Coloring Big Graphs with AlphaGoZero

Jiayi Huang [1] Mostofa Patwary [1] Gregory Diamos [1]

Abstract

We show that recent innovations in deep reinforcement learning can effectively color very large graphs – a well-known NP-hard problem with clear commercial applications. Because the Monte Carlo Tree Search with Upper Confidence Bound algorithm used in AlphaGoZero can improve the performance of a given heuristic, our approach allows deep neural networks trained using high performance computing (HPC) technologies to transform computation into improved heuristics with zero prior knowledge. Key to our approach is the introduction of a novel deep neural network architecture (FastColorNet) that has access to the full graph context and requires $O(V)$ time and space to color a graph with V vertices, which enables scaling to very large graphs that arise in real applications like parallel computing, compilers, numerical solvers, and design automation, among others. As a result, we are able to learn new state of the art heuristics for graph coloring.

1. Introduction

Current approaches for quickly solving NP-hard optimization problems like graph coloring rely on carefully designed heuristics. These heuristics achieve good general purpose performance, and are fast enough to scale up to very large problems. However, the best performing heuristic often depend on the problem being solved, and it has been widely recognized that machine learning methods have the potential to develop improved heuristics for specific application domains (Silver et al., 2017b; Dai et al., 2017; Li et al., 2018; Lederman et al., 2018).

In order to learn strong heuristics for specialized applications of graph coloring, we need training data that includes solutions that outperform existing heuristics. However, the extremely large search spaces of even single instances of NP-hard problems like graph coloring presents a significant challenge, and we need a solution that can perform well on such large spaces. Deep reinforcement learning algorithms represent one such approach that has found recent success in games like Go, Chess, and Shogi (Silver et al., 2017a) with very large search spaces. These algorithms use deep neural networks to store knowledge learned during self-play. They build on this knowledge using search procedures such as Monte Carlo Tree Search (MCTS) with Upper Confidence Bound (UCB), leading to better solutions with more training. Although they require many training steps to achieve good performance, the inexpensive evaluation of solutions to NP-hard problems enables very fast training.

In this paper we demonstrate our approach by introducing a framework for learning new heuristics using deep reinforcement learning, depicted in Figure 1. A reinforcement learning algorithm is used to extend the performance of the best existing heuristic on a training set of problems. Training is still NP-hard, even on a single problem. However, high performance computing (HPC) systems can devote large scale computation and significant training time to building stronger heuristics. The best heuristics discovered during training may be distilled into models that are fast to evaluate (e.g. P-TIME, parallel, etc), and stored in a model zoo. These heuristics could be periodically downloaded into production tools, which would then be capable of quickly finding good solutions to the same problems that were encountered during training. We seek to address the following question: "how well do learned heuristics generalize, and for which application domains"?

Our results suggest that a similar approach can be successfully applied to other combinatorial optimization problems, and that our results can be further improved by using even faster training systems to run deep reinforcement learning on larger datasets of representative problems. The specific contributions of this paper are as follows:

- We introduce a framework for learning fast heuristics using deep reinforcement learning algorithms inspired by AlphaGoZero (Silver et al., 2017b) on HPC systems, and use it to learn new graph coloring heuristics that improve the state of the art accuracy by up to 10%.
- We demonstrate how to express the graph coloring problem as a Markov Decision Process and apply the self-play reinforcement learning algorithm in AlphaGoZero to graph coloring in Section 3.

[1]SVAIL, Baidu, USA. Correspondence to: Gregory Diamos <gregory.diamos@gmail.com>.

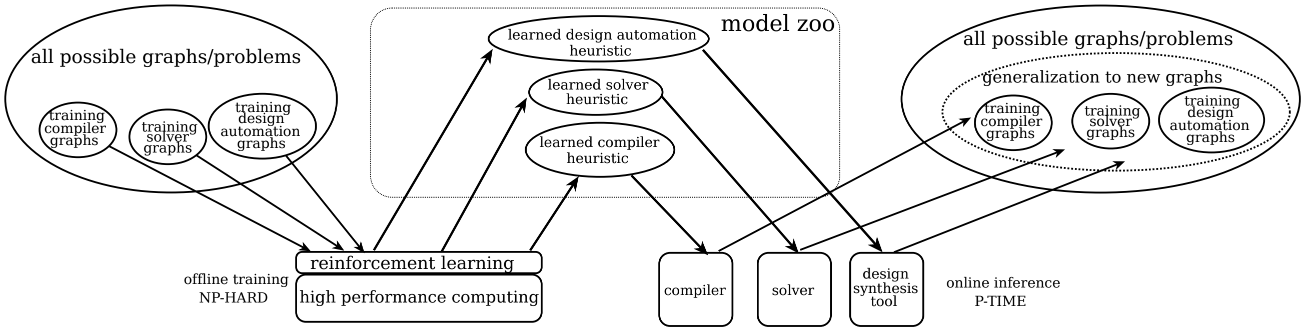
all possible graphs/problems
training compiler graphs
training solver graphs
training design automation graphs
model zoo
learned design automation heuristic
learned solver heuristic
learned compiler heuristic
reinforcement learning
high performance computing
offline training NP-HARD
compiler
solver
design synthesis tool
online inference P-TIME
all possible graphs/problems
generalization to new graphs
training compiler graphs
training solver graphs
training design automation graphs

Figure 1. **Deep Heuristic Learning**. High performance training systems use deep reinforcement learning to improve heuristics offline, which are then deployed online in production tools. These tools focus on speed and solution quality in specific domains.

- We introduce a new highly optimized neural network FastColorNet in Section 3.4 that has access to the full graph context, exposes significant parallelism enabling high performance GPU implementations, and can scale up to very large graphs used in production tools, which have many order of magnitude ($\sim 100K$) larger search space than Go.

2. The Graph Coloring Task

Graph coloring as illustrated in Figure 2 is one of the core kernels in combinatorial scientific computing with many practical applications including parallel computing, VLSI, and pattern matching (Naumann & Schenk, 2012). It assigns colors to the vertices of a given graph $G = (V, E)$ such that no adjacent vertices are of the same color. The objective of the graph coloring problem is to minimize the number of colors used, and it is known to be NP-hard to solve optimally (Gebremedhin et al., 2005). In fact, graph coloring is even NP-hard to approximate in specific scenarios (Zuckerman, 2006). Therefore, linear time greedy algorithms are often used, which yield near optimal solutions in practice (Çatalyürek et al., 2012). It is worthwhile to mention that the order of vertices used in the greedy coloring algorithm plays an important role on the quality of the solutions. Therefore several ordering techniques have been developed, e.g. largest and smallest degree based vertex ordering (Gebremedhin et al., 2013).

In this paper, we use matrix $C_{i,j}$ to represent the assignment of colors to graph G where i represents a vertex in G, and j represents a unique color. The above mentioned greedy coloring heuristics hence progressively update C, each vertex at a time until all vertices in G are assigned colors.

2.1. Graph Coloring as a Markov Decision Process

In order to apply deep reinforcement learning to the graph coloring problem, we represent graph coloring as a Markov Decision Process (MDP). Here, $C^{(t)}$ encodes the MDP state $s^{(t)}$ at step t. Recall that $C^{(t)}$ represents the assignment of colors to vertices at step t. The set of actions, A_i, is the set of valid colors that may be assigned to the next vertex i at step t. All actions are deterministic, and the intermediate reward R_a is the negative total number of colors used so far. Our goal is to learn $\pi_*(s)$, the optimal policy mapping from state $s^{(t)}$ to action a_i, and $V(s^{(t)})$ the expected reward of $s^{(t)}$ while following π_*. It is important to recognize that unlike games of the Go or Chess, graph coloring heuristics need to support diverse graphs, and different graphs imply different MDPs.

2.2. A Zero Sum Game of Graph Coloring

One difference between graph coloring and zero sum games like Go and Chess is the reward. In graph coloring the most obvious choice of reward is the number of colors used. In games like Chess and Go, the reward is typically win, lose, or in some cases tie. In these games it is natural for algorithms to exploit self-play, where the best performing learned algorithm plays against itself.

We experiment with a new reward for graph coloring inspired by self-play. We use the best learned algorithm so far to color the graph. We define a new reward for any solution with fewer colors to win, any solution with more colors to lose, and any solution with the same number of colors to tie. We find that this choice simplifies the design of reward scaling and alpha-beta-pruning, although both reward formats are able to achieve comparable results.

2.3. Is Graph Coloring Harder Than Go?

To put graph coloring in context with other deep reinforcement learning tasks, we estimate the size of the MDPs in terms of number of states for graph coloring as well as Go and Chess by raising the average number of actions per move to the power of the average number of moves on a set of graphs. Table 1 shows that even moderately sized graphs imply very large MDPs, and the largest graphs imply

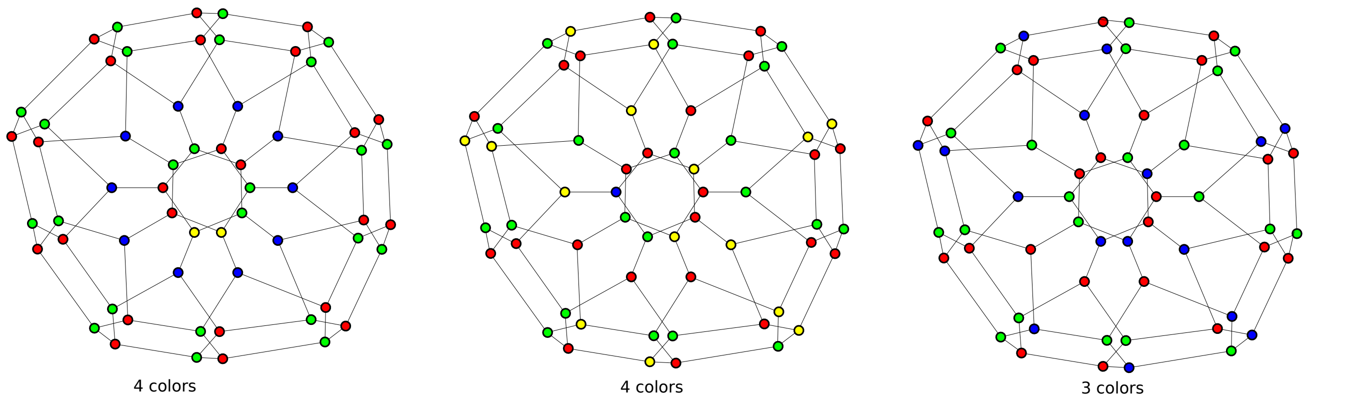

Figure 2. Graph coloring or, more precisely, vertex coloring is a way of assigning colors to the vertices of a graph such that no two adjacent vertices are of the same color. For example, the left and center graphs are colored with greedy heuristics. The right graph is colored optimally. Graph coloring has found many practical applications in diverse domains. The popular game of Sudoku can be expressed as graph coloring.

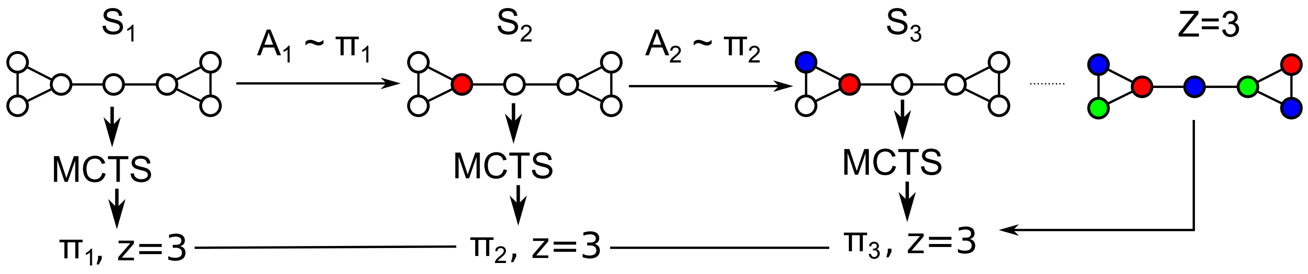

Figure 3. The reinforcement learning algorithm. At each state, a MCTS computes probabilities π_i for the current move, and the next action A_i is selected. When the graph is colored, the final score z is stored as a label for all previous moves.

MDPs that are many order of magnitude larger than Go. As we will see, these much larger problems present significant scalability challenges in training and inference. In this work, we extend the AlphaGoZero approach to address them.

3. Graph Coloring with AlphaGoZero

AlphaGoZero used a deep neural network f_θ with parameters θ to map a representation s of the Go board, pieces, and history to move probabilities and a value, $(p, v) = f_\theta(s)$. p represents the probability of selecting each move, and v is a scalar estimating the probability of the current player winning from position s.

This network was trained using a self-play reinforcement learning algorithm that uses a search procedure (MCTS with UCB) to extend the performance of the neural network. This approach of self-play with search can be viewed as generating new (better) labels (π, z) for the current state s, which the neural network can then be trained on, resulting in an even stronger neural network. This process is then repeated in a policy iteration procedure.

In this work we show how this approach can be applied to graph coloring, depicted in Figure 3. We use a deep neural network f_θ with parameters θ to map a representation s of the graph state and coloring history to next-color probabilities and a value, $(p, v) = f_\theta(s)$. p represents the probability of selecting each valid color next, and v is a scalar estimating the probability of finding a better final solution than the current best heuristic from position s. We use MCTS+UCB to search for better labels (π, z), which the neural network can then be trained on, resulting in an even stronger neural network. This process is then repeated in a policy iteration procedure.

The two biggest differences between our approach and AlphaGoZero are: i) we need our neural network to find single good solutions to many graphs of different sizes and structures, whereas AlphaGoZero needs to handle different opponents on the same board, and ii) our biggest graphs are much bigger (i.e. up to many order of magnitude) than Go both in terms of number of MDP states and number of moves. It was also not obvious to us a-priori that self-play and zero-sum value representations are appropriate choices for graph coloring. In spite of these challenges, we are able to achieve strong results.

	Graph-32	**Chess**	Graph-128	**Go**	Graph-512	Graph-8192	Graph-10^7
Avg. MDP States	10^{21}	$\mathbf{10^{60}}$	10^{141}	$\mathbf{10^{460}}$	10^{790}	$10^{19,686}$	$10^{45,830,967}$
Avg. Moves Per Game	32	**40**	128	**200**	512	8,192	10^7

Table 1. Estimated MDP states for **single** random graphs of various sizes compared to the well known games of Chess and Go. Note that graphs with over 10^4 vertices are common in compilers and with over 10^7 vertices in numerical solvers and design automation. Big graphs imply vastly bigger problems than Go, and they present significant scalability challenges in training and inference.

3.1. Monte Carlo Tree Search

Following the implementation in (Silver et al., 2017b), the Monte-Carlo tree search uses the neural network f_θ as a guide. The search tree stores a prior probability $P(s, a)$, a visit count $N(s, a)$, and a state action-value $Q(s, a)$ at each edge (s, a). Starting from the root state, the next move is selected that maximizes the upper confidence bound $Q(s, a) + U(s, a)$ until a leaf node s_l is reached as shown in the **Selection** panel of Figure 4. The leaf node s_l is expanded as shown in the **Expansion** panel of Figure 4, and is evaluated by the neural network to compute $(P(s_l, *), V(s_l)) = f_\theta(s_l)$, corresponding to the **Simulation** panel of Figure 4. Each edge visited during this process is updated to increment its visit count $N(s, a)$ and set its action-value to the mean value $Q(s, a) = (Q(s, a) * N(s, a) + V(s_l))/(N(s, a) + 1)$. This is shown in the **Backpropagation** panel of Figure 4.

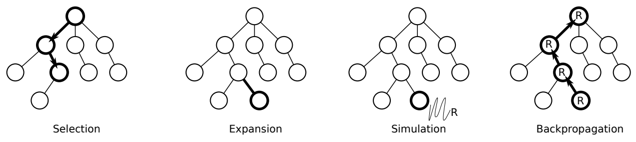

Figure 4. The four stages of the Monte-Carlo Tree Search algorithm. A path is **selected** that maximizes the UCB, the tree is **expanded**, the neural network performs a **simulation**, and the result is **back-propagated** to update nodes along the current path.

3.2. Upper Confidence Bound

A core issue in reinforcement learning search algorithms is maintaining the balance between the exploitation of moves with high average win rate and the exploration of moves with few simulations. We follow the variant of UCB used in (Silver et al., 2017b) for computing $U(s, a)$ and handling this tradeoff.

$$U(s, a) = c * p(s, a) * \frac{\sqrt{\sum_{i=0}^{M} N(s, i))}}{1 + N(s, a)} \tag{1}$$

Here M is the number of available actions at the current state, and c is the exploration factor hyperparameter, typically set between 0.1 and 3.0. The combined MCTS+UCB algorithm initially focuses on moves with high prior probability and low visit count (exploration), and eventually prefers moves with high $Q(s, a)$ value (exploitation).

3.3. Self-Play

To generate labels (π, z) for each move in a new graph G, the graph is first colored by running the neural network over the entire graph, selecting the highest probability move each time, producing a baseline score $\chi(G)$. MCTS+UCB is then used to select each move. The existing search tree along the selected move is reused for the next move. New search probabilities π are computed and the next move is selected by sampling from them. Sampling is used for the first several moves to encourage exploration (controlled by a hyperparameter), then max decoding is used for subsequent moves. We use an approach inspired by alpha-beta-pruning (Knuth & Moore, 1975) to abort plays early that are clearly won or clearly lost compared to the baseline score $\chi(G)$. Results from self play are stored as tuples (G, $C_{i,j}$, π, z), one for each move. The neural network is trained by sampling moves uniformly from a replay buffer (Lin, 1992) of the most recent moves. The replay buffer stores a single copy of the graph for all moves on that graph and lazily materializes a cache of embeddings to save memory.

To handle large graphs, we add two new techniques to self-play: i) *limited-run-ahead*, and ii) *move-sampling*.

Games of Go typically end after a few hundred moves, but some graphs have millions of nodes. Running MCTS all the way to the end is too expensive. So we use a *limited-run-ahead* technique to restrict the MCTS to a limited number of future moves that is typically set to several hundred. To evaluate z, we compare the score of the baseline after the run-ahead-limit with the score produced by the MCTS.

Given the computational cost of training deep neural networks, it is only feasible to train on about 1 billion moves in a reasonable amount of time, even on high performance clusters of accelerators. For example, AlphaGoZero trained on a TPU cluster for about 1.4 billion moves. Given that some graphs have tens of millions of vertices, this would only allow training on about 100 games of self-play. To allow for more diversity of labels, we use *move-sampling*, where we only choose a subset of all moves sampled uniformly from all moves in all graphs in the training set to apply MCTS and to produce training samples. After choosing a move, we run MCTS for several consecutive moves to avoid resetting the search tree. We use a faster (smaller) model to fast-forward to the next sampled move.

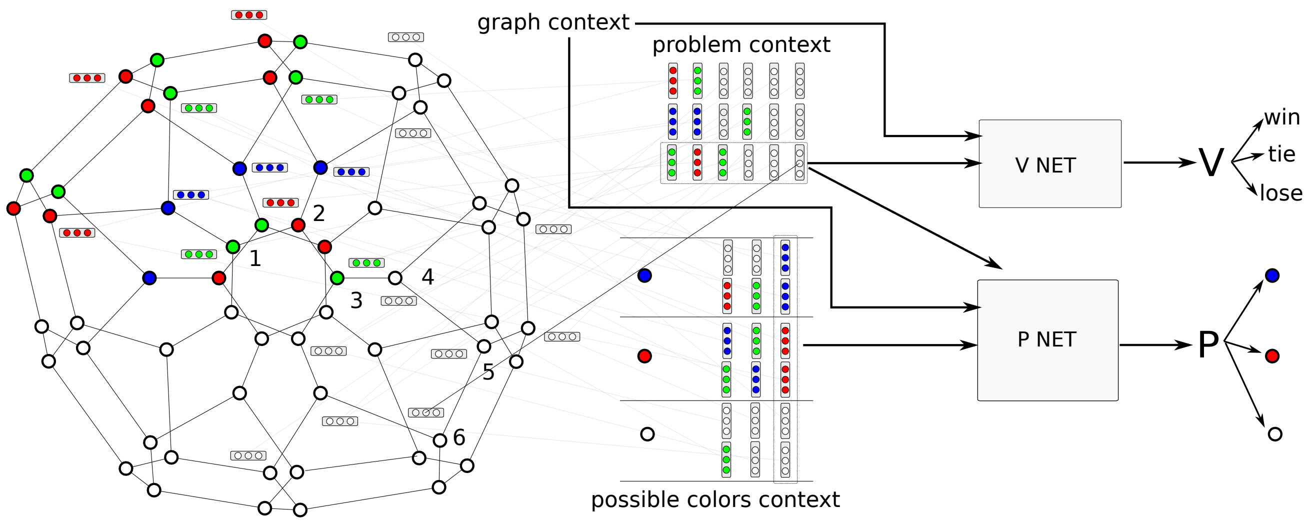

Figure 5. Our FastColorNet architecture computes (p, v) for graph G and colors $C_{i,j}$. The numbered vertices represent the coloring order immediately before and after the current move. Embeddings are gathered from these vertices to form the problem context. Embeddings are also gathered from randomly chosen vertices with each of the allowed colors to form the possible colors context. These are fed into P-Net, and V-Net to compute (p, v) respectively. Truncated and sampled backpropagation is used to update each of the referenced embeddings.

3.4. FastColorNet

Heuristics are used in place of exact solvers because they are fast. In order to compete, we need a neural network that can keep up. This leaves us with the following goals:

- **Scalability**. Fast graph coloring heuristics run in linear $O(V)$ or $O(E + V log V)$ time, enabling scaling to big graphs. For example, some graphs in SuiteSparse have over 10 million vertices.
- **Full Graph Context**. We expect different graphs to require different coloring strategies, so our network needs information about the structure of the graph being colored.

FastColorNet accomplishes both of these goals by i) using a scalable message passing algorithm for computing node embeddings that can capture the global graph structure, and ii) performing a dynamically sized softmax that assigns a probability of selecting each of the valid colors for the current node.

The complete FastColorNet architecture is shown in Figure 5. The network takes graph embeddings and a few simple features such as the number of vertices in the graph as inputs. It predicts V, the expected result of coloring the remainder of the graph, and P, a distribution over available colors for the current node. Note that P is variably sized. FastColorNet is trained end-to-end from V and P labels to embeddings. It used stochastic selection of embeddings and an extension of truncated back propagation through time to keep the computational requirement to color each vertex constant for both training and inference.

We note that FastColorNet represents the first architecture that we were able to design that fulfills all of these requirements while being able to consistently fit the V and P labels produced by the MCTS. We believe that there is room for future work to significantly improve the accuracy of this model with more extensive architecture search.

3.4.1. Graph Embeddings

In order to compute V, information about G and $C_{i,j}$ is required. Simply passing G and $C_{i,j}$ to a fully connected net would be prohibitively expensive and unable to generalize to different graph sizes. We provide this information to FastColorNet through graph embeddings, which are defined per vertex, and are intended to learn relevant local and global information about G and $C_{i,j}$.

Designing graph embeddings that capture local and global graph structure is challenging given the large dynamic ranges of sizes and structures that must be supported with a fixed size vector (Hamilton et al., 2017). Our design is inspired by (Dai et al., 2016; 2018)'s work extending message passing algorithms such as loopy belief propagation to graph embeddings.

Like structure2vec (Dai et al., 2016), we start from near-zero initialized embeddings and run loopy belief propagation using a learned transfer function $\widetilde{\mathcal{T}}$. Intuitively, neighboring vertices exchange messages which are processed by transfer function $\widetilde{\mathcal{T}}$. After enough iterations, this series of messages is likely to converge to a consensus.

Like (Dai et al., 2018), we seek to avoid running back propagation through the entire graph each time an embedding is used, and instead take a sampling approach. Using an analogy of truncated back propagation (Williams & Peng, 1990) applied to graphs, we apply back propagation only along a random walk ending at each referenced vertex embedding.

We represent the transfer function $\widetilde{\mathcal{T}}$ as a neural network (in our case an LSTM) with weights that are learned using back propagation through the embeddings. We experimented with several architectures for the $\widetilde{\mathcal{T}}$ and found training to be more stable with an LSTM than the fully connected net in (Dai et al., 2016).

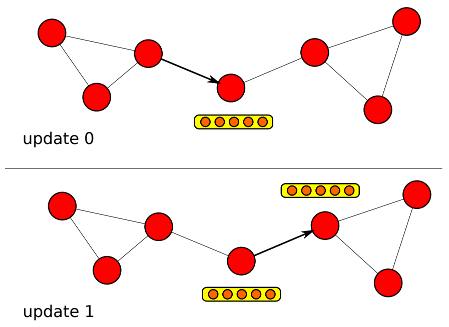

Figure 6. Graph Embeddings are updated for each vertex using the transfer function $\widetilde{\mathcal{T}}$ and one of the neighbor's embeddings. Edges in the graph are selected randomly for multiple iterations to reach convergence.

During inference, we compute embeddings for each vertex by iteratively applying the transfer function to vertices as shown in Figure 6. This approach naturally allows batching up to the size of the graph, making it extremely compute intensive and suitable for GPU or distributed acceleration. The total amount of computation required can be controlled by the number of iterations. In our experiments we run for three iterations. The algorithm is described in Algorithm 1.

3.4.2. Inputs and Outputs

FastColorNet predicts (v, p) for the next move, and is trained on labels (z, π) from the MCTS. Recall that V can be one of the following labels (win, tie, lose), and that π has one entry for each valid color that has already been assigned in addition to a label corresponding to allocating a new color. The loss is the sum of individual cross entropy losses:

$$L(\pi, p, z, v) = \pi^T log(P) + z^T log(v) \qquad (2)$$

Graph context shown in the top of Figure 5 contains the following one-hot encoded values (the number of vertices in G, the total number of assigned colors, the number of

Algorithm 1 Graph Embedding

```
1:  **Input:** parameters $\theta \in \widetilde{\mathcal{T}}$
2:  Initialize $\widetilde{\mu}_i^{(0)} = \mathbf{0}$, for all $i \in \mathcal{V}$
3:  **for** $t = 1$ **to** $T$ **do**
4:      **for** $i \in \mathcal{V}$ **do**
5:          $\nu_i = [d_i, \widetilde{\mu}_i]$, $d_i$ is $i$'s degree (one-hot)
6:          $l_i = [\nu_i, d_j, \widetilde{\mu}_j^{(t-1)}]$, where $j$ = random($\mathcal{N}(i)$)
            $c_i = 0$
7:          **for** $k = 1$ **to** $L$ **do**
8:              $l_i, c_i = LSTM_\theta(c_i, l_i)$
9:          **end for**
10:         $\widetilde{\mu}_i^{(t)} = LSTM_\theta(c_i, v_i)$
11:     **end for**
12: **end for**{fixed point equation update}
13: return $\{\widetilde{\mu}_i^T\}_{i \in \mathcal{V}}$
```

vertices that have already been colored) concatenated with multi-hot encoded set of valid colors for the current vertex.

Problem context shown in the middle of Figure 5 contains the embeddings of vertices that have just been colored, and vertices are scheduled to be colored next, in order. Problem context is zero padded at the beginning and end of the graph. The number of vertices included in the problem context is a hyperparameter that we typically set to 8 in experiments.

Possible colors context shown in the bottom of Figure 5 contains the embeddings of fixed size sets of vertices that have been colored with each of the possible colors. Possible colors context is zero padded if not enough vertices have been assigned to fill out a complete set for one of the possible colors. The set size is a hyperparameter that we typically set to 4 in experiments.

3.4.3. P-Network

The P network architecture is shown in Figure 7. It computes the probability of assigning each of the valid color choices to the current vertex, using the global graph context, the problem context, and local context for each possible color. We draw inspiration from pointer-networks (Vinyals et al., 2015) and represent colors with the embeddings of vertices that have previously been assigned the same colors. In this analogy, the P network selects a pointer to a previously colored node rather than directly predicting a possible color. However, our approach is different than pointer networks because it considers a set of pointers at a time with the same color rather than a single pointer. It is also different because it considers a fixed set of possible colors instead of all previously encountered vertices. These changes are important to exploit locality among nodes with the same color, boosting accuracy, and to bound the computational requirement for very large graphs to linear time in the order of the graph rather than quadratic time for pointer networks.

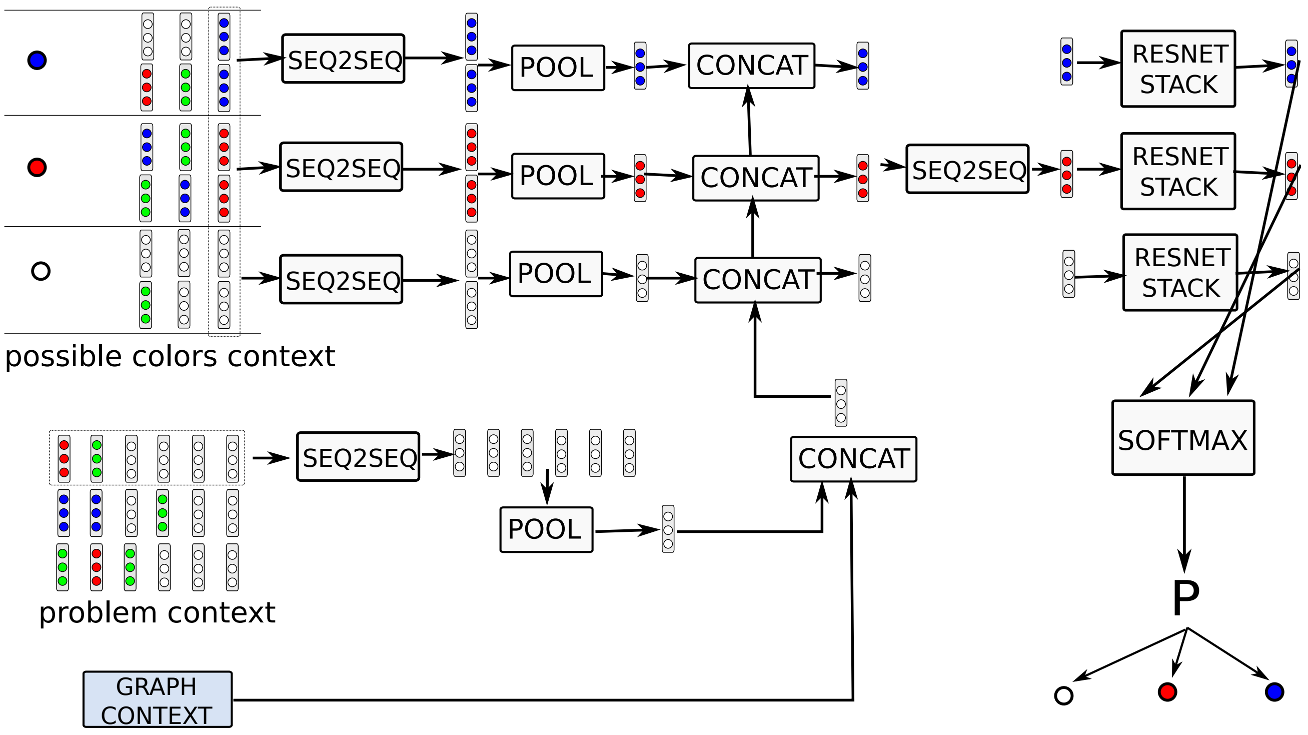

Figure 7. The P-Network handles a dynamic sized set of colors while incorporating coloring problem and graph context using a sequence-to-sequence model. It exchanges information stored in vertex embeddings with each of the valid colors for the next move.

In order to support a dynamic number of possible colors, each color is first processed independently, producing an unnormalized score. This score is then post processed by a sequence-to-sequence model that incorporates dependencies on the other possible colors. The final scores are normalized by a softmax operation.

3.4.4. V-Network

The V network architecture shown in Figure 8 maps from the graph context and the problem context to the expected outcome of the coloring problem. The vertex embeddings in the problem context are stored in a sequence corresponding to the order that vertices are colored. We use a sequence-to-sequence model to exploit locality in this sequence, followed by a pooling operation over the sequence to summarize this information. We typically use stacks of residual 1-D convolution layers for the sequence-to-sequence model. This local information is then concatenated together with the global graph context and fed into a deep stack of fully connected relu layers with residual connections between each layer. The result is fed into a softmax, which selects the label from the (win, lose, or tie) outcomes.

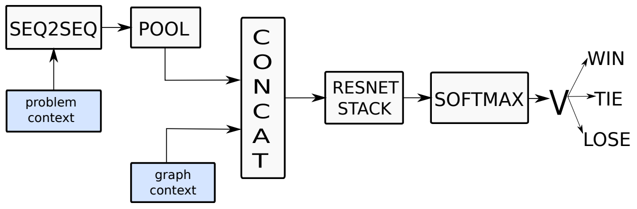

Figure 8. The V-Network estimates the outcome of the coloring problem from the current state using embedding information about the recently colored nodes as well as information about the graph.

3.5. High Performance Training System

The computational requirements of training our networks on even a single large graph are vast. In order to perform experiments quickly, we built a highly optimized training system. The system uses tightly integrated compute nodes composed of commodity off-the-shelf GPUs, high performance networking, and high bandwidth IO system designed to serve training data.

This system is organized into **AI POD**s as shown in Figure 9. Dense compute nodes with 10 1080Ti or 2080Ti GPUs are connected using FDR Infiniband. A high performance IO node uses an SSD array to serve training data to the compute nodes at up to 7 GB/s. Collectively the POD can sustain

3.12 single-precision PFLOP/s when running a single large training job. When configured using 2080Ti GPUs, a single POD can sustain 32.1 mixed-precision PFLOP/s.

We use data-parallelism for training on multiple GPUs using synchronous stochastic gradient descent with MPI for communication. FastColorNet is designed to have high enough compute intensity to enable high GPU efficiency with a small batch size of 1-4 per GPU. Using the entire AI POD requires a maximum batch size of 1200. The MCTS is completely data-parallel, with each data-parallel worker generating independent samples from different graphs. Data parallelism is also used within a single GPU for inference to batch model evaluations for different graphs.

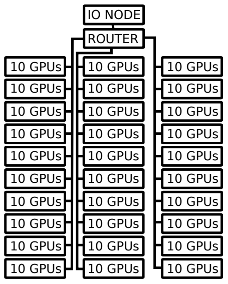

Figure 9. The AI POD architecture, containing up to 300 tightly integrated GPUs with high bandwidth access to training data.

4. Empirical Analysis of Results

We applied this reinforcement learning process to train FastColorNet on a diverse set of commonly studied graph coloring problems, including synthetic Erdős–Rényi (Erdos & Rényi, 1960) and Watts–Strogatz (Watts & Strogatz, 1998) (sometimes called small-world) graphs, as well as the SuiteSparse collection (Davis & Hu, 2011). Graph sizes varied from 32 vertices to 10 million vertices.

A single 1080Ti GPU was able to train FastColorNet on about 300,000 mini-batches of 4 moves in one day for a medium size graph with 512 vertices. We use Adam optimizer for training with a fixed learning rate of 0.001, and did not use learning rate annealing or explicit regularization. We experimented with a variety of architectures for different datasets, and typically used stacks of residual 1D-CONV layers with batch normalization for sequence to sequence models, layer sizes ranged from 64-1024, and batch normalization resnet stacks with 2-20 layers. We report results from the best performing architecture for each test set.

To assess performance over a range of graph sizes, Figure 10 shows how the number of required colors grows with the graph size for the small-world test set . FastColorNets in this experiment were trained on similar small-world graphs using the same graph generator parameters. FastColorNet significantly improves on the best baseline.

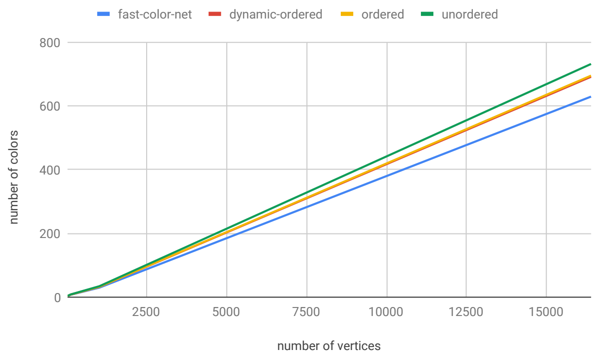

Figure 10. The number of colors used on the Watts–Strogatz (small-world) graph test sets as a function of vertex count. FastColorNet-train achieves the best performance, and the gap widens for bigger graphs.

To explore how learning policies improve with more computation in training, Figure 11 shows the performance improvement of FastColorNet on Erdős–Rényi graphs as a function of the number of policy iteration steps. Performance matches the best heuristic in under 20 steps, and significantly exceeds it in under 100 steps.

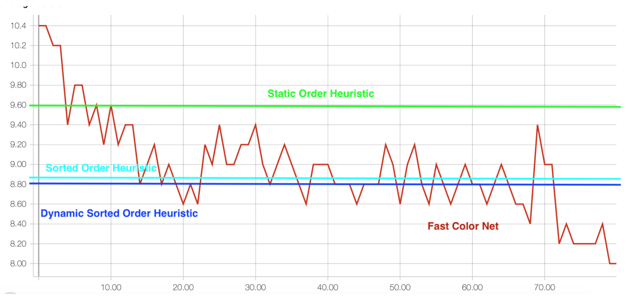

Figure 11. Improvement in learned heuristic with more training.

Table 2 shows the performance of FastColorNet on test sets of different sizes from different application domains. Test sets have 20 graphs each, representing 32K-16M moves. [1] The FCN-train represents performance when the learned heuristic is used on a graph in the training set, showing that our reinforcement learning pipeline can improve on heuristics by $5\% - 10\%$ with more training time. Although training on the test set might seem like a bad idea, this is an important use-case for some production tools, e.g., compilers that are repeatedly used on the same function or program, e.g. the linux kernel. FCN-test trains on graphs of the same size randomly sampled from the same domain that are not included in the test set. Performance is also $1\% - 2\%$ better than the heuristics, showing the potential to improve performance through domain specialization. FCN-gen is trained

[1] While FastColorNet runs in linear time, our python inference code is not optimized, and we did not have time to include test sets with millions of vertices.

	ER-32	SW-32	ER-1K	SW-1K	ER-16K	SW-16K	SS-CIR	SS-LP	SS-Web	SS-FE
Unordered	3.75	5.35	34.3	59.2	732.8	265.35	4.2	4.25	3.75	4.85
Ordered	3.1	4.5	32.45	57.35	715.2	261.8	3.15	**2.95**	2.6	4.05
Dynamic	3.25	4.55	32.2	57.15	708.5	261.2	3.55	3.15	2.7	4.25
FCN-train	**2.95**	**4.08**	**29.58**	**52.5**	**660.19**	**237.03**	**3.0**	**2.95**	**2.4**	**3.75**
FCN-test	3.03	4.3	31.7	56.59	702.57	258.3	3.1	**2.95**	2.55	4.1
FCN-gen	3.65	5.25	33.9	57.66	708.13	267.53	4.15	4.3	3.7	4.95

Table 2. The average number of colors across our test sets. ES abbreviates Erdős–Rényi, and SW abbreviates Watts–Strogatz. FCN is our FastColorNet architecture. SS means SuiteSparse, CIR are graphs labeled as circuits, LP are labeled linear programming, and FE are labeled finite-element. FCN-train represents performance when a graph present in the training set is evaluated on, FCN-test uses a model trained on the same type of graph, and FCN-gen tests generalization performance of a model trained on random graphs of many sizes.

on random graphs. It improves on the unordered heuristic (it itself is unordered), but loses to the other heuristics.

5. Related Work

Deep RL for Games. Reinforcement learning has been researched for decades for games (Schraudolph et al., 1994; Tesauro, 1995). Mnih et al. proposed DQN, a combination of deep neural network and q-learning with experience replay, to achieve human-level performance on Atari games (Mnih et al., 2015). More recently, Deepmind published a series of AlphaGo algorithms for more complex game Go and defeated human experts (Silver et al., 2016; 2017b). They apply MCTS to explore the large MDP state space while balancing explore and exploit by using UCB for decision selection. In addition, alpha-beta-pruning (Knuth & Moore, 1975) is also adopted to early stop game play that is known to win or lose to reduce tree search space. Our approach for graph coloring takes the similar one of AlphaGoZero. Furthermore, in order to learn for much bigger problems compared to the relatively small one of Go, we apply other innovations to make our solution scalable and computational efficiently.

RL/ML for Combinatorial Optimization. Recently, reinforcement learning has been applied for combinatorial optimization. Bello et al. combined pointer networks (Vinyals et al., 2015) with actor and critic network to optimize Traveling Salesman Problem (TSP) (Bello et al., 2016), which does not make good use of graph structure and is not generalized to arbitrary size graphs. In (Dai et al., 2017), a Q-learning framework is introduced for greedy algorithms to learn over MVC, MAXCUT and TSP problems using structure2vec (Dai et al., 2016) graph embedding. This algorithm cannot be directly applied to graph coloring since the reward design and state representation for colored graphs is non trivial in its problem formulation. Both algorithms only evaluated on small graphs and are not scalable to big graphs, which are typical in real applications. In contrast, our approach is easy to scale to train on bigger graphs to solve bigger problems in graph coloring applications.

Graph Coloring. Graph coloring are being studied for several decades due to its usefulness in many practical applications, including linear algebra, parallel computing, resource assignment and register allocation (Naumann & Schenk, 2012). The graph coloring problem is known to be NP-Hard and so is its approximation (Gebremedhin et al., 2005). Heuristics are therefore widely used, which include greedy algorithms, finding independent sets, local search and population based algorithms (Johnson et al., 2012). An in-depth experimental study has been presented in (Gebremedhin et al., 2013), which demonstrated that greedy coloring algorithm with appropriate vertex ordering gives close to optimal coloring in reasonable time on wide range of applications. In this paper, we used the implementation of (Gebremedhin et al., 2013) when comparing our reinforcement based technique. Several variations of graph coloring (e.g. star and acyclic coloring) and heuristics algorithms are studied in (Gebremedhin et al., 2005; 2009; 2007).

6. Conclusions

Our results demonstrate that deep reinforcement learning can be applied to large scale problems of clear commercial importance, such as the well-known graph coloring problem. The MCTS+UCB algorithm used in AlphaGoZero defeated state of the art heuristics by a large margin. The learned heuristics generalized to new graphs in the same application domain, and were distilled into a model (FastColorNet) that is fast enough to color very large graphs.

References

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. Neural combinatorial optimization with reinforcement learning, 2016.

Çatalyürek, Ü. V., Feo, J., Gebremedhin, A. H., Halappanavar, M., and Pothen, A. Graph coloring algorithms for multi-core and massively multithreaded architectures.

Parallel Computing, 38(10-11):576–594, 2012.

Dai, H., Dai, B., and Song, L. Discriminative embeddings of latent variable models for structured data. In Balcan, M. F. and Weinberger, K. Q. (eds.), *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pp. 2702–2711, New York, New York, USA, 20–22 Jun 2016. PMLR.

Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., and Song, L. Learning combinatorial optimization algorithms over graphs, 2017.

Dai, H., Kozareva, Z., Dai, B., Smola, A., and Song, L. Learning steady-states of iterative algorithms over graphs. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1106–1114, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL `http://proceedings.mlr.press/v80/dai18a.html`.

Davis, T. A. and Hu, Y. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

Erdos, P. and Rényi, A. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

Gebremedhin, A. H., Manne, F., and Pothen, A. What color is your jacobian? graph coloring for computing derivatives. *SIAM review*, 47(4):629–705, 2005.

Gebremedhin, A. H., Tarafdar, A., Manne, F., and Pothen, A. New acyclic and star coloring algorithms with application to computing hessians. *SIAM Journal on Scientific Computing*, 29(3):1042–1072, 2007.

Gebremedhin, A. H., Tarafdar, A., Pothen, A., and Walther, A. Efficient computation of sparse hessians using coloring and automatic differentiation. *INFORMS Journal on Computing*, 21(2):209–223, 2009.

Gebremedhin, A. H., Nguyen, D., Patwary, M. M. A., and Pothen, A. Colpack: Software for graph coloring and related problems in scientific computing. *ACM Transactions on Mathematical Software (TOMS)*, 40(1):1, 2013.

Hamilton, W. L., Ying, R., and Leskovec, J. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

Johnson, D., Mehrotra, A., and Trick, M. Color02/03/04: Graph coloring and its generalizations, 2012.

Knuth, D. E. and Moore, R. W. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

Lederman, G., Rabe, M. N., and Seshia, S. A. Learning heuristics for automated reasoning through deep reinforcement learning. *CoRR*, abs/1807.08058, 2018. URL `http://arxiv.org/abs/1807.08058`.

Li, Z., Chen, Q., and Koltun, V. Combinatorial optimization with graph convolutional networks and guided tree search. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31*, pp. 537–546. Curran Associates, Inc., 2018.

Lin, L.-J. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 2015.

Naumann, U. and Schenk, O. *Combinatorial scientific computing*. CRC Press, 2012.

Schraudolph, N. N., Dayan, P., and Sejnowski, T. J. Temporal difference learning of position evaluation in the game of go. In Cowan, J. D., Tesauro, G., and Alspector, J. (eds.), *Advances in Neural Information Processing Systems 6*, pp. 817–824. Morgan-Kaufmann, 1994.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017a. URL `http://arxiv.org/abs/1712.01815`.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017b.

Tesauro, G. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995. ISSN 0001-0782.

Vinyals, O., Fortunato, M., and Jaitly, N. Pointer networks. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 28*, pp. 2692–2700. Curran Associates, Inc., 2015.

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

Williams, R. J. and Peng, J. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, 2(4):490–501, 1990.

Zuckerman, D. Linear degree extractors and the inapproximability of max clique and chromatic number. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pp. 681–690. ACM, 2006.