Learning to Sample Hard Instances for Graph Algorithms

Ryoma Sato   R.SATO@ML.IST.I.KYOTO-U.AC.JP
Makoto Yamada   MYAMADA@I.KYOTO-U.AC.JP
Hisashi Kashima   KASHIMA@I.KYOTO-U.AC.JP
Kyoto University, Kyoto 606-8501, Japan.
RIKEN Center for Advanced Intelligence Project, Tokyo 103-0027, Japan.

Editors: Wee Sun Lee and Taiji Suzuki

Abstract

Hard instances, which require a long time for a specific algorithm to solve, help (1) analyze the algorithm for accelerating it and (2) build a good benchmark for evaluating the performance of algorithms. There exist several efforts for automatic generation of hard instances. For example, evolutionary algorithms have been utilized to generate hard instances. However, they generate only finite number of hard instances. The merit of such methods is limited because it is difficult to extract meaningful patterns from small number of instances. We seek for a probabilistic generator of hard instances. Once the generative distribution of hard instances is obtained, we can sample a variety of hard instances to build a benchmark, and we can extract meaningful patterns of hard instances from sampled instances. The existing methods for modeling the hard instance distribution rely on parameters or rules that are found by domain experts; however, they are specific to the problem. Hence, it is challenging to model the distribution for general cases. In this paper, we focus on graph problems. We propose HiSampler, the hard instance sampler, to model the hard instance distribution of graph algorithms. HiSampler makes it possible to obtain the distribution of hard instances without hand-engineered features. To the best of our knowledge, this is the first method to learn the distribution of hard instances using machine learning. Through experiments, we demonstrate that our proposed method can generate instances that are a few to several orders of magnitude harder than the random-based approach in many settings. In particular, our method outperforms rule-based algorithms in the 3-coloring problem.

Keywords: hard instances, graph problems, neural networks, reinforcement learning

1. Introduction

Given an algorithm for a combinatorial problem, how do we find instances that take a long time to be solved? We call such instances *hard instances* [1]. Finding hard instances is important in algorithm design for the following reasons:

- **Reason 1:** Hard instances help analyze and accelerate the algorithm.

- **Reason 2:** Hard instances help evaluate the performance of other algorithms.

---

1. Hard instances are also referred to as instances that require a long time for *any algorithm* to solve it. In this paper, they are only referred to as instances that require a long time for *a specific algorithm*.

The following illustrative example shows how hard instances help analyze an algorithm: Consider a sorting problem and quicksort, which uses the first element as a pivot. Suppose we do not know that the worst time complexity of quicksort is $\Theta(n^2)$. If we try running quicksort on some random sequences, quicksort seems to solve all instances in $O(n \log n)$ time. However, once a hard instance $[n, n-1, \ldots, 3, 2, 1]$ is found, we can see that the worst time complexity is $\Theta(n^2)$. Moreover, such an observation shows that choosing an elaborate pivot improves the algorithm (Reason 1). Besides, such an extreme case is useful for benchmarks because it reveals whether algorithms are robust to worst cases or are efficient only for average cases (Reason 2). Furthermore, in general, if we have hard instances for state-of-the-art algorithms in a benchmark problem instances, we can quickly check whether a new algorithm conquers weakness of the state-of-the-art algorithms (Reason 2).

Finding hard instances is helpful not only for academic subjects but also for practical and industrial subjects. For example, a task scheduler program solves the vertex coloring problem to optimize schedules. However, if a user inputs a malicious schedule (i.e., a hard instance), the scheduler takes a significant amount of time to solve the problem and may hang up. If the developers have such inputs beforehand, they can cope with the issue by setting the appropriate timeout period or maximum size of the input. Another example is the preparation of competitive programming contests. If we create hard instances for each problem, we can accurately check whether the submission is correct or not, which is useful for preparing competitions. Advantages of knowing hard instances are discussed in Cotta and Moscato (2003); van Hemert (2006); Smith-Miles et al. (2010) further.

There exist several efforts for automatic generation of hard instances. For example, Cotta and Moscato (2003); van Hemert (2006); Smith-Miles et al. (2010) generate hard instances using evolutionary algorithms. However, they generate only finite number of hard instances. The merit of such methods is limited because it is difficult to extract meaningful patterns from small number of instances. We seek for a probabilistic generator of hard instances. Once the generative distribution of hard instances is obtained, we can sample a variety of hard instances from the distribution to build a benchmark, and we can extract meaningful patterns of hard instances from sampled instances.

When we tackle this problem, we must specify the underlying set to model distribution. However, the form of instances depends on the problem. For example, instances are represented by an array in the sorting problem, and they are represented by a set of clauses in the SAT problem. In this paper, we focus on graph problems to fix the underlying set of distributions. Graph problems appear in many important problems. For example, the register allocation problem is formulated as the graph coloring problem (Chaitin et al., 1981), and the maximum clique problem can be utilized for community detection (Luce and Perry, 1949). It motivates us to focus on graph algorithms.

A straightforward method for modeling the hard graph distribution is to use a probabilistic graph model such as the Erdős-Rényi model (Erdős and Rényi, 1959). Though this is generic and simple, this is not efficient because worst time complexity is often far worse than average time complexity (Wilf, 1984). It indicates that, to model the hard graph distribution efficiently, we must develop a method that can capture the structure of the problem and generate *rare* instances.

Table 1: Qualitative comparison with other methods for generating hard instances.

	Random	Rule-based	Generic Algorithm	HISAMPLER
Effective		✓	✓	✓
Without hand-engineering	✓		✓	✓
Problem-agnostic	✓		✓	✓
Distribution	✓			✓
Sample-efficient		✓		✓

In this paper, we propose HISAMPLER, the <u>hard</u> <u>i</u>nstance sampler, to obtain a generative distribution of hard instances of graph algorithms. It models the hard graph distribution using a neural network and trains the model via reinforcement learning.

Through experiments on seven algorithms of four typical graph problems, we demonstrate that HISAMPLER can generate instances that are a few to several orders of magnitude harder than the random-based approach in many settings, and that our method outperforms rule-based algorithms in the 3-coloring problem. The implementation of HISAMPLER is publicly available in https://github.com/joisino/HiSampler as an open source project.

The major contributions of this paper are as follows:

- **Novel formulation:** We formulate the problem of modeling the hard instance distribution, which is practically important.

- **Novel method:** We propose HISAMPLER, an effective method to model the hard instance distribution for a given graph algorithm.

- **Experimental evidence:** We demonstrate the effectiveness of HISAMPLER through extensive experiments using seven algorithms of four problems.

Table 1 contrasts HISAMPLER against other methods for generating hard instances.

2. Proposed Method

We first describe the problem setting of this paper. Then, we propose HISAMPLER, an effective method to learn the distribution of hard instances for graph algorithms.

2.1. Problem Setting

We specify the task of learning the hard instance distribution of graph algorithms. In particular, we develop a method that models the hard instance distribution for algorithms for undirected, unweighted, and simple graphs. It is because they include many important problems. For example, the register allocation problem is formulated as the graph coloring problem (Chaitin et al., 1981) and the maximum clique problem can be utilized for community detection (Luce and Perry, 1949).

We aim to develop a method that relies on no problem specific properties; instead, we use only the hardness measures of the problem instances: $\text{hardness}(\boldsymbol{A}, L)$, where L is the given algorithm, $\boldsymbol{A} \in \{0, 1\}^{n(n-1)/2}$ is the adjacency matrix, and n is the number of vertices. The design of the hardness value is arbitrary if it can be obtained by actually running the

Table 2: Notations.

Notation	Description
G	A graph (i.e., an instance)
V	The whole set of nodes
n	The number of nodes (i.e., $n = \|V\|$)
$\boldsymbol{A} \in \{0,1\}^{n(n-1)/2}$	The upper triangular part of an adjacency matrix
$\boldsymbol{P} \in [0,1]^{n(n-1)/2}$	The upper triangular part of the a probabilistic adjacency matrix
L	A graph algorithm
$\mathrm{hardness}(\boldsymbol{A}, L) \in \mathbb{R}$	The hardness value of graph A for algorithm L
B	The maximum number of evaluations
l	The number of layers of the neural network
$d_i \ (i = 0, 1, \ldots, l)$	The dimensions of the hidden layers of the neural network

algorithm on the instance. For example, in our experiment, the hardness is measured by the number of recursive calls DSATUR (Brélaz, 1979) makes to solve the instance and by the real time Nauty (McKay and Piperno, 2014) spends to solve the instance.

Formally, given an algorithm L, we aim to develop a method that models a generative distribution $\mathcal{D}_L$ of graphs that maximizes $\mathbb{E}_{\boldsymbol{A} \sim \mathcal{D}_L}[\mathrm{hardness}(\boldsymbol{A}, L)]$. Besides, we develop a method that satisfies the following key assumptions.

Assumption1. Small instance: We fix the number of vertices n because we can generate arbitrarily hard instances just by increasing the number of vertices, which is not practical. Moreover, since small instances can be visualized and are easy to interpret and analyze, it is important to generate hard instances without increasing the size of the instance.

Assumption2. Sample efficiency: Evaluating the hardness value is time consuming, especially when the instance is hard. Besides, algorithms that require special devices such as GPU and multiple cores cost much even if they run in a short period of time. Therefore, we cannot evaluate too many instances, which motivates us to find hard instances more efficiently. To overcome this problem, we set the budget B of evaluation. In other words, we do not evaluate more than B instances during training. It should be noted that evolutionary algorithms are not sample-efficient because they evaluate the fitness functions of large population in each iteration.

Table 2 summarizes the notations we use throughout the paper.

2.2. Hard Instance Sampler

We propose HiSampler to model the distribution of hard instances of graph algorithms. Figure 1 illustrates the overview of HiSampler.

Probabilistic Model: We consider distributions on the binary vector $\{0,1\}^{n(n-1)/2}$ because a graph G is represented by an adjacency matrix $\boldsymbol{A} \in \{0,1\}^{n(n-1)/2}$. HiSampler models the distribution by a fully-connected neural network N with parameters $\boldsymbol{\theta}$. Let l denote the number of layers of N and let d_i be the dimensions of the hidden layers of N. d_0 is the input dimension and $d_l = n(n-1)/2$ is the output dimension. The neural network N takes a noise $\boldsymbol{z} \sim \mathcal{N}(0, I_{d_0})$ from the standard normal distribution as input and outputs a
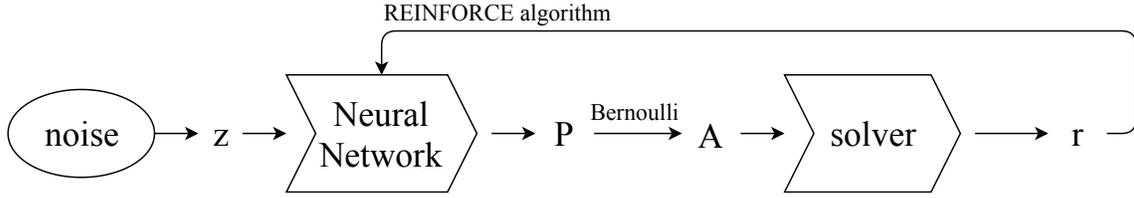
Figure 1: The overview of HiSampler

probabilistic adjacency matrix $\boldsymbol{P} \in [0,1]^{n(n-1)/2}$. Then, an adjacency matrix $\boldsymbol{A}$ is sampled from Bernoulli($\boldsymbol{P}$). Namely, $Pr[\boldsymbol{A}_i \mid \boldsymbol{P}] = \boldsymbol{P}_i$ $(i = 1, 2, \ldots n(n-1)/2)$ and each dimension of $\boldsymbol{A}$ is conditional independent given $\boldsymbol{P}$. It is worth noting that $\boldsymbol{A}_i$ and $\boldsymbol{A}_j$ $(i \neq j)$ are not independent without any condition because $\boldsymbol{P}_i$ and $\boldsymbol{P}_j$ are not independent. Therefore, HiSampler can model nonlinear relationships between edges.

Optimization: HiSampler optimizes the parameters of the neural network N using immediate-reinforcement learning. In this framework, the environment gives a noise $\boldsymbol{z}$ to the agent, the agent generates a graph $\boldsymbol{A}$ as an action, and the cost of solving the instance is fed back to the agent as a reward r (i.e., $r = \text{hardness}(\boldsymbol{A}, L)$). The policy of the agent is modeled by the neural network N. We optimize the parameters of the neural network N using REINFORCE algorithm (Williams, 1992):

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha r \frac{\partial}{\partial \boldsymbol{\theta}} \sum_{i=1}^{n(n-1)/2} (\log \boldsymbol{P}_i^{\boldsymbol{A}_i} + \log(1 - \boldsymbol{P}_i)^{(1-\boldsymbol{A}_i)}),$$

where α is the learning rate. The procedure used to train the neural network model is shown in Algorithm 1.

Prioritized Experience Replay: The main challenge of learning the distribution of hard instances is that hard instances are sparse in the instance space. This tendency is significant especially for sophisticated algorithms. To alleviate this issue, we use a variant of prioritized experience replay (Schaul et al., 2016). Namely, we maintain an experience pool that contains top-K hard instances. In each iteration, we sample an instance from the pool, and we train the model using the sampled experience. We refer to HiSampler with prioritized experience replay as HiSampler-PER to distinguish it from HiSampler-vanilla. The training procedure of HiSampler-PER is shown in Algorithm 2. We simplify the original prioritized experience replay by (1) using uniform distribution of top-K instances instead of weighted sampling and (2) using reward to prioritize experiences instead of TD-Error. Though this is simple, we found this works well in practice for HiSampler. We empirically demonstrate effectiveness of this method in Section 5.

2.3. Complexity Analysis

We analyze the time complexity of HiSampler. The bottleneck step of HiSampler is forward and backward calculation of the neural network. It takes $O(\sum_{i=0}^{l-1} d_i d_{i+1})$ time per iteration. Therefore, the time complexity of HiSampler is $O(n^2)$ with respect to the graph size because the dimension of the output layer is $d_l = n(n-1)/2$. The additional compu-

---

Algorithm 1 HiSampler-vanilla

Require: Algorithm L; Budget B.
Ensure: Hard graph distribution.
1: Initialize the parameters of the neural network N.
2: $r_{best} \leftarrow 0$
3: **for** B iterations **do**
4:     $\boldsymbol{z} \leftarrow$ a sample from $\mathcal{N}(0, I_d)$
5:     $\boldsymbol{P} \leftarrow N(\boldsymbol{z}; \boldsymbol{\theta})$
6:     $\boldsymbol{A} \sim \text{Bernoulli}(\boldsymbol{P})$
7:     $r \leftarrow \text{hardness}(\boldsymbol{A}, L)$
8:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha r \frac{\partial}{\partial \boldsymbol{\theta}} \sum_{i=1}^{n(n-1)/2}(\log \boldsymbol{P}_i^{\boldsymbol{A}_i} + \log(1 - \boldsymbol{P}_i)^{(1-\boldsymbol{A}_i)})$ $\triangleright$ REINFORCE algorithm
9: **end for**
10: **return** N

Algorithm 2 HiSampler-PER

Require: Algorithm L; Budget B; Pool size K.
Ensure: Hard graph distribution.
1: Initialize the parameters of the neural network N.
2: $r_{best} \leftarrow 0$
3: $Q \leftarrow$ an empty priority queue
4: **for** B iterations **do**
5:     $z \leftarrow$ a sample from $\mathcal{N}(0, I_d)$
6:     $\boldsymbol{P} \leftarrow N(\boldsymbol{z}; \boldsymbol{\theta})$
7:     $\boldsymbol{A} \sim \text{Bernoulli}(\boldsymbol{P})$
8:     $r \leftarrow \text{hardness}(\boldsymbol{A}, L)$
9:     $Q.\text{push}((r, \boldsymbol{z}, \boldsymbol{A}))$ $\triangleright$ Add a new sample to the queue
10:     **if** $\text{size}(Q) > K$ **then**
11:        $Q.\text{pop}()$ $\triangleright$ Remove the least significant sample
12:     **end if**
13:     $(r', \boldsymbol{z}', \boldsymbol{A}') \leftarrow$ a uniform random sample from Q.
14:     $\boldsymbol{P}' \leftarrow N(\boldsymbol{z}'; \boldsymbol{\theta})$
15:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha r \frac{\partial}{\partial \boldsymbol{\theta}} \sum_{i=1}^{n(n-1)/2}(\log \boldsymbol{P}_i'^{\boldsymbol{A}_i'} + \log(1 - \boldsymbol{P}_i')^{(1-\boldsymbol{A}_i')})$
16: **end for**
17: **return** N

---

tation needed for HiSampler-PER is maintaining the priority queue. It takes $O(\log K)$ time per iteration, which is negligibly small in practice.

3. Extensions

In our proposed method, the choice of the hardness function is arbitrary. Therefore, HiSampler can find not only instances that take a long time to be solved but also hard instances in terms of other criteria. We introduce two important examples.

3.1. Estimating Approximation Ratio

Let L be an approximation algorithm, let $\boldsymbol{A}$ be an instance of the problem, let $L(\boldsymbol{A})$ be the object value of the solution that L outputs for $\boldsymbol{A}$, and let $\mathrm{OPT}(\boldsymbol{A})$ be the optimal objective value of an optimal solution of $\boldsymbol{A}$. The approximation ratio of L is defined by $r(L) = \max_{\boldsymbol{A} \text{ is an instance}} \frac{L(\boldsymbol{A})}{\mathrm{OPT}(\boldsymbol{A})}$ for a minimizing problem and by $r(L) = \max_{\boldsymbol{A} \text{ is an instance}} \frac{\mathrm{OPT}(\boldsymbol{A})}{L(\boldsymbol{A})}$ for a maximizing problem. Estimating the approximation ratios is important for investigating the performance of the approximation algorithms. However, it is not trivial what instance maximizes the term. Here, we use $\frac{L(\boldsymbol{A})}{\mathrm{OPT}(\boldsymbol{A})}$ or $\frac{\mathrm{OPT}(\boldsymbol{A})}{L(\boldsymbol{A})}$ as the hardness value of the instance $\boldsymbol{A}$; then, we can search the maximizer by HISAMPLER. We will show an illustrative example with the well-known minimum vertex cover algorithm in Section 5.5.

3.2. Hard Instances for Enumerating Algorithms

Enumerating algorithms output all the elements that satisfy some property. The amortized time and maximum delay are sometimes investigated for evaluating the efficiency of enumerating algorithms. HISAMPLER can generate hard instances in terms of the amortized time and maximum delay by setting these measures as the hardness value.

4. Related Work

Constructing Hard Instances: There have been several researches on constructing hard instances of combinatorial problems. Hard instance generation was first studied in relation to the phase transition phenomena (Cheeseman et al., 1991; Hogg and Williams, 1994), which utilizes order parameters to generate hard instances. The three coloring instance generation by Mizuno and Nishihara (2008) and the graph isomorphism instance generation by Neuen et al. (2017) generate hard instances with rule-based algorithms. However, these works depend on problem specific knowledge, whereas our method is independent of the problem. Another approach is to generate instances by reducing other related problems. For example, the Latin square problem is found useful for constructing a benchmark for the graph-coloring problem (Gomes and Shmoys, 2002) and SAT (Achlioptas et al., 2000). However, the conversion of an instance of the Latin square problem into those of other problems also requires problem-specific knowledge. The methods that are most related to this paper are evolutionary algorithms (Cotta and Moscato, 2003; van Hemert, 2006; Smith-Miles et al., 2010). They optimize the hardness of instances using the evolutionary algorithm. However, they require designing gene representation for each task and cannot find a distribution but find a finite set of hard instances.

Deep Generative Graph Models: Recently, several generative graph models utilizing deep learning techniques have been proposed. The variational graph auto-encoder (Kipf and Welling, 2016) is one of the first models of this kind. It is a variant of the Variational Auto Encoder (VAE), which outputs a probabilistic adjacency matrix. This model was used for the link prediction of citation networks. Then, VAE (Simonovsky and Komodakis, 2018; Grover et al., 2019; Ma et al., 2018), Generative Adversarial Networks (GAN) (Wang et al., 2018; Bojchevski et al., 2018), and sequential generation (You et al., 2018b; Liu et al., 2018; You et al., 2018a) based generating models were proposed. In particular, they succeeded in

generating various de-novo chemical materials and modeling real-world networks. ORGAN (Guimaraes et al., 2017) utilizes SeqGAN (Yu et al., 2017) and reinforcement learning to generate molecular graphs with the desired properties. It uses SMILES (Weininger, 1988) to represent a molecular graph because SeqGAN generates a sequence of symbols rather than a graph itself. MolGAN (Cao and Kipf, 2018) is another graph generative model utilizing GAN and reinforcement learning. It models the probabilistic adjacency matrix and attributes of graphs directly instead of using SMILES. The differences between HiSampler and deep generative graph models are (1) these models use training data that contain graphs with high objective values whereas HiSampler uses no training data and (2) many of these models are designed for generating molecular graphs, where the size of graphs is typically at most dozens, whereas HiSampler can generate graphs with more than a hundred nodes.

5. Experiments

We will answer the following questions through experiments:

Q1. Scalability: How fast is HiSampler?

Q2. Effectiveness: Does HiSampler generate harder instances than existing methods?

Q3. Knowledge Extraction: Can HiSampler provide insights for algorithm design?

Q4. Diversity: Is the distribution of HiSampler rich in diversity?

Q5. Extensions: Can HiSampler estimate an approximation ratio?

Q6. Effective Patterns: How can we extract effective patterns from the obtained hard graph distribution?

Common Experimental Setup: We set the number of layers of HiSampler to three and the dimensions of the hidden layers to $d_0 = 10$, $d_1 = 100$, $d_2 = 500$, and $d_3 = n(n-1)/2$ throughout experiments. The activation functions in the hidden layers are ReLU, and the final output is processed by sigmoid activation. We use Adam (Kingma and Ba, 2015) with learning rate 0.0001 to train the model. We set the pool size of HiSampler-PER to $K = 10$ throughout experiments. We conduct experiments with Intel Xeon E5-2690 CPU. It should be noted that we can speed up the computation of HiSampler by GPUs, but we do not use GPUs for fair comparison.

5.1. Scalability

As we mentioned in Section 2.3, the complexity of HiSampler is $O(n^2)$. We investigate time consumption of training and sampling of HiSampler through experiments. We sample 100 instances from each of 10 HiSamplers, and we execute one step of training for each sample. We omit the time of evaluating the hardness value during training because the overhead of evaluation is common with other methods. Furthermore, we consider that training is already done when the evaluation time overwhelms model computation. If the evaluation takes much time in the initial evaluation, we should make the graph size smaller because generating small instances is important (Assumption 1 in Section 2.1).

Figure 2 reports the mean time of a single iteration of sampling and training. This shows that the computation does not grow much even if the number of nodes increases. In
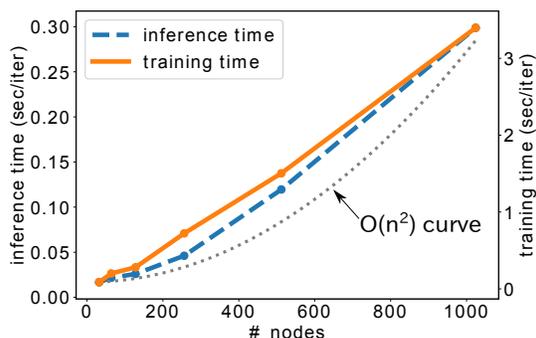
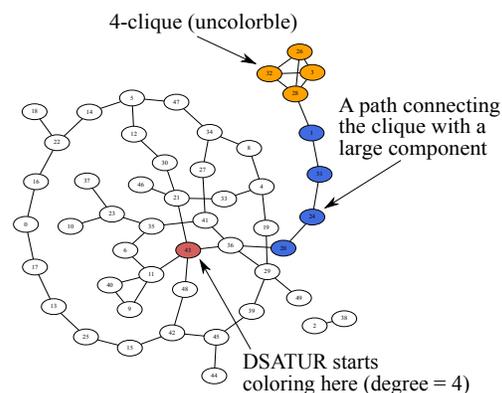Figure 2: The time consumption of the inference and training processes of HiSampler.

Figure 3: An example of 3-coloring instances that HiSampler generates. DSATUR takes more than one billion recursive calls to solve this instance.

particular, one iteration of the training takes only four seconds even with 1024 nodes. It indicates that HiSampler is highly efficient.

5.2. Effectiveness

We demonstrate how hard instances HiSampler can generate compared to existing methods. We use the 3-coloring problem, the minimum vertex cover problem, the maximum clique problem, and the graph isomorphism problem, and the following seven algorithms for these problems

DSATUR (3-coloring): This is a backtracking search method based on DSATUR (Brélaz, 1979). It assigns colors to the vertices one by one. At each step, it chooses one of the uncolored vertices that have the least number of candidate colors. If there are many such vertices, it chooses the vertex with the maximum degree. If the color assignment becomes inconsistent, it backtracks until it finds a consistent assignment. This method always outputs exact solution whereas the original DSATUR is not. We use the number of recursive calls as the hardness value.

MiniSat (3-coloring): This reduces the 3-coloring problem to the SAT problem, and this solves the reduced instance using MiniSat (Eén and Sörensson, 2003). We use the number of decisions MiniSat reports as the hardness value.

B&B (minimum vertex cover): This is a branch and bound algorithm that uses a greedy maximal matching as an upper bound. We use the number of recursive calls as the hardness value.

BK (maximum clique): This is a branch and bound algorithm based on the Bron-Kerbosch method (Bron and Kerbosch, 1973). This prunes the state when the union of the selected nodes and candidate nodes is smaller than the maximum clique found so far. Note that the original Bron-Kerbosch method enumerates all the maximal cliques whereas this algorithm only outputs a maximum clique. We use the number of recursive calls as the hardness value.

MCS (maximum clique): This is MCS (Tomita et al., 2010), a branch and bound algorithm. We use the time consumption as the hardness value (10^{-2} sec).

FMC (maximum clique): This is Fast Max-Clique Finder (Pattabiraman et al., 2013), a hierarchical-pruning based algorithm. We use the time consumption as the hardness value (10^{-6} sec)

Nauty (graph isomorphism): This is Nauty (McKay and Piperno, 2014), one of the state-of-the-art graph isomorphism solvers. We use the time consumption as the hardness value (10^{-6} sec)

To compare the effectiveness of HiSampler, we use the following baseline methods.

Generic algorithm: This searches hard instances using an evolutionary algorithm. We use the adjacency matrix A as gene representation. We use the same hyperparameters as van Hemert (2006). Namely, the population size is 30, crossover is performed uniformly, mutation occurs with uniform probability with adapting mutation rate, and the fitness is the hardness value.

Random graphs: This samples B graphs from the Erdős-Rényi model (Erdős and Rényi, 1959) and reports the hardest one.

Rule-based: We use several rule-based methods in the 3-coloring problem and the graph isomorphism problem. Cheeseman et al. (1991) and Hogg and Williams (1994) used the Erdős-Rényi model with carefully tuned parameters for the 3-coloring problem (i.e., $p = \frac{4.6}{n-1}$ and $p = \frac{3.4}{n-1}$, respectively). Vlasie (1995) found that a regular structure plays a key role for hard instances and generated graphs with less 3-paths for the 3-coloring problem. Mizuno and Nishihara (2008) and Neuen and Schweitzer (2017) used characteristic gadgets to construct hard instances. We generate B graphs using these methods and report the hardest one.

The most important step for HiSampler and the generic algorithm is initialization. It is known that the algorithm takes long time for graphs with certain range of edge density and that it takes short time for graphs with other density (Cheeseman et al., 1991). For example, 3-coloring algorithms can easily assert there is no solutions for dense graphs. If the initial distribution of HiSampler or initial population of the generic algorithm is far from the hard region, it takes much time for them to generate hard instances. To alleviate this problem, we determine the edge probability p^* beforehand where each algorithm takes long time to process graphs with this density. We can use prior knowledge about the algorithm to determine p^*. Alternatively, if we do not have such knowledge, we evaluate random instances of different edge probabilities (e.g., $p = 0.01, 0.02, \ldots, 0.99$), and we can use the hardest one as p^*. It consumes negligibly small budget. We initialize the population of the generic algorithm by Erdős-Rényi model with $p = p^*$, and we initialize the bias of the last layer of HiSampler as $b = -\log(\frac{1}{p^*} - 1)$ so that $\sigma(b) = p^*$, where σ is the sigmoid function. We initialize the weight matrices of HiSampler with Xavier initializer (Glorot and Bengio, 2010) and biases of the lower layers with zeros as the default setting of the library. It should be noted that the other hyperparameters than the graph size n and the initial edge probability p^* are fixed throughout experiments.

We set the budget size as $B = 10000$, and we stop a method when it takes more than a day. We measure the hardness value of the hardest instance each method finds. We run 5 experiments for each method with different seeds and we report the mean of 5 runs. Table 3 summarizes the result of the experiments. We can see the following observations.

Observation 1. Prioritized experience replay is effective: HISAMPLER-PER consistently outperforms HISAMPLER-VANILLA except for FMC, where HISAMPLER-VANILLA slightly outperforms HISAMPLER-PER. It indicates that prioritized experience replay works well for HISAMPLER.

Observation 2. HISAMPLER-PER **outperforms the generic algorithm:** HISAMPLER-PER consistently outperforms the generic algorithm especially in DSATUR algorithm. It shows that HISAMPLER learns the hard distribution effectively.

Observation 3. HISAMPLER **outperforms random-based methods:** HISAMPLER consistently outperforms the Erdős-Rényi model with $p = p^*$. It demonstrates that the distribution of HISAMPLER is not random. HISAMPLER learns effective structure of the hard graph distribution.

Observation 4. HISAMPLER **outperforms rule-based methods:** HISAMPLER consistently outperforms rule-based methods. It indicates that HISAMPLER can find highly effective structure for hard instances that could not be found manually.

It should be noted that we measured the hardest instance that each method found because the generic algorithm aims at searching a hard instance instead of modeling the hard instance distribution. We will investigate the properties of the distribution (e.g., diversity, mining patterns) in the later experiments.

5.3. Knowledge Extraction

We demonstrate how a hard instance provides helpful insight into making a better search algorithm using a concrete example. Figure 3 shows an example of the instances HISAMPLER generates. There are no solutions for this instance because it has a 4-clique C (highlighted in orange). However, DSATUR cannot explicitly detect 4-cliques. It first assigns colors to $V \backslash C$. Every time it finds a solution for $V \backslash C$, the partial solution is immediately rejected when the algorithm starts to color the 4-clique C. Then, the search is back-tracked and the algorithm starts to find other assignments of $V \backslash C$. However, it does not obtain any result because any assignment will be rejected by the 4-clique C. Finally, the algorithm finds all the valid assignments of $V \backslash C$, and reports that there are no solutions for this instance. The key point is that the 4-clique C is connected to $V \backslash C$ by a path P (highlighted in blue). P plays a role of a "bottleneck". When the algorithm is coloring $V \backslash (C \cup P)$, the number of color candidates of vertices of P are at most two, and the degrees of them are only two. Therefore, DSATUR is reluctant to color these vertices. From this analysis, we can improve the backtracking search by preprocessing: deleting vertices whose degree is not more than two. Deleting such vertices does not change the answer because we can color the vertices whose degree is not more than two whatever the coloring assignment of the other vertices is: just color the vertex with the color that is not the same as the colors of adjacent nodes. This improvement helps avoid the problem described above. This discussion is a good example

Table 3: The experimental results

Problem	3-coloring		Vertex Cover
Algorithm	DSATUR	MiniSat	B&B
n	50	200	50
p^*	0.1	0.025	0.1
HiSampler-vanilla	261331027.8	1120.4	8145.2
HiSampler-PER	**610024238.8**	**2674.2**	**21376.4**
Generic Algorithm	2464.8	660.8	8127.6
Erdős-Rényi $p = p^*$	407.0	693.6	3259.2
Erdős-Rényi $p = 0.1$	407.0	351.8	3259.2
Erdős-Rényi $p = 0.5$	2.0	282.4	2227.8
Erdős-Rényi $p = 0.9$	2.0	276.0	1160.8
Cheeseman et al. (1991)	597.8	810.2	N/A
Hogg and Williams (1994)	3883.8	815.8	N/A
Vlasie (1995)	240867.8	708.2	N/A
Mizuno and Nishihara (2008)	166294.6	875.4	N/A

Problem	Clique			Isomorphism
Algorithm	BK	MCS	FMC	Nauty
n	32	150	32	50
p^*	0.9	0.9	0.9	0.9
HiSampler-vanilla	65460.4	573.4	**6119052.0**	604.0
HiSampler-PER	**110591.0**	**1877.0**	5682186.0	**786.0**
Generic Algorithm	25019.6	1117.4	2588840.0	57.6
Erdős-Rényi $p = p^*$	6380.8	276.2	58655.6	10.2
Erdős-Rényi $p = 0.1$	82.8	0.0	3259.2	10.5
Erdős-Rényi $p = 0.5$	502.8	1.0	5284.2	2.7
Erdős-Rényi $p = 0.9$	6380.8	276.2	58655.6	10.2
$R(B(G_n, \sigma))$ (2017)	N/A	N/A	N/A	230.0
shrunken multipedes (2017)	N/A	N/A	N/A	102.6

to show that analyzing a hard instance helps design an algorithm robust to hard instances (corresponds to Reason 1 in Section 1).

5.4. Diversity

We show that a variety of hard instances are sampled from the distribution HiSampler learns. Diversity of hard instances helps build a benchmark and extract meaningful pattern. We train HiSampler-vanilla for DSATUR algorithm. The hardness value of the hardest instance x^* that HiSampler finds is 1637666819. We sample 1000 instances from the distribution for which the Jaccard indices of edges between x^* are less than 0.7. The mean of the hardness values of these instances is 3943028.974, which is still harder than the random models and the rule-based methods, and the mean of the Jaccard indices is 0.646. Moreover, the hardness value of the hardest instance among them is 819309215, keeping

(a) Support: 999   (b) Support: 999   (c) Support: 962   (d) Support: 959   (e) Support: 959
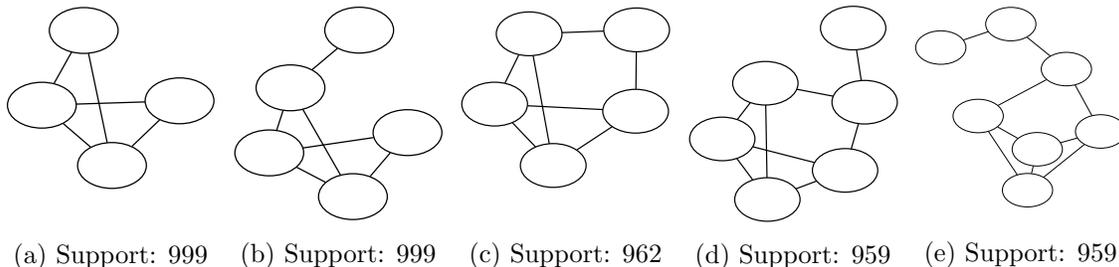
Figure 4: Frequent subgraphs of the hard distribution for DSATUR.

the Jaccard index 0.694. It shows that HiSampler retains diversity, whereas the genetic algorithm only generates a fixed number of instances.

5.5. Extensions

We show an illustrative example to estimate the approximation ratio using the greedy algorithm for the minimum vertex cover problem. It is known that the approximation ratio of the greedy algorithm is 2. We use the Erdős-Rényi model with $p = 0.1$ and HiSampler-vanilla with $p^* = 0.1$. We set the number of vertices as $n = 50$. The other settings are common with previous experiments. We use $r(\boldsymbol{A}) = \exp(10 \cdot L(\boldsymbol{A})/\mathrm{OPT}(\boldsymbol{A}))$ as the hardness value, which is monotonically increasing for $L(\boldsymbol{A})/\mathrm{OPT}(\boldsymbol{A})$. We found that the slope of $L(\boldsymbol{A})/\mathrm{OPT}(\boldsymbol{A})$ is too gentle to train the model, and used the objective function instead. We ran 5 experiments with difference seeds. None of the uniformly random graph models found an instance $\boldsymbol{A}$ that satisfies $L(\boldsymbol{A})/\mathrm{OPT}(\boldsymbol{A}) = 2$. However, all five HiSampler models succeeded in finding an instance $\boldsymbol{A}$ that satisfies $L(\boldsymbol{A})/\mathrm{OPT}(\boldsymbol{A}) = 2$. It shows that HiSampler is useful for estimating the approximation ratios of approximation algorithms.

5.6. Effective Patterns

We demonstrate how to extract meaningful patterns from the hard graph distribution. Toward this end, we use frequent subgraph mining (Inokuchi et al., 2000; Kuramochi and Karypis, 2001). This discovers frequent patterns appeared in a database of graphs. We sampled 1000 graph from the distribution that HiSampler learns for DSATUR. We utilize gSpan (Yan and Han, 2002) to extract frequent subgraphs of them. Figure 4 lists the frequent subgraphs that has at least 5 edges and appears in more than 950 sampled graphs. Figure 4a corresponds to the 4-clique highlighted in orange in Figure 3, and Figure 4e is a subgraph where a small unsolvable graph is connected to a path, which is effective structure as we analyzed in Section 5.3. It indicates that frequent subgraph mining tools are useful for extracting meaningful structure from the hard graph distribution.

6. Conclusion

This work tackled the problem of learning the distribution of hard instances using machine learning for the first time. We proposed HiSampler to model the hard instance distribution of graph algorithms. HiSampler is applicable to any graph algorithm without any

prior knowledge. We demonstrated the effectiveness of HiSampler using seven algorithms for four graph problems. Furthermore, we showed that hard instances provided insight to analyze and accelerate the algorithm. We also showed that frequent subgraph mining extracts meaningful patterns from the hard graph distribution.

We discuss some future work of this work. Many existing works have tackled molecular generation using deep learning models. Comparing HiSampler with these methods and utilizing them for modeling the hard graph distribution is important future work. Besides, we model the distribution of graphs using adjacency matrices. We do not take isomorphism into account because some algorithms utilize node indices for tie-breaking. However, this limits the effectiveness of HiSampler for algorithms that utilize isomorphism by, for example, restarting with randomization. Modeling symmetry of graphs for such algorithms is promising future work.

7. Acknowledgments

References

Dimitris Achlioptas, Carla P. Gomes, Henry A. Kautz, and Bart Selman. Generating satisfiable problem instances. In *AAAI*, pages 256–261, 2000.

Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. Net-GAN: Generating graphs via random walks. In *ICML*, pages 609–618, 2018.

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

Coenraad Bron and Joep Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

Nicola De Cao and Thomas N. Kipf. MolGAN: An implicit generative model for small molecular graphs. *CoRR*, abs/1805.11973, 2018. URL http://arxiv.org/abs/1805.11973.

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

Peter C. Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *IJCAI*, pages 331–340, 1991.

Carlos Cotta and Pablo Moscato. A mixed evolutionary-statistical analysis of an algorithm's complexity. *Appl. Math. Lett.*, 16(1):41–47, 2003.

Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.

Paul Erdős and Alfréd Rényi. On random graphs I. *Publicationes Mathematicae*, 6:290–297, 1959.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, pages 249–256, 2010.

Carla P. Gomes and David Shmoys. Completing quasigroups or latin squares: A structured graph coloring problem. In *Computational Symposium on Graph Coloring and Generalizations*, 2002.

Aditya Grover, Aaron Zweig, and Stefano Ermon. Graphite: Iterative generative modeling of graphs. In *ICML*, pages 2434–2444, 2019.

Gabriel Lima Guimaraes, Benjamin Sanchez-Lengeling, Pedro Luis Cunha Farias, and Alán Aspuru-Guzik. Objective-reinforced generative adversarial networks (ORGAN) for sequence generation models. *CoRR*, abs/1705.10843, 2017.

Tad Hogg and Colin P. Williams. The hardest constraint problems: A double phase transition. *Artif. Intell.*, 69(1-2):359–377, 1994.

Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, pages 13–23, 2000.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Thomas N. Kipf and Max Welling. Variational graph auto-encoders. *CoRR*, abs/1611.07308, 2016. URL http://arxiv.org/abs/1611.07308.

Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *ICDM*, pages 313–320, 2001.

Qi Liu, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. Constrained graph variational autoencoders for molecule design. In *NeurIPS*, pages 7806–7815, 2018.

R. Duncan Luce and Albert D. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, Jun 1949.

Tengfei Ma, Jie Chen, and Cao Xiao. Constrained generation of semantically valid graphs via regularizing variational autoencoders. In *NeurIPS*, pages 7113–7124, 2018.

Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60(0):94 – 112, 2014.

Kazunori Mizuno and Seiichi Nishihara. Constructive generation of very hard 3-colorability instances. *Discrete Applied Mathematics*, 156(2):218–229, 2008.

Daniel Neuen and Pascal Schweitzer. Benchmark graphs for practical graph isomorphism. In *ESA*, pages 60:1–60:14, 2017.

Bharath Pattabiraman, Md. Mostofa Ali Patwary, Assefaw Hadish Gebremedhin, Wei-keng Liao, and Alok N. Choudhary. Fast algorithms for the maximum clique problem on massive sparse graphs. In *Algorithms and Models for the Web Graph, WAW*, pages 156–169, 2013.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *ICLR*, 2016.

Martin Simonovsky and Nikos Komodakis. Graphvae: Towards generation of small graphs using variational autoencoders. In *ICANN*, pages 412–422, 2018.

Kate Smith-Miles, Jano I. van Hemert, and Xin Yu Lim. Understanding TSP difficulty by learning from evolved instances. In *Learning and Intelligent Optimization, 4th International Conference, LION, 2010.*, pages 266–280, 2010.

Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *WAL-COM*, pages 191–203, 2010.

Jano I. van Hemert. Evolving combinatorial problem instances that are difficult to solve. *Evolutionary Computation*, 14(4):433–462, 2006.

Romulus Dan Vlasie. Systematic generation of very hard cases for graph 3-colorability. In *ICTAI*, pages 114–119, 1995.

Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. GraphGAN: Graph representation learning with generative adversarial nets. In *AAAI*, pages 2508–2515, 2018.

David Weininger. SMILES, a chemical language and information system. 1. introduction to methodology and encoding rules. *J. Chem. Inf. Comput. Sci.*, 28(1):31–36, 1988.

Herbert S. Wilf. Backtrack: An O(1) expected time algorithm for the graph coloring problem. *Inf. Process. Lett.*, 18(3):119–121, 1984.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, 1992.

Xifeng Yan and Jiawei Han. gSpan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.

Jiaxuan You, Bowen Liu, Zhitao Ying, Vijay S. Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. In *NeurIPS*, pages 6412–6422, 2018a.

Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and Jure Leskovec. GraphRNN: Generating realistic graphs with deep auto-regressive models. In *ICML*, pages 5694–5703, 2018b.

Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. SeqGAN: Sequence generative adversarial nets with policy gradient. In *AAAI*, pages 2852–2858, 2017.