Greedy Graph Coloring with Reinforcement Learning

Sadhika Malladi

Abstract

In this paper, we seek to return to the classical problem of graph coloring with a machine learning augmentation. Adapting the techniques developed by (Khalil et al., 2017), we demonstrate that colorings on large graphs can be improved significantly beyond the solution provided by the classical greedy approach. We furthermore test how well these algorithms learn the underlying problem by testing trained models on different distributions of graphs, including real-world ones known to be difficult for heuristics.

1. Introduction

The graph coloring problem is a famous problem that has applications in many subfields such as scheduling and assignment problems. The problem is as follows. Given a graph $G = (V, E)$, find a coloring function $c : V \to \mathbb{N}$ such that for all $(u, v) \in E$, $c(u) \neq c(v)$. The numbers can be interpreted as colors assigned to the vertices, so the problem essentially requires that any two vertices joined by an edge do not have the same color. In most practical applications of the problem, it is desirable to minimize the number of colors required to color every vertex in the graph, as this minimizes the overall runtime of the algorithm and maximizes how parallelized solution computation can be. The minimum number of colors needed, a property of G, is the *chromatic number*, which is denoted $\chi(G)$. Finding $\chi(G)$ is an NP-hard problem, and correspondingly, it is NP-complete to find the coloring that minimizes the number of colors in the graph.

A number of heuristic and approximation algorithms have been developed to try to solve this problem (Husfeldt, 2015). In this paper, we consider the naive greedy algorithm, outlined in Algorithm 1, and seek to improve its ordering via reinforcement learning.

2. Theory

We will give a brief overview of theoretical results that demonstrate the importance of the ordering of the vertices in Algorithm 1 as well as bounds on how well the algorithm performs.

Algorithm 1 Naive Greedy Graph Coloring

Input: $G = (V, E)$
Randomly order the vertices in V: $v_1, v_2, ..., v_n$
for $i = 1$ **to** n **do**
  Compute C, the set of colors already assigned to neighbors of v_i.
  Color v_i with smallest $c \notin C$.
end for

In choosing the color c for a vertex, it is clear that $c \leq |C|$. $|C|$ is moreover upper-bounded by $\Delta(G) + 1$, where $\Delta(G)$ is the largest vertex degree in the graph. So, we have that the maximum color we assign is thusly upper-bounded, telling us that this algorithm yields a coloring using at most $\Delta(G) + 1$ colors.

For every graph G, there exists an optimal ordering such that Algorithm 1 yields the optimal coloring. This ordering can be easily constructed if we already know the optimal coloring c^* by ordering the vertices such that $c^*(v_i) \leq c^*(v_{i+1})$. However, without access to the optimal coloring ahead of time, we see there is little chance of a random ordering achieving this optimality.

It has been shown that Algorithm 1 performs decently well on random graphs, where it uses $n/(\log n - 3 \log \log n)$ colors on an n-vertex graph, roughly $2\chi(G)$ (Grimmett & McDiarmid, 1975). However, for every $\epsilon > 0$, there exists a graph G with $\chi(G) = n^\epsilon$ where Algorithm 1 uses $\Omega(n/\log n)$ colors with high probability (Kučera, 1991). As such, the naive greedy algorithm is not a good probabilistic algorithm.

There have been a number of algorithms that seek to improve the ordering of the vertices, but we will not be considering those in this paper. We consider one additional greedy algorithm known as DSatur, which uses *degree of saturation*, a measure of how much of a neighborhood of an uncolored vertex has been colored, as a heuristic to order the vertices dynamically (Brélaz, 1979). The algorithm is described procedurally in Algorithm 2. DSatur has been shown to be exact for bipartite, cycle, and wheel graphs (Brélaz, 1979). Moreover, DSatur has been shown to be empirically better than the greedy algorithm on random graphs (Lewis, 2015).

Algorithm 2 DSatur Greedy Coloring (Brélaz, 1979)

Input: $G = (V, E)$
Order the vertices of V decreasing in their degree.
Order the first vertex with the first color.
while G is not fully colored **do**
  Find the vertex v with highest degree of saturation. Break ties with degree and further with randomness.
  Compute C, the set of colors already assigned to neighbors of v.
  Color v with the smallest $c \notin C$.
end while

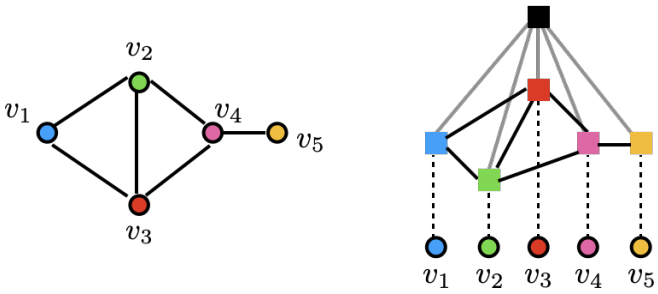

Figure 1. Example of embedding of graph (left) into message-passing network (right). The squares are nonlinear embeddings of the states (i.e., coloring status) of the vertices. The black box represents a chosen vertex to color next, taken as the vertex with maximum probability when the states are viewed as a probability distribution. Messages are passed 5 times in this structure before the output is computed.

3. Methods

In this section, we outline the two reinforcement learning algorithms used to solve the graph coloring problem.

3.1. Pointer Network Actor-Critic

We briefly discuss the Pointer Network Actor-Critic (PN-AC) architecture proposed in (Bello et al., 2016), which serves as a baseline reinforcement learning method in this paper. This work was one of the first papers to investigate the use of reinforcement learning for combinatorial optimization problems, but it focused on the traveling salesman problem.

A pointer network is a recurrent framework that allows flexibility in the input (i.e., graph) size. It was first introduced in (Vinyals et al., 2015) and refined for end-to-end training by (Bello et al., 2016). The pointer network uses the attention mechanism to choose an output from among the inputs instead of from a fixed-size vocabulary. In the PN-AC framework, a pointer network is used to represent the policy of the agent. To adapt the problem for graph coloring, we encode the vertices using a state vector that contains a binary adjacency vector.

The actor-critic framework is used as follows. The critic generates a baseline reward, which is often the exponential moving average of the rewards obtained by the actor over time. In the original PN-AC paper, the critic is parameterized by a network, but in this work, we use the exponential moving average as it significantly improved performance on the graph coloring problem. The gradient of the policy is computed with respect to the generated baseline guess using the REINFORCE algorithm from (Williams, 1992), and standard stochastic gradient descent is used to improve the policy.

3.2. Structure2Vec

The PN-AC struggles to manage larger graphs because it must process the sometimes sparse adjacency vector (i.e., state) of each vertex in one pass. Alternatively, (Khalil et al., 2017) proposes using structure2vec, a message-passing algorithm that can more comprehensively assess the topology of the graph. As such, the authors use structure2vec to parameterize the policy instead of a pointer network.

Structure2Vec, illustrated in Figure 1, constructs a nonlinear embedding of the states of the vertices and connects them per the graph structure. In the graph coloring problem, the states of the vertices were simply binary indicating if a vertex was colored or not already. Then, for 5 rounds, the embeddings are updated in terms of the means of the adjacent states. For details on the message-passing computation, we refer readers to Algorithm 1 in (Dai et al., 2016).

As a result of this structure, the embedded state of a vertex is a function also of the states adjacent to it. In the coloring problem, this resembles the saturation heuristic of DSatur, as we have some measure of how many vertices in a local neighborhood have been colored. It is worth noting that the radius of these neighborhoods depends on the density of the graphs. In a very dense graph, 5 rounds of message passing could conceivably update each vertex with information from all the vertices.

3.3. Reinforcement Learning Framework

We will now explain how the graph coloring problem translates to a reinforcement learning problem.

1. States: a state S on the graph is the subset of nodes that have already been colored. At the start of training, $S = \emptyset$, and at the end, $S = V$. S is represented as a binary vector of length $|V|$.

2. Action: an action is the selection of a node to color (i.e., update the state with).

3. Transition: once an action is chosen, the vertex is colored per the greedy algorithm described previously. That is, the smallest allowable color is assigned to the given vertex, and the state is updated.

4. Reward: we give a reward of +10 if the transition did not need to assign a new color to the chosen vertex and -1 if it did. We chose an asymmetric reward function because there are many situations in which the algorithm *must* use a new color, so penalizing it heavily for that would be unreasonable.

5. Policy: the policy is parameterized by a structure2vec framework, which outputs an action (i.e., node to color).

With this framework in mind, we can describe the learning algorithm. Although structure2vec was initially proposed for a supervised learning setting, (Khalil et al., 2017) trains the parameters end-to-end as one does with a policy. The learning algorithm is unchanged from (Khalil et al., 2017), so we discuss it at a high level and refer readers to Algorithm 1 in the paper to see details.

The learning algorithm is a combination of n-step Q-learning and fitted Q-iteration. We define an *episode* as a full coloring of a graph and a *step* as a single action taken during an episode. n-step Q-learning updates the policy parameters based on the reward earned after n steps, which helps the policy compensate for delayed rewards (i.e., moves that are suboptimal in the moment but lead to a bigger payoff later).

Fitted Q-iteration is a methodology applied when training a function approximator. Our structure2vec mechanism can be seen as such, so the fitted Q-iteration is appropriate here. The method uses experience replay, taking a batch of episodes from throughout training as the input to compute the gradient descent.

3.4. Computing Environment

Each network was trained on an Nvidia GeForce GTX 1080Ti. Runtime for training took between 12 and 20 hours, depending on the edge density of the graphs and the number of vertices in them. Each model used between 5% and 15% of the volatile GPU memory, allowing multiple models to be trained simultaneously on the same GPU. The PN-AC networks took less time to train, roughly 8 to 12 hours.

The code used for PN-AC and the structure2vec framework is written in PyTorch, which led to some speed decrease for the structure2vec algorithm in comparison to the original paper's code, which was written in C++. We could not extend the original repository because we do not know C++ well enough. Doing so would have likely sped up training greatly though.

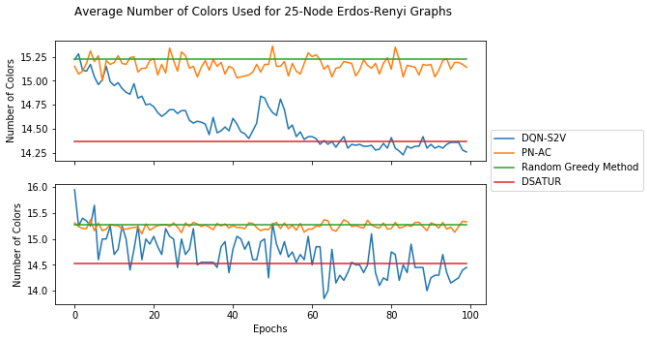

Figure 2. Training (top) and validation (bottom) curves for the four algorithms: DQN-S2V, PN-AC, the random greedy coloring in Algorithm 1, and the DSatur heuristic in Algorithm 2. The dataset is random graphs with 25 nodes and edge density 0.9.

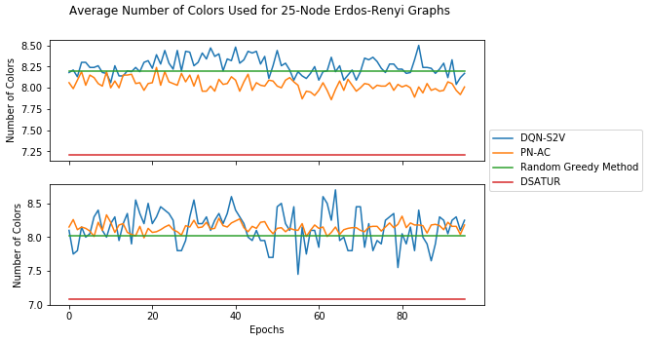

Figure 3. Training (top) and validation (bottom) curves for the four algorithms: DQN-S2V, PN-AC, the random greedy coloring in Algorithm 1, and the DSatur heuristic in Algorithm 2. The dataset is random graphs with 25 nodes and edge density 0.5.

4. Results

4.1. Random Graphs

We generate random graphs of various densities and sizes. In particular, we made graphs with 25 and 50 nodes, each with the probability of forming an edge between any two nodes as 0.5 and 0.9. We generated 100 such graphs for each training dataset and 100 for the testing dataset. To account for the variability characteristic of reinforcement learning algorithms, we ran each experiment (PN-AC and structure2vec) three times on a fixed dataset.

We used Algorithms 1 and 2 as baselines to assess the performance of the structure2vec and PN-AC models. For Algorithm 1, we generated 50 different vertex orderings and chose the most favorable one as the baseline. Figures 2, 3, 4, and 5 show the results averaged over three runs for the RL algorithms.

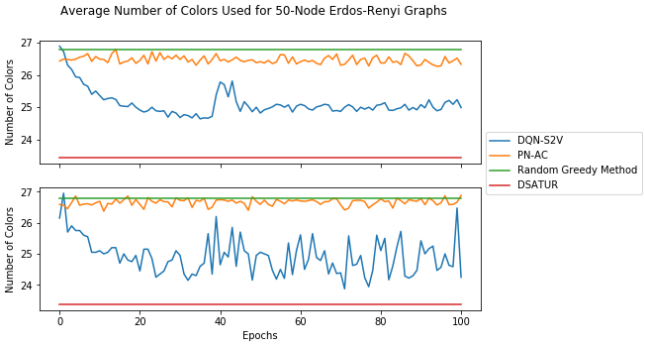

Figure 4. Training (top) and validation (bottom) curves for the four algorithms: DQN-S2V, PN-AC, the random greedy coloring in Algorithm 1, and the DSatur heuristic in Algorithm 2. The dataset is random graphs with 50 nodes and edge density 0.9.

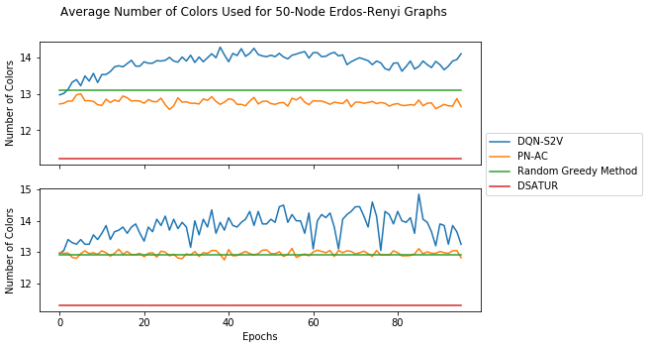

Figure 5. Training (top) and validation (bottom) curves for the four algorithms: DQN-S2V, PN-AC, the random greedy coloring in Algorithm 1, and the DSatur heuristic in Algorithm 2. The dataset is random graphs with 50 nodes and edge density 0.5.

4.2. Transferring Performance

We tested to see how well performance would transfer between random graphs of different sizes and densities as well as to real-world graphs of varying coloring difficulty.

Real-World Dataset	Comparative # of Colors
Small Graphs (23)	-0.39
NP-s (56)	-0.07
NP-m (14)	$+0.01$
NP-h (10)	-0.2

Table 1. Table of how well the S2V RL algorithm did in comparison to the greedy coloring in Algorithm 1 on transfer experiments. The comparative number of colors is the average number of colors S2V used in comparison to the greedy coloring algorithm. We used the S2V algorithm trained on 50 node Erdos-Renyi graphs with 0.9 edge density to test transfer ability. The real-world datasets were stratified by size to form the small graphs dataset, which has all graphs with less than 100 nodes. The NP-s, NP-m, and NP-h datasets indicated how long it took for heuristics to find the best solution on the graph (on the order of s = seconds, m = minutes, h = hours).

Table 1 has the results for transferring from the S2V algorithm to larger and more difficult real-world graphs, drawn from the freely available dataset at `https://sites.google.com/site/graphcoloring/files`, which contains benchmark graphs to test coloring algorithms. We refer readers to that website for further information about the dataset. In addition to the average performance shown above, we also note that the RL algorithm often used one fewer color than the baseline, indicating that performance was boosted across the distribution of graphs instead of just on one graph.

5. Discussion

5.1. Interpreting Results

The S2V algorithm performed noticeably worse on the sparse graphs. We used the standard 5 rounds of message-passing for these graphs, but the algorithm may have benefitted from more rounds on sparser graphs so it could gather the topological information from more distant nodes as well. We saw that on the 25-node dense graphs, the S2V algorithm trained to become better than the DSatur heuristic (Algorithm 2). Moreover, we saw that S2V worked reasonably well on different size graphs.

We also saw that the validation performance did not match the training performance. This may be indicative of a bigger issue with the generalizability of the RL algorithm, but we also note that in light of the transfer experiments, it is difficult to draw any conclusions about this model in that regard without further experimentation.

The transfer experiments showed us that although the real-world dataset in consideration had very sparse graphs (e.g., edge density 0.1) in comparison to the set that S2V was trained on, S2V was able to color them better than or almost as well as greedy algorithms. Moreover, S2V showed promise on the graphs that were classified as harder for greedy algorithms, making it a viable option as a coloring algorithm. In the future, some ensemble method with traditional coloring and the S2V algorithm may yield the optimal performance.

In the 50-node experiments, the algorithm likely would have benefitted from longer training, but this was computationally intensive and we did not have the resources to run that. It would be interesting to see how well the algorithm does when it starts overfitting the graphs in the dataset.

5.2. Challenges

As mentioned previously, runtime was the biggest hinderance in this study. Because each model took so long to train, it was difficult to do a thorough hyperparameter search (e.g., increasing the number of rounds in message-passing for the sparser graphs) for the algorithms. Such a search may yield different performances for the algorithms at their optimal hyperparameter settings.

We also could not run training on the real-world dataset because the graphs had different sizes. As a result, when the algorithm would form a batch to perform the fitted Q-iteration learning step, it would break because the state representations could not be concatenated along those dimensions.

Although we hoped to augment the states in the structure2vec message passing with the colors that each vertex had gotten, we found it difficult to directly extend the algorithm meaningfully to do this. In particular, adding a dimension with the colors would make the interpretation of the states as probabilities over the vertices impossible, since we now have two pieces of information. It was unclear how to do this augmentation meaningfully, but it may greatly boost the performance of the algorithm.

5.3. Future Work

As mentioned above, we want to augment the states of the vertices with the specific colors used so that the message-passing algorithm has more information for each vertex. Furthermore, we hope to be able to extend the algorithm to other datasets and commonly known graph structures, so that we can use more specific heuristic algorithms as baseline algorithms.

References

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016. URL http://arxiv.org/abs/1611.09940.

Brélaz, D. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, April 1979. ISSN 0001-0782. doi: 10.1145/359094.359101. URL http://doi.acm.org/10.1145/359094.359101.

Dai, H., Dai, B., and Song, L. Discriminative embeddings of latent variable models for structured data. *CoRR*, abs/1603.05629, 2016. URL http://arxiv.org/abs/1603.05629.

Grimmett, G. R. and McDiarmid, C. J. H. On colouring random graphs. *Mathematical Proceedings of the Cambridge Philosophical Society*, 77(2):313–324, 1975. doi: 10.1017/S0305004100051124.

Husfeldt, T. Graph colouring algorithms. *CoRR*, abs/1505.05825, 2015. URL http://arxiv.org/abs/1505.05825.

Khalil, E., Dai, H., Zhang, Y., Dilkina, B., and Song, L. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pp. 6348–6358, 2017.

Kučera, L. The greedy coloring is a bad probabilistic algorithm. *Journal of Algorithms*, 12 (4):674 – 684, 1991. ISSN 0196-6774. doi: https://doi.org/10.1016/0196-6774(91)90040-6. URL http://www.sciencedirect.com/science/article/pii/0196677491900406.

Lewis, R. R. *A Guide to Graph Colouring: Algorithms and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2015. ISBN 3319257285, 9783319257280.

Vinyals, O., Fortunato, M., and Jaitly, N. Pointer networks. In *Advances in Neural Information Processing Systems*, pp. 2692–2700, 2015.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, May 1992. ISSN 0885-6125. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.