learns from repeated interactions with the matrix structure. Methods like Minimum Degree and Minimum Fill-In treat each solve independently, missing opportunities to refine the elimination order over time. Most importantly, they are as-is, their fill-in cannot be improved.

We address this challenge with ReFill, a reinforcement learning framework that combines Graph Neural Networks (GNNs) and a sequential decision-making process to dynamically learn effective elimination orders over repeated solves. Unlike traditional methods, ReFill adapts to the problem's structure by refining its strategy as it interacts with the matrix. Our experiments show that ReFill consistently outperforms classical heuristics, offering a promising, data-driven approach to a longstanding problem in scientific computing.

A Bird's-Eye View of Our Method At the heart of our approach is the idea of casting fill-in minimization as a reinforcement learning (RL) problem, where each sparse matrix (or equivalently its associated graph) becomes a separate "game", analogous to "atari game levels". The current graph structure, along with any vertices already eliminated, constitutes the game's "state," and each possible vertex elimination action yields a "cost" equal to the fill-in caused by eliminating that vertex. The agent attempts to learn an order dynamically, by making sequential elimination decisions at each iteration guided by a GNN.

In principle, any of the remaining vertices at a state could be eliminated at any given iteration. However, this leads to a blow up in size of the the action space which makes the sampling-complexity required to learn a "good" ordering high. To keep learning tractable despite the large action space, we employ a targeted action masking strategy that restricts choices to vertices that either have minimum degree or would induce minimum fill-in upon elimination. This is a combination of both the MDH and MFillH heuristics. Essentially, the agent is learning when to apply which heuristic on which graph, as well as the tie breaking rules using the Neural Network. This controlled subset of actions dramatically reduces sampling complexity necessary to learn good policies, while still preserving a high chance of including the truly optimal choice in most scenarios since these heuristics often perform exceptionally well and are near-optimal. We then train our policy network, implemented via Graph Convolutional Networks, using the Proximal Policy Optimization (PPO) algorithm, allowing it to systematically refine its choices to produce increasingly better elimination orderings.

Reinforcement Learning for Combinatorial Optimization. Reinforcement learning (RL) has emerged as a powerful paradigm for solving combinatorial optimization (CO) problems such as the Traveling Salesman Problem (TSP), knapsack, and vehicle routing. Traditional exact methods like branch-and-bound often fail to scale, while heuristic approaches require significant domain expertise. Recent work leverages RL to learn policies that iteratively construct solutions, achieving competitive results on benchmarks like the TSP [25], Scheduling [49] and many other problems; see a recent survey at [14]. Frameworks such as *OR-Gym* [52] standardize RL-based CO experimentation, enabling reproducible comparisons between learned policies and classical algorithms. Notably, graph neural networks (GNNs) have become instrumental in encoding combinatorial structures, with architectures like Graph Attention Networks (GATs) enabling RL agents to reason over graph-based CO problems [53, 21]. For example, [22] demonstrated that GNNs paired with RL can learn heuristics for graph traversal tasks, achieving near-optimal solutions on Maximum Cut instances with 100 nodes.

Surveys emphasize the growing adoption of RL-GNN frameworks [14], particularly for their ability to generalize across problem sizes and constraints [33, 4, 32]. Despite progress, challenges persist in reward shaping for sparse CO environments and scaling to industrial-scale instances.

Organization Section 2 presents a review of related work, focusing primarily on reinforcement learning, graph neural networks, and Proximal Policy Optimization (PPO). Readers already familiar with these topics may wish to skip directly to Section 3, where we formally describe our new RL-based framework for fill-in minimization and detail the underlying architecture. Section 4 provides experimental results that demonstrate the effectiveness of the proposed method, and Section 5 offers future work and limitations.

2 Related Work

Beside the related work on fill-in minimization discussed in the introduction, we briefly recap known results on Graph Neural Networks (GNNs), Markov Decision Processes (MDPs), and the PPO optimization algorithm. If the reader is familiar with said topics, they are encouraged to skip this section.

Evolution of Graph Neural Networks. The emergence of graph neural networks (GNNs) has revolutionized machine learning on non-Euclidean data through their ability to capture complex relational patterns [54, 16, 59]. This paradigm shift has spawned diverse architectural families: recurrent frameworks that propagate states through iterative refinement [27, 37], spectral-spatial convolutional operators [58, 24], reconstruction-driven auto-encoders [23, 56], attention-based neighborhood weighting mechanisms [50, 53], and transformer-inspired architectures [57, 36].

These architectures power a wide application landscape spanning structural analysis tasks like node classification [20, 29, 3] and link prediction [10, 46, 60], to whole-graph characterization for molecular property prediction [39, 18, 1] and community detection [11, 26, 30]. Such versatility stems from their foundational mechanism: learnable message passing that recursively transforms node representations through localized information aggregation [44].

Consider an undirected graph $G = (V, E)$ with adjacency matrix $\mathbf{A} \in \{0,1\}^{|V|\times|V|}$. The message propagation dynamics can be formalized through layer-wise updates:

$$\boldsymbol{H}^{(k+1)} = \sigma\left(\tilde{\boldsymbol{A}}\boldsymbol{H}^{(k)}\boldsymbol{W}^{(k)}\right),$$

where $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$ introduces self-connections, $\boldsymbol{W}^{(k)}$ represents learnable parameters, and σ denotes nonlinear activation. This operation induces local node update rules:

$$h_i^{(k+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)} \boldsymbol{W}^{(k)} h_j^{(k)}\right),$$

with $\mathcal{N}(i)$ denoting node i's neighborhood. Modern implementations generalize this through learnable aggregation functions:

$$h_i^{(k+1)} = \mathrm{MLP}^{(k)}\left(h_i^{(k)}, \bigoplus_{j\in\mathcal{N}(i)} \phi(h_i^{(k)}, h_j^{(k)}, e_{ij})\right),$$

where $\bigoplus$ represents permutation-invariant aggregation (e.g., sum, mean, or max) and ϕ encodes edge features when available. This propagation enables GNNs to develop good representations while preserving structural relationships.

Reinforcement Learning in Games. Reinforcement Learning (RL) has demonstrated remarkable success in complex game environments, wherein an autonomous agent interacts with an environment (or game) by observing states, taking actions, and receiving rewards [51, 34]. Formally, RL problems are often framed as a Markov Decision Process (MDP), defined by a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, a transition distribution $p(s_{t+1} \mid s_t, a_t)$, and a reward function $r(s_t, a_t)$. In the context of *games*, the *observation space* (or state space) might include raw pixels (in video games), board configurations (in board games), or symbolic features (in card games). The *action space* typically comprises valid moves or control signals (e.g. joystick commands in Atari games, or placing a stone on a board in Go). A *reward* is then provided by the environment based on the outcome of each action, for example, immediate point increments in Atari games or win/loss signals at the end of a board game [47]. By learning a policy that maps observations to actions with the objective of *maximizing* expected cumulative reward, RL agents have achieved superhuman performance in diverse domains, such as Atari games [34] and the game of Go [48].

Proximal Policy Optimization (PPO). Among numerous RL algorithms, *Proximal Policy Optimization (PPO)* [45] is a popular policy gradient method that balances sample efficiency, stability, and ease of implementation. It is often used to learn neural network policies for RL algorithms. PPO alternates between sampling data through interaction with the environment and optimizing a clipped objective function that constrains large policy updates. Specifically, PPO uses a *clipped* surrogate objective,

$$L^{\mathrm{CLIP}}(\theta) = \mathbb{E}_t\Big[\min\big(r_t(\theta)\hat{A}_t,\ \mathrm{clip}\big(r_t(\theta), 1-\epsilon, 1+\epsilon\big)\hat{A}_t\big)\Big],$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t|s_t)}$ is the probability ratio of the new policy π_θ to the old policy $\pi_{\theta_{\mathrm{old}}}$, and $\hat{A}_t$ is an estimator of the advantage function.

By clipping the probability ratio $r_t(\theta)$ to the interval $[1-\epsilon,\ 1+\epsilon]$, PPO avoids excessively large policy updates, thus preventing instability or catastrophic performance collapses. This mechanism has made PPO a standard choice for many benchmark tasks in continuous control, robotics, and game-based RL environments. In this paper, we use the MaskedPPO implementation which allows action masking by sampling only from "valid" actions in each state [19].

3 ReFill

Problem Statement. The problem is modeled as a sequential decision-making task. Given the matrix $\mathbf{A}$ and an ordering π of the vertices of $G(\mathbf{A})$, let $\textsc{FillInCost}(G(\mathbf{A}), \pi)$ denote the total fill-in cost from eliminating the vertices in order $\pi(1), \ldots, \pi(n)$. The NP-hard optimization problem is

$$\min_{\pi\in\mathbb{S}_n} \textsc{FillInCost}(G(\mathbf{A}), \pi). \tag{1}$$

Sequential Decision-Making. Instead of learning the entire order at once, we aim to learn a policy $\pi^*(u \mid G) \in (0,1)$, which gives the probability of eliminating a vertex $u \in V(G)$. For any graph G, this policy determines the next vertex to eliminate, ensuring that the permutation π^* for Eq. (1) can be recovered (breaking ties arbitrarily).

Supervised Learning Challenges. Developing data-driven heuristics to outperform both the minimum degree (MDH) and minimum fill-in (MFillH) heuristics is inherently challenging due to the lack of high-quality training data. While the minimum degree heuristic is computationally efficient and often produces near-optimal results, there exist graph instances where the optimal elimination order deviates from MDH or MFillH. To train a model capable of identifying such deviations, we would require representative graphs with known optimal elimination orders that highlight scenarios where neither heuristic suffices. However, obtaining such data is fundamentally difficult, as computing the optimal solution for fill-in minimization is NP-hard. This computational barrier makes it infeasible to generate large-scale datasets of optimal or near-optimal solutions. Consequently, this lack of accessible ground truth poses a bottleneck in leveraging supervised learning approaches to learn policies that generalize beyond existing heuristics.

First Attempt: Reinforcement Learning Over All Actions. One way we can tackle this problem is using reinforcement learning with a massive action and state space. The states correspond to the current graph, and the action at each step is to select a node for elimination. The reward for each action is defined as the negative of the number of fill-in edges added by eliminating the chosen node. The objective is to find an elimination sequence that maximizes the sum of rewards; effectively minimizing the total fill-in. However, the action space is combinatorially large, as it includes all possible nodes that can be selected at each step, and the state space, defined by the set of possible graphs resulting from different elimination orders, grows exponentially with the number of vertices. This makes the training very computationally expensive and intractable from a sampling-complexity point of view. It also makes it more likely that the agent can get stuck in a local optima elimination order.

A Glimmer Of Hope. While this combinatorial explosion poses a significant challenge, a glimmer of hope is that traditional heuristics like MDH and MFillH, and their tie-breaking rules, are extremely **local heuristics**. These heuristics make locally optimal choices based on simple local graph properties, such as a node's degree and its neighbors degrees. Despite this, these methods do extremely well in practice. So one would hope that a shallow GNN using the message passing mechanism of information from its "nearby" vertices can learn a more clever heuristic. Moreover, one would hope that this heuristic is **graph-size agnostic**. Specifically, it calculates a "goodness" score for each node based on it's nearby neighborhood, and finally take the node with the best goodness score with a softmax layer.

ReFill. The previous discussion points to a natural solution framework. The environment is initialized with the input graph for which we are trying to find a good elimination order. At any given time, the agent evaluates the graph's current state, which is derived from the original graph by eliminating vertices. To make an elimination decision, a graph neural network (GNN) processes the node features producing a "goodness" score for each node that reflects its suitability for elimination. The GNN is trained to learn these scores by leveraging local graph properties through message passing. It should also be noted that we are not explicitly deleting a node during elimination; we are simply masking over the decision of deleting it again using the deleted mask stored in the observation space. The "game" ends when all vertices have been deleted (i.e. the deleted mask is all ones).

Table 1: ReFill Fill-in compared to MDH and MFillH on all datasets. Percentages reported are $\frac{\text{MDH}-\text{ReFill}}{\text{MDH}}$ and $\frac{\text{MFillH}-\text{ReFill}}{\text{MFillH}}$. Positive values indicate ReFill outperforming the heuristic.

Graph	V	E	ReFill	MDH	MFillH
Grid 5x5	25	40	37	0.0%	0.0%
Grid 6x6	36	60	69	2.8%	2.8%
Grid 7x7	49	84	111	6.7%	1.8%
Grid 8x8	64	112	166	9.3%	9.8%
Grid 9x9	81	144	240	10.4%	0.8%
Grid 10x10	100	180	325	13.6%	7.4%
2.graph	129	4943	195	2.5%	7.1%
3.graph	101	840	286	12.0%	0.0%
11.graph	126	1095	183	6.2%	3.2%
13.graph	119	161	92	0.0%	1.1%
18.graph	150	259	105	2.8%	5.4%
23.graph	200	661	799	4.9%	5.4%
26.graph	120	4904	229	9.8%	0.9%
40.graph	147	7303	352	6.6%	3.3%
92.graph	132	255	202	4.7%	-1.5%
99.graph	166	396	382	18.6%	5.0%
100.graph	152	377	360	6.0%	1.1%

The action space is constrained by masking out nodes that do not satisfy key heuristic properties, such as having the minimum degree or inducing the least fill-in. This dramatically reduces the action space's size, allowing the reinforcement learning (RL) agent to focus on decisions that are more likely to yield optimal or near-optimal results. *Essentially, the agent is learning when to apply which heuristic based on the graph properties.*

The policy is optimized using the Proximal Policy Optimization (PPO) algorithm. PPO balances exploration and exploitation by limiting large updates to the policy. The reward signal is the negative of the fill-in edges added during the elimination step, encouraging the agent to minimize fill-in over the sequence of elimination steps. This iterative process enables the RL agent to refine its elimination policy through repeated interactions with the graph. Figure 3 summarizes the architecture and training process for ReFill.

4 Experiments

In this section, we evaluate the performance of our reinforcement learning (RL)-based algorithm, ReFill. We compare our method against the established minimum degree heuristic (MDH) and the minimum fill-in heuristic (MFillH) on both synthetic and real-world graphs. We also present an analysis of training dynamics, fill-in cost, and briefly discuss ablation and runtime overhead.

4.1 Experimental Setup

Graph Instances. We evaluate our methods on the following graph classes:

Grid graph. We use the $N = n \times n$ grid graph for $5 \leq n \leq 10$. The $n \times n$ grid system occurs naturally when solving differential equations with finite differences method; it was the initial motivation for introducing the nested dissection heuristic [17].

PACE 2017 Track-B Public Dataset. The goal of the yearly PACE challenge is to investigate the applicability of algorithmic ideas studied and developed in the subfields of multivariate, fine-grained, parameterized, or fixed-parameter tractable algorithms. The objective of the PACE 2017 Track-B challenge [15] was to solve the NP-hard Minimum Fill-In problem **exactly** (i.e., no heuristics) within 30 minutes on real-world data. The test datasets were curated by the authors from multiple sources. For optimizing Gaussian elimination, they selected instances from the Matrix Market [6] and the Network Repository [43], where exact solutions could be computed. We use 11 medium-sized graphs from the public datasets, N.GRAPH for $N \in \{2, 3, 11, 13, 18, 23, 26, 40, 92, 99, 100\}$. Table 1 shows the number of vertices and edges for each graph.

Generalization Experiment. In addition, to test the generalization of the model, we conduct the following experiment. We sample 35 graphs from $G(50, 0.2)$, where $G(n, p)$ is the Erdős–Rényi graph with 50 vertices and each (undirected) edge selected with probability 0.2. We train the GNN on all these graphs *simultaneously* using action masking, then evaluate the trained heuristic on 200 *new* $G(50, 0.2)$ graphs, and report the average improvement using the learned heuristic. We emphasize this is using a **single set** of learned parameters, and not one per graph.

Evaluation Metrics. For every graph, we measure the total fill-in cost, defined as the number of missing edges that are added when deleting nodes in a particular order. We select the best elimination order found by our RL algorithm. These costs are then compared to MDH and MFillH.

4.1.1 Implementation Details

Our RL model uses a policy network based on a 2-layer graph convolutional network (GCN) with average aggregation. Each node has three features: (i) the normalized degree in the current graph, (ii) the fill-in that would be introduced by deleting the node, and (iii) whether the node is already eliminated. We train using PPO with masking [19] for 500,000 timesteps, where a timestep here represents the elimination of a single vertex. See Appendix A for the remaining implementation details.

4.2 Performance on Grid Graphs and PACE Dataset

Table 1 summarizes the total fill-in cost for ReFill, MDH, and MFillH on both grid graphs and the PACE dataset. Except for the 5×5 grid (where ReFill matches both heuristics), ReFill outperforms them on all grid sizes by up to 13.6% (vs. MDH) and 9.8% (vs. MFillH).

On the PACE dataset, ReFill generally achieves better fill-in costs than MDH (up to 18.6% improvement) and MFillH (up to 5.4%), except on 92.graph, where ReFill underperforms MFillH by 1.5%. One of the reasons that ReFill underperforms on 92.GRAPH was that during elimination, the number of vertices with minimum degree or minimum fill-in were comparatively high, which leads to the same action space explosion without masking. This often caused the agent to get stuck at a local optima.

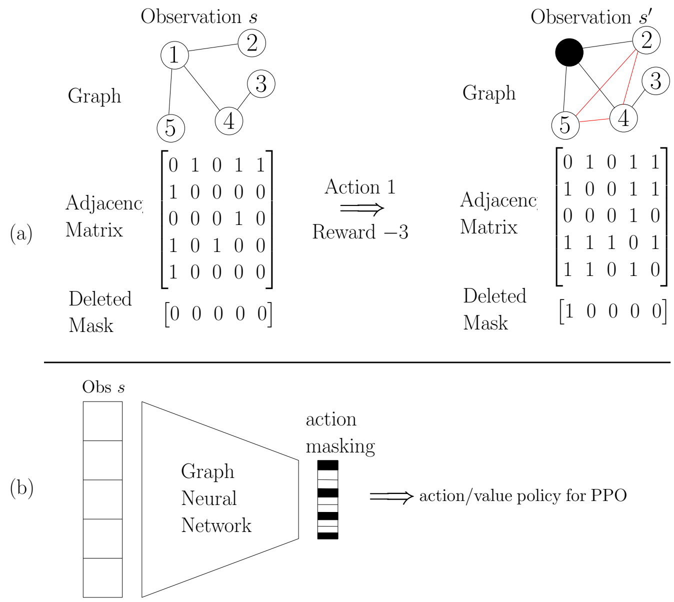

Figure 3: ReFill reinforcement learning environment. (a) An example observation s is updated by deleting vertex 1, introducing 3 fill-in edges (-3 reward). (b) Schematic of the RL loop using a GNN and masked PPO.

4.3 Generalization Experiment

Appendix D discusses the generalization experiment in more details. On average, ReFill learns elimination heuristics that give an elimination order that is a 2.21% improvement over MDH and 1.05% improvement over MFillH.

4.4 Ablation Study on Effect of Masking

Figure 4 shows the effect of masking vs not-masking the action space on the fill-in reached by for the graph Grid 8×8. For an ablation study on the effect of masking on training, see Appendix C. As seen, not only does masking help with significantly faster convergence to better fill-in than both heuristics, not masking can also cause the agent to get stuck in a local-minima elimination ordering that is far from the optimal solution.

4.5 Runtime Overhead

ReFill has a one-time training cost but infers node orderings efficiently once trained. In contrast, MDH and MFillH require negligible setup but can be less accurate. For repeated solves on similar graphs, the amortized cost of ReFill becomes favorable. Throughout, training took less than 30 minutes, with most graphs training in 15 minutes.

As for the generalization experiment, after training, the inference speed of all heuristics were comparable, which further reinforces that GNN-based elimination orders can hopefully be deployed in practical solvers in the future.

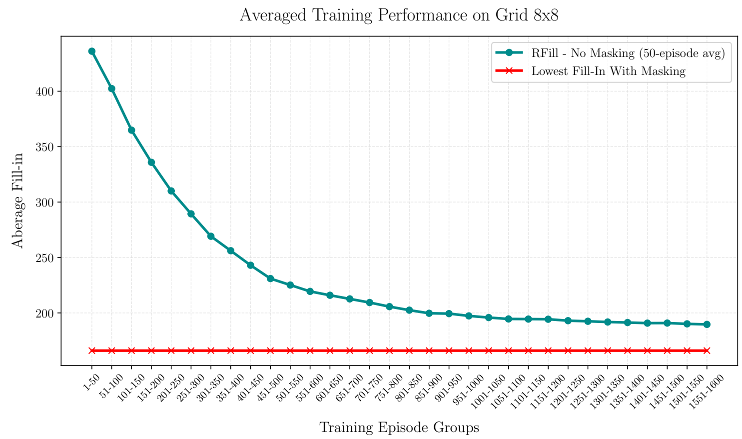

Figure 4: Average fill-in (lower is better) on 8×8 grid, comparing masking vs. no masking during training.

4.6 Summary

Overall, ReFill demonstrates that a tailored RL approach with GCN-based policies and action masking can outperform standard heuristics on both grid graphs and real-world PACE instances. Masking notably improves convergence and solution quality, while moderate GCN depth and feature dimension suffice to capture the local graph information crucial for effective elimination orders. Once trained, ReFill generates improved orderings with minimal inference cost, and can generalize on new unseen graph instances drawn from the same graph distribution as the training data.

5 Future Work and Limitations

While our results demonstrate the promise of learning-based methods for fill-in minimization, several challenges remain. First, deploying these methods in practice, particularly in single-shot settings where a solver must handle arbitrary matrices without prior training, requires a lightweight, generalizable GCN model with a *single set* of parameters. However, training such a model across diverse graphs proved unstable, likely due to structural variations between datasets that hinder consistent learning. However, as we showed, when the graphs are sampled from the *same distribution* (say $G(n, p)$), then it is possible to learn a single set of parameters that outperforms both heuristics on average. Future research could explore transfer learning or domain adaptation techniques to stabilize training across graph distributions. Second, while masking actions to prioritize minimum degree or fill-in candidates accelerates convergence, it risks excluding elimination orders that diverge from these heuristics. Yet, removing masking often traps the agent in local minima as shown in our ablation studies. Developing adaptive masking strategies that balance exploration and exploitation, for instance, by gradually relaxing masking constraints as training progresses, could mitigate this trade-off. Finally, scalability remains a critical barrier. Extending this work to million-scale systems will require more scalable GNN architectures and integration with distributed optimizers and solvers. Addressing these challenges will be essential to bridge the gap between theoretical advances and real-world deployment.

References

[1] Han Altae-Tran, Bharath Ramsundar, Aneesh S. Pappu, and Vijay S. Pande. Low data drug discovery with one-shot learning. *CoRR*, abs/1611.03199, 2016.

[2] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[3] Ahmed Begga, Francisco Escolano, Miguel Angel Lozano, and Edwin R. Hancock. Diffusion-jump gnns: Homophiliation via learnable metric filters. *CoRR*, abs/2306.16976, 2023.

[4] 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.

[5] Ivan Bliznets, Marek Cygan, Paweł Komosa, Michał Pilipczuk, and Lukáš Mach. Lower bounds for the parameterized complexity of minimum fill-in and other completion problems. *ACM Transactions on Algorithms (TALG)*, 16(2):1–31, 2020.

[6] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix market: a web resource for test matrix collections. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement*, page 125–137, GBR, 1997. Chapman & Hall, Ltd.

[7] Matthias Bollhöfer, Olaf Schenk, Radim Janalik, Steve Hamm, and Kiran Gullapalli. State-of-the-art sparse direct solvers. *Parallel algorithms in computational science and engineering*, pages 3–33, 2020.

[8] Claude Brezinski, Gérard Meurant, and Michela Redivo-Zaglia. *A Journey through the History of Numerical Linear Algebra*. SIAM, 2022.

[9] Yixin Cao and R.B. Sandeep. Minimum fill-in: Inapproximability and almost tight lower bounds. *Information and Computation*, 271:104514, 2020.

[10] Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. Simple and deep graph convolutional networks. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 1725–1735. PMLR, 2020.

[11] Zhengdao Chen, Lisha Li, and Joan Bruna. Supervised community detection with line graph neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[12] Christophe Crespelle, Pål Grønås Drange, Fedor V Fomin, and Petr Golovach. A survey of parameterized algorithms and the complexity of edge modification. *Computer Science Review*, 48:100556, 2023.

[13] Robert Cummings, Matthew Fahrbach, and Animesh Fatehpuria. A fast minimum degree algorithm and matching lower bound. In *Proceedings of the Thirty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '21, page 724–734, USA, 2021. Society for Industrial and Applied Mathematics.

[14] Victor-Alexandru Darvariu, Stephen Hailes, and Mirco Musolesi. Graph reinforcement learning for combinatorial optimization: A survey and unifying perspective, 2024.

[15] Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In Daniel Lokshtanov and Naomi Nishimura, editors, *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*, volume 89 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[16] Vijay Prakash Dwivedi, Chaitanya K. Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *CoRR*, abs/2003.00982, 2020.

[17] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[18] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay S. Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[19] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. 2020.

[20] Mohammad Rasool Izadi, Yihao Fang, Robert Stevenson, and Lizhen Lin. Optimization of graph neural networks with natural gradient descent. In Xintao Wu, Chris Jermaine, Li Xiong, Xiaohua Hu, Olivera Kotevska, Siyuan Lu, Weija Xu, Srinivas Aluru, Chengxiang Zhai, Eyhab Al-Masri, Zhiyuan Chen, and Jeff Saltz, editors, *2020 IEEE International Conference on Big Data (IEEE BigData 2020), Atlanta, GA, USA, December 10-13, 2020*, pages 171–179. IEEE, 2020.

[21] Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*, 2019.

[22] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

[23] Thomas N. Kipf and Max Welling. Variational graph auto-encoders. *CoRR*, abs/1611.07308, 2016.

[24] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[25] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2019.

[26] Ron Levie, Federico Monti, Xavier Bresson, and Michael M. Bronstein. Cayleynets: Graph convolutional neural networks with complex rational spectral filters. *IEEE Trans. Signal Process.*, 67(1):97–109, 2019.

[27] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[28] Joseph W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Softw.*, 11(2):141–153, June 1985.

[29] Sitao Luan, Chenqing Hua, Qincheng Lu, Jiaqi Zhu, Mingde Zhao, Shuyuan Zhang, Xiao-Wen Chang, and Doina Precup. Is heterophily A real nightmare for graph neural networks to do node classification? *CoRR*, abs/2109.05641, 2021.

[30] Yao Ma, Ziyi Guo, Zhaochun Ren, Jiliang Tang, and Dawei Yin. Streaming graph neural networks. In Jimmy X. Huang, Yi Chang, Xueqi Cheng, Jaap Kamps, Vanessa Murdock, Ji-Rong Wen, and Yiqun Liu, editors, *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*, pages 719–728. ACM, 2020.

[31] Harry M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3(3):255–269, 1957.

[32] Nina Mazyavkina, Sergei Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 134:105400, 2021.

[33] Abbas Mehrabian, Ankit Anand, Hyunjik Kim, Nicolas Sonnerat, Matej Balog, Gheorghe Comanici, Tudor Berariu, Andrew Lee, Anian Ruoss, Anna Bulanova, Daniel Toyama, Sam Blackwell, Bernardino Romera Paredes, Petar Veličković, Laurent Orseau, Joonkyung Lee, Anurag Murty Naredla, Doina Precup, and Adam Zsolt Wagner. Finding increasingly large extremal graphs with alphazero and tabu search, 2023.

[34] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. In *Nature*, volume 518, pages 529–533, 2015.

[35] Marcin Mucha and Piotr Sankowski. Maximum matchings in planar graphs via gaussian elimination. *Algorithmica*, 45(1):3–20, May 2006.

[36] Luis Müller, Mikhail Galkin, Christopher Morris, and Ladislav Rampásek. Attending to graph transformers. *CoRR*, abs/2302.04181, 2023.

[37] Andrei Liviu Nicolicioiu, Iulia Duta, and Marius Leordeanu. Recurrent space-time graph neural networks. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 12818–12830, 2019.

[38] 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.

[39] Yu Rong, Yatao Bian, Tingyang Xu, Weiyang Xie, Ying Wei, Wenbing Huang, and Junzhou Huang. Self-supervised graph transformer on large-scale molecular data. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[40] Donald J Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597–609, 1970.

[41] Donald J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In RONALD C. READ, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, 1972.

[42] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.

[43] Ryan A. Rossi and Nesreen K. Ahmed. Networkrepository: An interactive data repository with multi-scale visual analytics, 2014.

[44] 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.

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

[46] Hamed Shirzad, Ameya Velingker, Balaji Venkatachalam, Danica J. Sutherland, and Ali Kemal Sinop. Exphormer: Sparse transformers for graphs. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 31613–31632. PMLR, 2023.

[47] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[48] 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.

[49] Igor G. Smit, Jianan Zhou, Robbert Reijnen, Yaoxin Wu, Jian Chen, Cong Zhang, Zaharah Bukhsh, Yingqian Zhang, and Wim Nuijten. Graph neural networks for job shop scheduling problems: A survey, 2024.

[50] Chengcheng Sun, Chenhao Li, Xiang Lin, Tianji Zheng, Fanrong Meng, Xiaobin Rui, and Zhixiao Wang. Attention-based graph neural networks: a survey. *Artif. Intell. Rev.*, 56(S2):2263–2310, 2023.

[51] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[52] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Or-gym: A reinforcement learning library for operations research problems. In *NeurIPS Datasets and Benchmarks*, 2021.

[53] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. *CoRR*, abs/1710.10903, 2017.

[54] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Trans. Neural Networks Learn. Syst.*, 32(1):4–24, 2021.

[55] Mihalis Yannakakis. Computing the minimum fill-in is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79, 1981.

[56] Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and Jure Leskovec. Graphrnn: A deep generative model for graphs. *CoRR*, abs/1802.08773, 2018.

[57] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J. Kim. Graph transformer networks. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 11960–11970, 2019.

[58] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: Algorithms, applications and open challenges. In Xuemin Chen, Arunabha Sen, Wei Wayne Li, and My T. Thai, editors, *Computational Data and Social Networks - 7th International Conference, CSoNet 2018, Shanghai, China, December 18-20, 2018, Proceedings*, volume 11280 of *Lecture Notes in Computer Science*, pages 79–91. Springer, 2018.

[59] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE Trans. Knowl. Data Eng.*, 34(1):249–270, 2022.

[60] Zhaocheng Zhu, Zuobai Zhang, Louis-Pascal A. C. Xhonneux, and Jian Tang. Neural bellman-ford networks: A general graph neural network framework for link prediction. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 29476–29490, 2021.

A Implementation Details

The learning rate is set to 1×10^{-4} or 5×10^{-5} depending on graph size. Hyperparameters are detailed in Appendix B; our code is in the supplemental section. Experiments are performed on a single machine with an NVIDIA A100-SXM4-80GB GPU (226 GiB system memory). We run 5 parallel environments to collect experiences using stable baselines3 [38] with maskable PPO [19]. The GCN's hidden feature dimension ranges between 8 and 32. The network outputs a single scalar score per node, used for action selection.

B Hypterparameter values used.

Here, we describe all the commands we used to run the experiments, which include all the hyperparameters. The figures in Table 1 were calculated using the main.py script with –action_masking set to 1, –policy_sizes set to an empty list, –total_timesteps set to 500,000, –parallel_envs set to 5, and the following hyperparameters.

Table 2: Hyperparameters used for ReFill.

Graph	**total_timesteps**	**parallel_envs**	**learning_rate**	**node_dim**
Grid 5x5	500000	5	0.0001	32
Grid 6x6	500000	5	0.0001	32
Grid 7x7	500000	5	0.0001	32
Grid 8x8	500000	5	0.00005	32
Grid 9x9	500000	5	0.00005	16
Grid 10x10	500000	5	0.00005	8
2.graph	500000	5	0.00005	16
3.graph	500000	5	0.0001	32
11.graph	500000	5	0.00005	32
13.graph	500000	5	0.00005	16
18.graph	500000	5	0.00005	16
23.graph	500000	5	0.00005	16
26.graph	500000	5	0.00005	16
40.graph	500000	5	0.00005	16
92.graph	500000	5	0.00005	16
99.graph	500000	5	0.00005	16
100.graph	500000	5	0.00005	16

C Non-Masking Ablation Study

The following ablation study for non-masking was performed using the command preceding the figure. Usually, the agent got close to the optimal fill-in, but gets stuck in a local optima without using action-masking.

```
$ python main.py datasets/grid.n8.graph --output_file non_masking_results/grid.n8.graph
--policy_sizes 16 16 --total_timesteps 500_000 --learning_rate 0.0001
--parallel_envs 5 --node_dim 8 --ent_coef 0.002 --action_masking 0
```

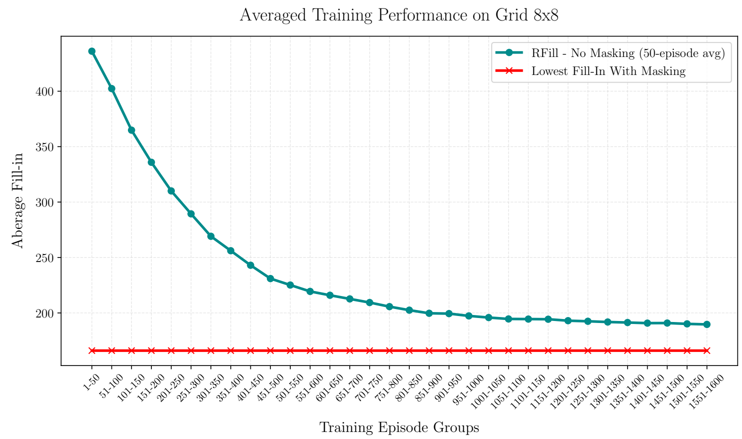

Figure 5: Average fill-in (lower is better) on 8×8 grid, comparing masking vs. no masking during training.

```
$ python main.py datasets/grid.n9.graph --output_file non_masking_results/grid.n9.graph
--policy_sizes 32 32 --total_timesteps 500_000 --learning_rate 0.0001
--parallel_envs 5 --node_dim 16 --ent_coef 0.002 --action_masking 0
```

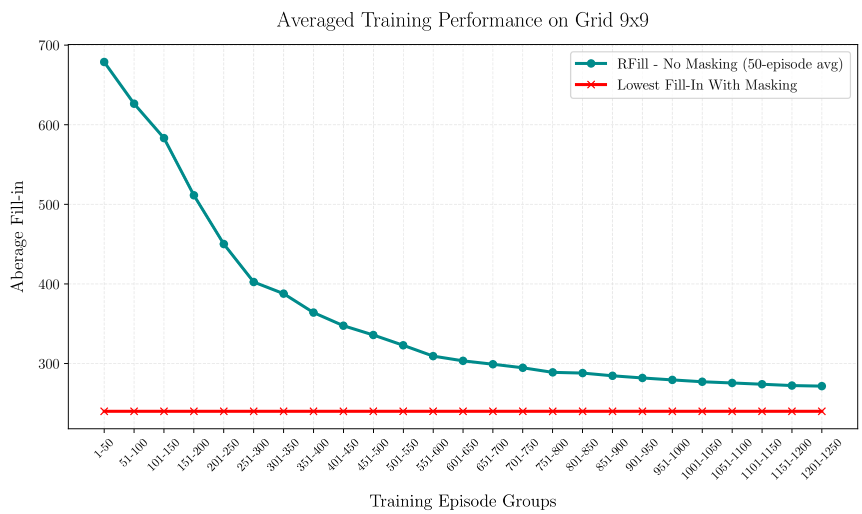

Figure 6: Average fill-in (lower is better) on 9×9 grid, comparing masking vs. no masking during training.

```
$ python main.py datasets/grid.n10.graph --output_file non_masking_results/grid.n10.graph
--policy_sizes 32 32 --total_timesteps 500_000 --learning_rate 0.0001
--parallel_envs 5 --node_dim 16 --ent_coef 0.002 --action_masking 0
```

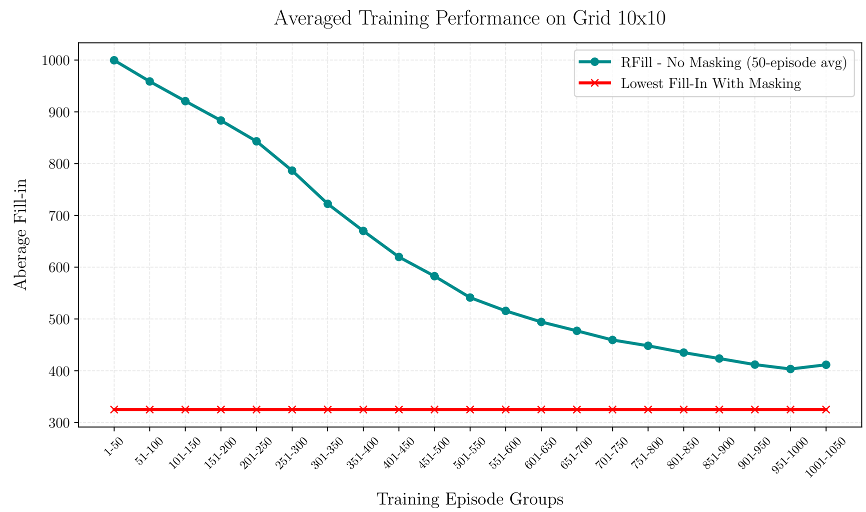

Figure 7: Average fill-in (lower is better) on 10×10 grid, comparing masking vs. no masking during training.

```
$ python main.py datasets/2.graph --output_file non_masking_results/2.graph
--policy_sizes 16 16 --total_timesteps 500_000 --learning_rate 0.0001
--parallel_envs 5 --node_dim 8 --ent_coef 0.002 --action_masking 0
```

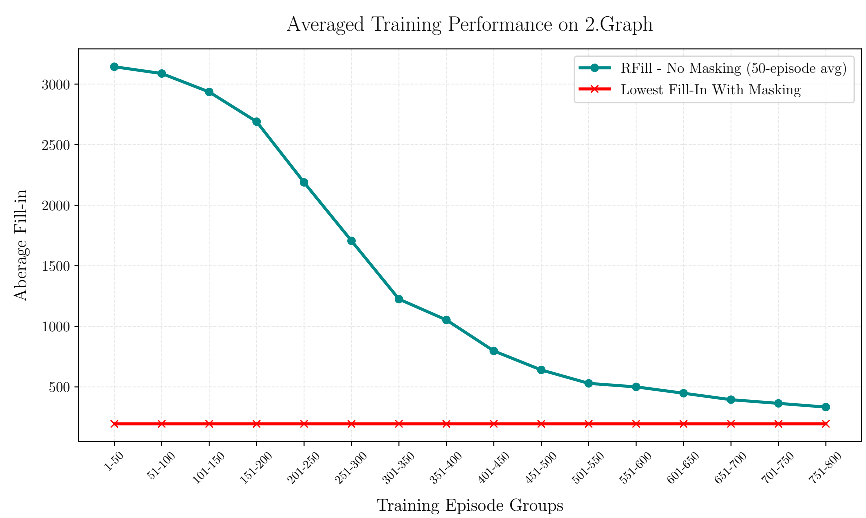

Figure 8: Average fill-in (lower is better) on 2.GRAPH, comparing masking vs. no masking during training.

```
$ python main.py datasets/3.graph --output_file non_masking_results/3.graph
--policy_sizes 32 32 --total_timesteps 500_000 --learning_rate 0.0001
```

```
--parallel_envs 5 --node_dim 16 --ent_coef 0.002 --action_masking 0
```

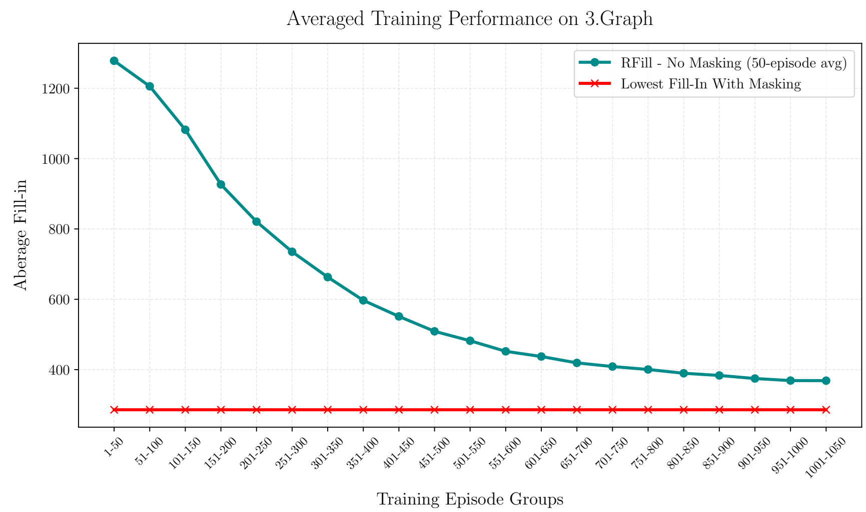

Figure 9: Average fill-in (lower is better) on 3.GRAPH, comparing masking vs. no masking during training.

```
$ python main.py datasets/11.graph --output_file non_masking_results/11.graph
--policy_sizes 32 32 --total_timesteps 500_000 --learning_rate 0.0001
--parallel_envs 5 --node_dim 16 --ent_coef 0.002 --action_masking 0
```

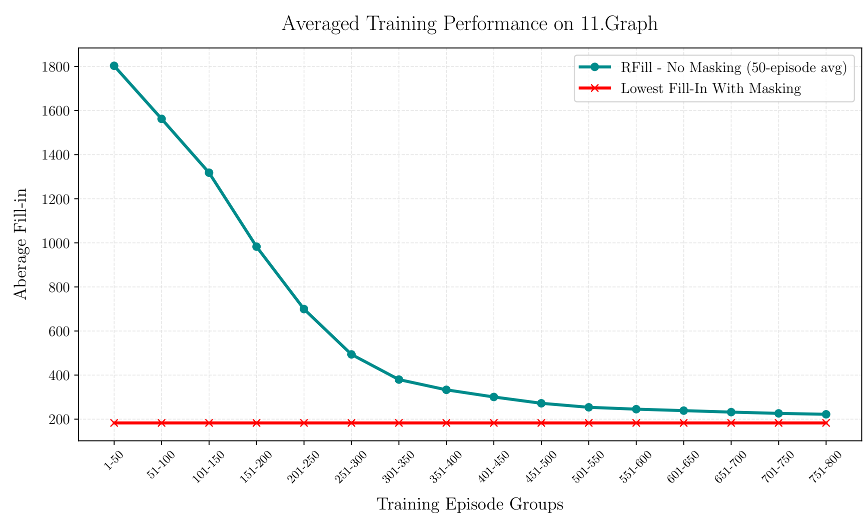

Figure 10: Average fill-in (lower is better) on 11.GRAPH, comparing masking vs. no masking during training.

```
$ python main.py datasets/13.graph --output_file non_masking_results/13.graph
--policy_sizes 32 32 --total_timesteps 500_000 --learning_rate 0.0001
--parallel_envs 5 --node_dim 16 --ent_coef 0.002 --action_masking 0
```

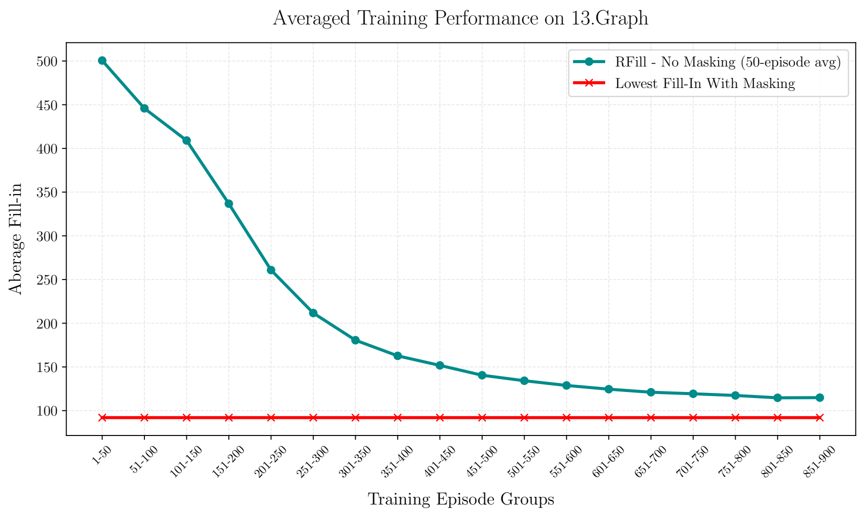

Figure 11: Average fill-in (lower is better) on 13.GRAPH, comparing masking vs. no masking during training.

```
$ python main.py datasets/18.graph --output_file non_masking_results/18.graph
--policy_sizes 32 32 --total_timesteps 500_000 --learning_rate 0.0001
--parallel_envs 5 --node_dim 16 --ent_coef 0.002 --action_masking 0
```

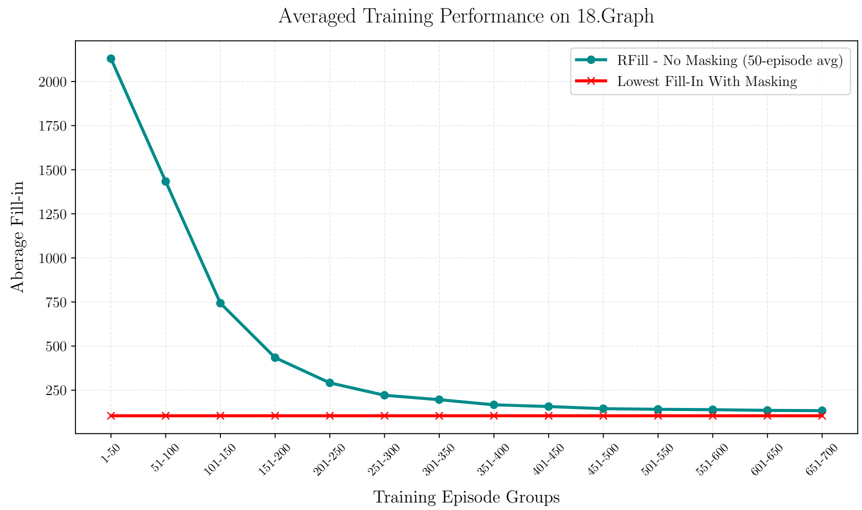

Figure 12: Average fill-in (lower is better) on 18.GRAPH, comparing masking vs. no masking during training.

```
$ python main.py datasets/23.graph --output_file non_masking_results/23.graph
--policy_sizes 64 32 --total_timesteps 1_000_000 --learning_rate 0.0001
--parallel_envs 10 --node_dim 16 --ent_coef 0.002 --action_masking 0
```

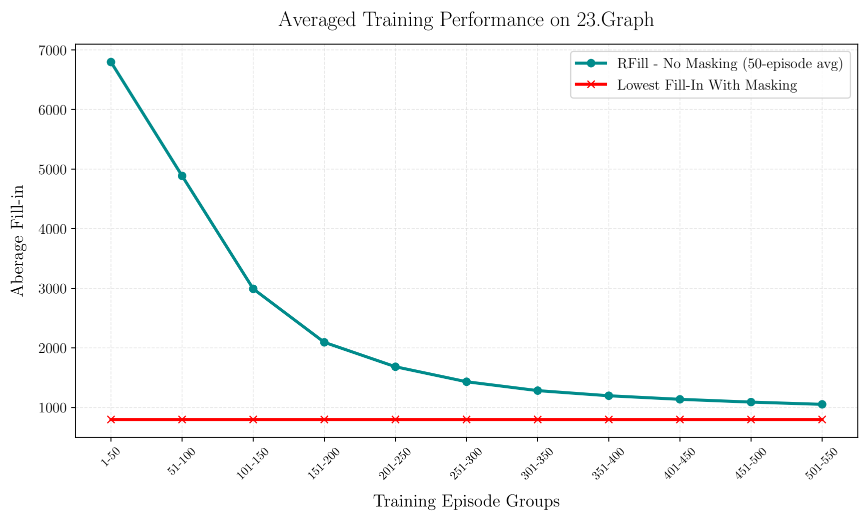

Figure 13: Average fill-in (lower is better) on 23.GRAPH, comparing masking vs. no masking during training.

```
$ python main.py datasets/26.graph --output_file non_masking_results/26.graph
--policy_sizes 32 32 --total_timesteps 500_000 --learning_rate 0.0001
```

```
--parallel_envs 5 --node_dim 16 --ent_coef 0.002 --action_masking 0
```

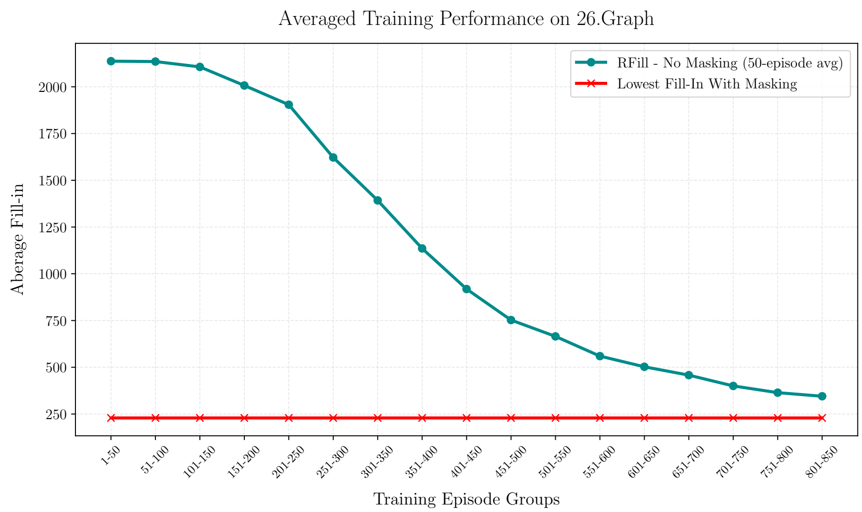

Figure 14: Average fill-in (lower is better) on 26.GRAPH, comparing masking vs. no masking during training.

```
$ python main.py datasets/40.graph --output_file non_masking_results/40.graph
--policy_sizes 64 32 --total_timesteps 1_000_000 --learning_rate 0.0001
--parallel_envs 10 --node_dim 16 --ent_coef 0.002 --action_masking 0
```

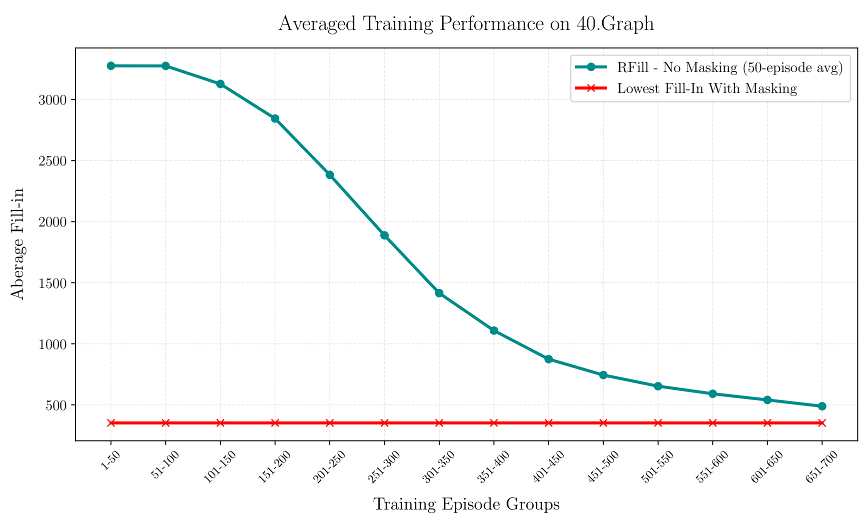

Figure 15: Average fill-in (lower is better) on 40.GRAPH, comparing masking vs. no masking during training.

```
$ python main.py datasets/92.graph --output_file non_masking_results/92.graph
--policy_sizes 64 32 --total_timesteps 1_000_000 --learning_rate 0.0001
--parallel_envs 10 --node_dim 16 --ent_coef 0.002 --action_masking 0
```

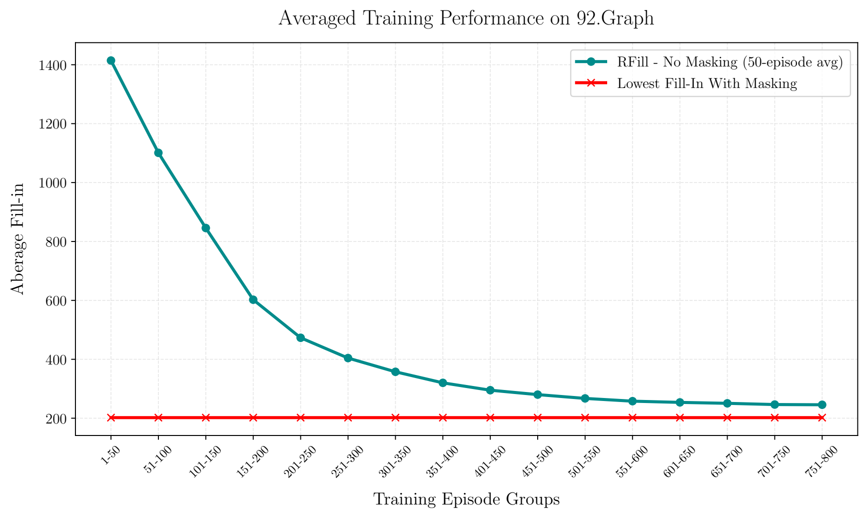

Figure 16: Average fill-in (lower is better) on 92.GRAPH, comparing masking vs. no masking during training.

```
$ python main.py datasets/99.graph --output_file non_masking_results/99.graph
--policy_sizes 64 32 --total_timesteps 1_000_000 --learning_rate 0.0001
--parallel_envs 10 --node_dim 16 --ent_coef 0.002 --action_masking 0
```

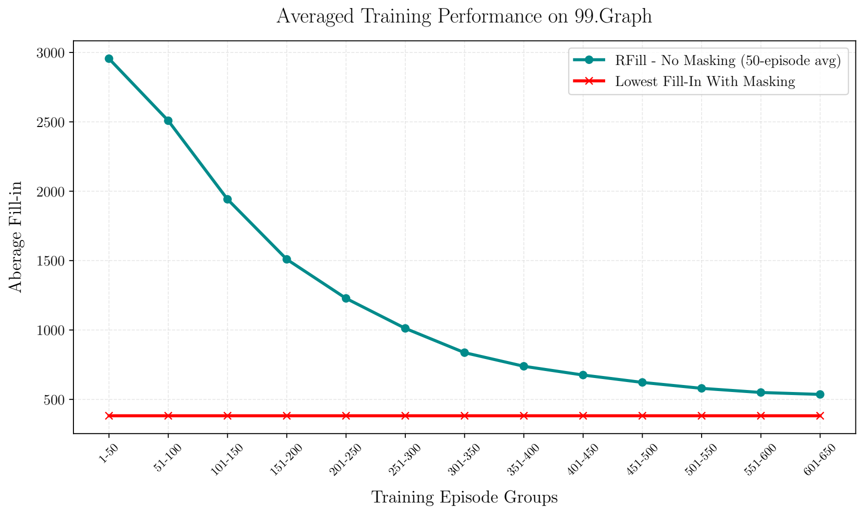

Figure 17: Average fill-in (lower is better) on 99.GRAPH, comparing masking vs. no masking during training.

```
$ python main.py datasets/100.graph --output_file non_masking_results/100.graph
--policy_sizes 64 32 --total_timesteps 1_000_000 --learning_rate 0.0001
--parallel_envs 10 --node_dim 16 --ent_coef 0.002 --action_masking 0
```

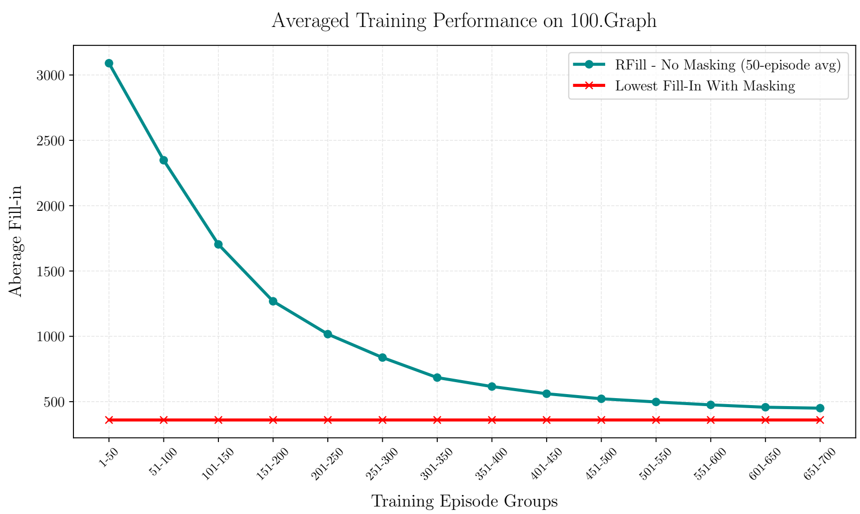

Figure 18: Average fill-in (lower is better) on 100.GRAPH, comparing masking vs. no masking during training.

D Generalization Experiment

For this experiment, we sample 35 graphs from $G(n,p), n = 50, p = 0.2$ using the following hyperparameters.

```
$ python gnp.py --output_file non_masking_results/gnp.graph
--policy_sizes --total_timesteps 500_000 --learning_rate 0.0001
--parallel_envs 35 --node_dim 16 --action_masking 1 --ent_coef 0.01
```

The model is trained on all 35 environments in parallel using action masking for $500,000$ timesteps. Next, we sample 200 new evaluation graph instances from $G(n,p), n = 50, p = 0.2$ and evaluate the fill-in from following the trained policy for each instance. To do this, we sample 25 elimination orders for each graph using the trained model, and report the minimum fill-in order found for each graph.

Figure 19 shows the improvements of the trained policy vs the MDH heuristic. Figure 20 shows the improvements of the trained policy vs the MFillH heuristic. Finally, Figure 21 shows the improvements of the trained policy vs both heuristics (i.e. compared to the minimum of both heuristics).

On average, the learned heuristic generalizes and gives an elimination order than is on average a 2.21% improvement over MDH, a 1.05% improvement over MFillH, and 0.63% over both heuristics. However, it does not always lead to a better fill-in for all graphs. Hence, the learned heuristic can be bootstrapped with the other heuristics to generate strictly better heuristics.

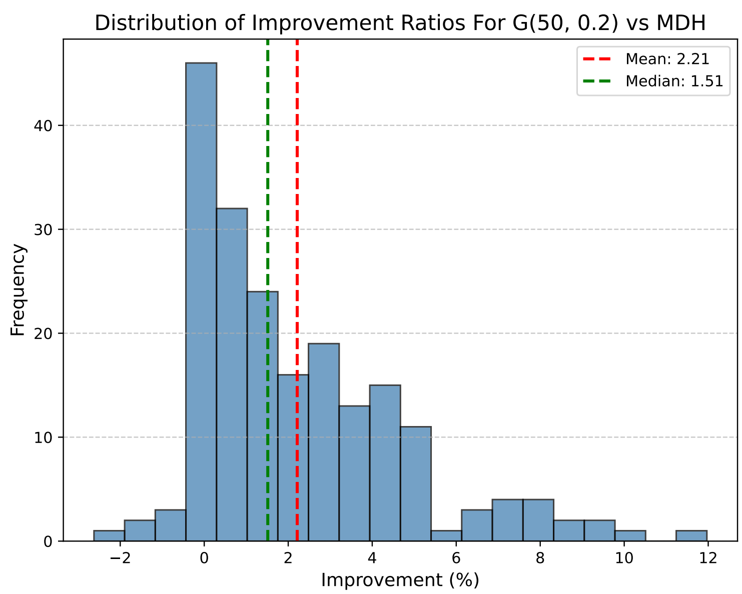

Figure 19: Improvement percentages over the 200 instances of $G(50, 0.2)$ evaluation graphs compared to MDH

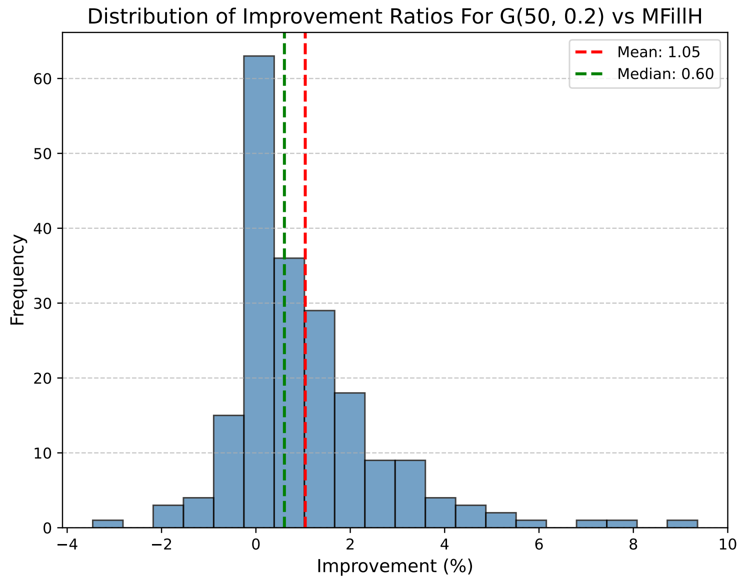

Figure 20: Improvement percentages over the 200 instances of $G(50, 0.2)$ evaluation graphs compared to MFillH

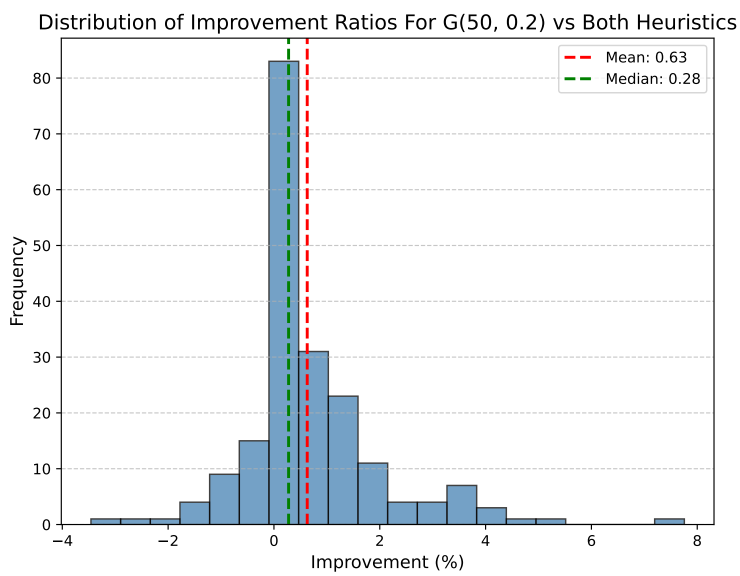

Figure 21: Improvement percentages over the 200 instances of $G(50, 0.2)$ evaluation graphs compared to both MDH and MFillH