ENS 491-492 – Graduation Project (Implementation)

Draft Final Report

Project Title : Reinforcement Learning for Graph Coloring

Group Members :

23653 - Arda Aşık

24071 - Barış Batuhan Topal

20652 - Berker Demirel

23744 - İbrahim Buğra Demir

Supervisor(s) : Kamer Kaya

Date : 03.05.2020

Contents

Executive Summary

Combinatorial Optimization is a fundamental problem in computer science, which deals with finding the optimal solution from a set of finite states or objects. Some of the most popular examples are finding shortest/cheapest round trips (TSP), finding models of propositional formulae (SAT), coloring graphs (GCP), etc. These problems are all NP-hard, and thus, have drawn considerable interest from the theorists and algorithm designers over the years. There are 3 main ways to approach an NP-hard graph optimization problem: exact algorithms, approximation algorithms, and heuristics. Exact algorithms are based on branch and bound with linear or integer programming, which can find the optimal solution but in exponential time. Approximation algorithms find optimal solutions within a range with polynomial time. They may be desirable but suffer from the quality of solution and empirical performance. Heuristics are usually fast but do not give theoretical guarantees.

In this project, we focused on the Graph Coloring Problem[1], which deals with assigning colors to vertices of a graph, where no adjacent vertices have the same colors. To address the solution, we simulated some heuristic algorithms and compared their differences. After that, we were able to analyze different heuristics to come up with a weighted algorithm that contained all the heuristics we learned to give a better solution. In the end, we were able to deeply understand the problem and implement a reinforcement learning model using deep Q-networks which is an action-reward based neural network method to solve the problem in a more efficient way.

[1] Our codes can be accessed from: https://github.com/barisbatuhan/Graph_Coloring_with_RL

Problem Statement

Graph Coloring Problem (GCP) is an NP-Hard problem [4], where color is chosen for each vertex in a way that the total number of colors is minimized and no neighboring vertices have the same color. There are 3 main approaches for solving GCP, but none of them exploit the structure of graphs since they solve the problem over and over again while only changing the data. In this project, we tried to build a reinforcement learning (RL) model, which is an action-reward system that helps machines to learn the given environment and model, in order to solve similar and more complex problems [7], to solve GCP by the exploitation of graph properties.

In literature, there are papers on using RL for combinatorial optimization like Learning Combinatorial Optimization over Graphs [8], Neural Combinatorial Optimization with RL [7], etc. in which Travelling Salesman, Minimum Vertex Cover, Maximum Cut problems are tackled. It can be argued that applying RL to NP-Hard problems is a hot topic but our problem -GCP- has not been investigated widely in the literature. There is a related work that has been done in Coloring Big Graphs with AlphaGoZero [9]. However, due to its large computational cost and considering the hardware they use for scaling the algorithm, that approach might not be tractable for especially small systems. On the other hand, the solution we propose focuses more on the graph topology rather than exhaustively searching for possible permutations of colors. Understanding and using graph structure is crucial, because even without proposing a learning algorithm, the coloring sequence formed using node attributes, changes performance significantly.

A) Objectives & Tasks

- **Project Planning & Initiation:**
  - Meeting Scheduling
  - Exact Problem Definition
  - Literature Research

- **Developing a Baseline:**
  - Repository on GitHub
  - Collecting Sample Graphs
  - Selecting and Implementing Heuristics to Use

- **Model Design:**
  - Creating a Weighted Ordering Model
  - Testing the Weighted Ordering Model
  - Research on RL and Existing RL Models
  - Creating the Initial RL model

- **Final Design and Evaluation:**
  - Testing and Improving the RL Model
  - Comparisons with Heuristics
  - Evaluation

B) Realistic Constraints

- **Hardware Constraints:**

During this project, we worked on one of the HPC clusters of Sabancı University (named Gandalf) which has 60 Intel(R) Xeon(R) E7-4870 v2 @ 2.30GHz as CPU and Nvidia Tesla K40c as GPU which has 2880 CUDA cores. Especially during the analysis of heuristics on the distance-2 coloring problem, we have faced memory inefficiency problems. Hence, we had to implement our program by considering this limitation and discard some of the large-sized graphs from our dataset.

- **Runtime Related Issues:**

Analysis of heuristics on large-sized graphs is too expensive. To solve this issue, parallelization is applied to each method and an approximator function is constructed for the ones that are still expected to take too long such as closeness centrality. However, processing the results on large-sized graphs takes several days. To overcome this problem, these methods may also be parallelized by using GPU and SIMD operations in our future works.

- **Knowledge Constraints:**

The problem requires research in multiple disciplines mainly in graph structures, combinatorial optimization problems, and reinforcement learning models based on deep learning. Therefore, we had to compensate for our deficiency in these fields.

Methodology

In order to start the project, the exact problem definition and the main focus areas had to be determined first, since there are various graph and coloring types. Therefore, we decided to narrow our implementation as first fit greedy distance-1 and 2 colorings [2] of undirected and unweighted graphs.

Next, a GitHub repository is built and a dataset is collected from Suite-Sparse Matrix Collection [10] that consists of 192 small-sized and 176 large-sized with the number of nodes ranging from ~10 to ~620 and ~80000 to ~1000000. Small-sized graphs are chosen especially for having the option to run an optimal coloring algorithm implemented with CPLEX [11] and to compare the results with our findings in next steps (since the optimal solution is NP-hard, time and resources required to compute the solution increase significantly for graphs with more number of nodes than the range specified above).

Among the group members and the instructor, a baseline was planned to be built on degree 1&2&3 orderings, closeness centrality, clustering coefficient and page rank vertex ordering heuristics, since these focus on different aspects of graphs, respectively:

- **Degree 1 & 2 & 3 Orderings** **:** properties about each node's neighbors
- **Closeness Centrality** **:** how central a node is
- **Clustering Coefficient** **:** how nodes are clustered
- **Page Rank** **:** how popular a node is

Thus, these algorithms are implemented and run on the dataset. All these heuristic are explained below:

- **Degree One Ordering:** For each node, its number of neighbors is calculated.

$$nbor_1(v) = \{ u : \{v, u\} \in E \}$$

Algorithm 1: Degree One Ordering

Input:
$g \leftarrow$ a graph to analyse
Output:
$ordering \leftarrow$ array of pairs, first index holds node number, second one holds value
Algorithm:
for $node \in g.nodes$ *in parallel* **do**
  ordering $\leftarrow$ insert $\{node, g.degree\_of(node)\}$
end

Figure 1: Degree One Ordering

- **Degree Two Ordering:** For each node, the number of nodes that have a distance less than or equal to 2, is calculated.

$$nbor_2(v) = \{ u \, \varepsilon \, V, \; d(v, u) \leq 2 \}$$

Algorithm 2: Degree Two Ordering

Input:
$g \leftarrow$ a graph to analyse
Output:
$ordering \leftarrow$ array of pairs, first index holds node number, second one holds value
Algorithm:
for $node \in g.nodes$ *in parallel* **do**
  $distances \leftarrow$ BFS(g, node)
  $count \leftarrow$ number of nodes in $distances$ with distance ≤ 2
  ordering $\leftarrow$ insert $\{node, count\}$
end

Figure 2: Degree Two Ordering

- **Degree Three Ordering:** For each node, the number of nodes that have a distance less than or equal to 3, is calculated.

$$nbor_3(v) = \{ u \, \varepsilon \, V, \; d(v, u) \leq 3 \}$$

Algorithm 3: Degree Three Ordering

Input:
$g \leftarrow$ a graph to analyse
Output:
$ordering \leftarrow$ array of pairs, first index holds node number, second one holds value
Algorithm:
for $node \in g.nodes$ *in parallel* **do**
  $distances \leftarrow$ BFS(g, node)
  $count \leftarrow$ number of nodes in $distances$ with distance ≤ 3
  ordering $\leftarrow$ insert $\{node, count\}$
end

Figure 3: Degree Three Ordering

- **Closeness Centrality:** For each node, the sum of total distances of it to other nodes is calculated. The smaller is the total sum, the more central is the node.

$$ccent(v) = \frac{|V|}{\sum_{u \varepsilon V} d(v,u)}$$

Algorithm 4: Closeness Centrality

Input:
$g \leftarrow$ a graph to analyse
Output:
$ordering \leftarrow$ array of pairs, first index holds node number, second one holds value
Algorithm:
for *node* $\in g.nodes$ *in parallel* **do**
  $distances \leftarrow$ BFS(g, node)
  $sum \leftarrow$ sum of all distances in $distances$
  if *sum is 0* **then**
    ordering $\leftarrow$ insert $\{node, 0\}$
  else
    ordering $\leftarrow$ insert $\{node, g.num\_nodes/sum\}$
  end
end

Figure 4: Closeness Centrality

- **Clustering Coefficient:** For each node, it measures the ratio of the actual number of triangles present and the possible number of triangles can be formed at most, by using the node selected for both cases.

$$ccent(v) = \frac{|\{\{u,w\} \in E : u,w \in nbor_1(v)\}|}{|nbor_1(v)| \, x \, (|nbor_1(v)| - 1)}$$

Algorithm 5: Clustering Coefficient

Input:
$g \leftarrow$ a graph to analyse
Output:
$ordering \leftarrow$ array of pairs, first index holds node number, second one holds value
Algorithm:
for *node* $\in g.nodes$ *in parallel* **do**
  $deg \leftarrow$ g.degree_of(node)
  $num\_links \leftarrow 0$
  $possible\_links \leftarrow$ deg x (deg - 1) / 2
  for *adj* $\in$ *neighbors of node* **do**
    for $adj_2 \in$ *neighbors of adj* **do**
      if $adj_2 \in$ *neighbors of node* **then** $num\_links \leftarrow$ num_links + 1;
    end
  end
  if *num_links is 0* **then**
    num_links $\leftarrow$ possible_links + 1
  else
    num_links $\leftarrow$ possible_links / num_links
  end
  ordering $\leftarrow$ insert $\{node, num\_links\}$
end

Figure 5: Clustering Coefficient

- **PageRank:** Provides the possibility of being in a node, while randomly traveling through edges. Therefore, the more popular a node is, the higher is the probability to visit that node.

$$pr_i(v) = \frac{1-\alpha}{|V|} + \alpha\, x \sum_{u\ \varepsilon\ nbor_1(v)} \frac{pr_{i-1}(u)}{nbor_1(u)}$$

Algorithm 6: Page Rank

Input:
$g \leftarrow$ a graph to analyse
$n \leftarrow$ number of nodes in the graph
$iter \leftarrow$ number of iterations to repeat
$\alpha \leftarrow$ learning rate
Output:
$ordering \leftarrow$ array of pairs, first index holds node number, second one holds value
Algorithm:
$ordering[v] \leftarrow \{v, 1/n\}$ for each v $\in$ [0, n)
for *i from 0 to iter* **do**
  $ordering_{copy}[t] \leftarrow \{t, (1-\alpha/n)\}$ for t $\in$ [0, n)
  for *node* $\in g.nodes$ **do**
    $pr_{node} \leftarrow ordering_{copy}[node].second$
    for *adj* $\in$ *neighbors of node* **do**
      $pr_{adj} \leftarrow$ ordering[adj].second
      $degree_{adj} \leftarrow$ g.degree_of(adj)
      $pr_{node} \leftarrow pr_{node} + \alpha$ x $(pr_{adj}/degree_{adj})$
    end
  end
  ordering $\leftarrow ordering_{copy}$
end

Figure 6: PageRank

The greedy graph coloring algorithm is applied to each graph, based on the orderings computed by the algorithms above. The results are compared with the values retrieved from the optimal coloring calculations. A success value is computed through the formula below:

$$sval_{heuristic} = \frac{\sum_{g\ \varepsilon\ G} \frac{result_{greedy,\,heuristic}(g)}{result_{optimal}(g)}}{|G|}\ ,\ G: \textit{all selected graphs in dataset}$$

After that, an ensembling method -weighted ordering model- is proposed, which linearly combines all previous algorithms. After normalizing the values obtained from each single heuristic by using $\frac{value-\mu}{\sigma}$ (μ: mean, σ: standard deviation), a weighted result value is obtained for each node by [6]:

$$val_{node} = \sum_{h\ \varepsilon\ heuristics} w_h + val_h(node)$$

To find the best weight values, a brute force search is applied for each heuristic's weight starting from 0 to 1, with a step size of 0.05. The closest result compared to the optimal coloring is found with the following weights:

Algorithm	Weights
Degree 1 Ordering	0.15
Degree 2 Ordering	0
Degree 3 Ordering	0.01
Clustering Coefficient	0.05
Closeness Centrality	0.7
Page Rank	0

Figure 7: Weights of Weighted Ordering Model

In order to evaluate our work on different scales, large graphs in our dataset are also tested with these weight values. However, since it is too expensive to find the optimal color number for large graphs, a new evaluation metric is introduced. Instead of taking the ratio of the heuristic results with the optimal number of colors, the values obtained are divided to degree-1 ordering results for distance-one coloring (in only large graphs) and degree-2 ordering results for distance-two coloring (for both small and large-sized graphs):

$$sval_{heuristic,\ divider} = \frac{\sum_{g\ \varepsilon\ G} \frac{result_{greedy,\ heuristic}(g)}{result_{divider}(g)}}{|G|},\ G : \textit{all selected graphs in dataset}$$

In distance-1 coloring, ~3% improvement is achieved compared to the best-resulting heuristic from the ones mentioned above (degree-1 ordering) and in distance-2 coloring, ~1% improvement is seen in large graphs (compared to closeness centrality). A detailed table of outcomes can be seen in the Results & Discussion part.

Until now, our work included only static orderings, which means; an order is found first, then the graph is colored regarding that order. However, since our RL algorithm will do the coloring dynamically, we thought that also working on heuristics that create dynamic orderings during the coloring phase, is also necessary for our further evaluations. For this purpose, incidence-degree ordering and saturation-degree ordering heuristics are implemented.

- **Incidence-Degree Ordering**

While coloring the graph, the nodes with the most number of colored neighbors should be colored first [1]. Ordering changes during the coloring phase, therefore this process can be counted as a dynamic ordering. A pseudo-code of the implementation can be seen below:

Algorithm 7: Incidence Coloring

Input:
$g \leftarrow$ a graph to analyse
$ordering \leftarrow$ array of pairs, first index holds node number, second one holds value
Output:
$total_{colors} \leftarrow$ total number of colors used for coloring the graph, initially -1
Algorithm:
$colors_{node}[t] \leftarrow$ number of adjacents of t which are colored, initially 0
$max\_coloreds \leftarrow$ max heap of nodes regarding to the values in $colors_{node}$
while *not all nodes are colored* **do**
  $node \leftarrow$ max_coloreds[0]
  $color \leftarrow$ select greedily the first fitting color to node
  if $color > total_{colors}$ **then** $total_{colors} \leftarrow color$;
  for *adj in neighbors of node* **do**
    $colors_{node}[adj] \leftarrow colors_{node}[adj] + 1$
  end
  Update $max\_coloreds$
end

Figure 8: Incidence Ordering

- **Saturation-Degree Ordering:**

Saturation ordering follows a similar logic with incidence ordering. However, instead of storing the total number of colors of each node's neighbors, only the different colored neighbors of each node are counted [1].

Algorithm 8: Saturation Coloring

Input:
$g \leftarrow$ a graph to analyse
$ordering \leftarrow$ array of pairs, first index holds node number, second one holds value
Output:
$total_{colors} \leftarrow$ total number of colors used for coloring the graph, initially -1
Algorithm:
$colors_{node} \leftarrow$ array of sets holding different colors of nodes' neighbors, initially empty
$max\_coloreds \leftarrow$ max heap of nodes regarding to the sizes of sets in $colors_{node}$
while *not all nodes are colored* **do**
  $node \leftarrow$ max_coloreds[0]
  $color \leftarrow$ select greedily the first fitting color to node
  if $color > total_{colors}$ **then** $total_{colors} \leftarrow color$;
  for *adj in neighbors of node* **do**
    $colors_{node}[adj].insert(adj)$
  end
  Update $max\_coloreds$
end

Figure 9: Saturation Ordering

For the heuristic we call as weighted, the calculation process remained the same as defined earlier. However, we added incidence-degree as one of the measurements along with the ones we have chosen beforehand. The updated weights are below for all the ordering heuristics we have used in weighted method:

Color Dist.	Incidence	Degree 1	Degree 2	Degree 3	Closeness	Clustering	Page Rank
1	0.9	0	0.1	0	0	0	0
2	0.1	0	0	0.05	0.85	0	0

Figure 10: Updated Weights of Weighted Model

But using these weights on our weighted model did not create a significant improvement compared to previous weighted model results.

Then, we started to think about adjusting our reinforcement learning formulation to the GCP. Basic reinforcement is modeled with Markov Decision Process (MDP) using the following definitions [3].

- **S** is a set of environment and agent states.
- **A** is a set of actions of the agent.

- **$Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$** is the probability of transition from state s to s´ under action a.
- **$R_a(s, s')$** is the immediate reward after transition from s to s´ with action a.
- **π**, or a policy p is a probability distribution, or mapping over actions given states. That is the likelihood of every action when an agent is in a particular state. The goal of the reinforcement learning algorithms is to discover an optimal policy.

We decided to formulate GCP as Markov Decision Process as follows:

- ***State:*** A state S is the embedding of the graph, which we will mention later in the report.
- ***Action:*** An action v is the next node to be colored, that has not been colored yet.
- ***Probability*****:** Transition between states are deterministic.
- ***Reward:***. The reward, r(St, v), changes in the number of different colors with respect to the previous step.
- ***Policy:*** Based on Q-function, the deterministic greedy policy is going to be used as the equation below. The neural network structure is chosen to find the true parameters of the Q function.

$$\pi(v \mid S) = argmax_{v'} Q\,(S, v)$$

- ***Embeddings:*** Graphs and nodes are going to be represented as their vector embeddings.
- ***Node Embedding:*** Our node feature embedding contains some static values like its number of distance 1, 2, and 3 neighbors, closeness centrality, clustering coefficient, page rank value, as well as dynamic values such as its number of colored neighbors (inspired by our dynamic model), number of different colored neighbors (inspired by saturation coloring), distance to the closest colored node, distance to the closest uncolored node and a boolean variable that tags if the node itself is colored or not. What we hope from this 11-dimensional embedding is to explain the node's characteristics at any moment of the coloring process. For this approach, we should update dynamic values after every action.

- ***Graph Embedding:*** Our graph feature embedding contains its number of nodes, edges, the diameter of the graph, number of colored nodes, total number of neighbors of the colored nodes, total number of colored neighbors of the colored nodes, sum of closeness values of all nodes, sum of closeness values of the colored nodes, number of colored nodes whose degree are above average and number of uncolored nodes whose degree are above average. This embedding consists of mostly dynamic features. However, considering the computational requirements it needs, we plan to update it every T action (e.g. T = 4).

- ***Optimization:*** In order to tune the parameters of the Q-function (neural network), it is required to define a loss function. For our problem we plan to use:

  - ***Loss:*** Loss will be selected as the equation below, where ɣ is the discount factor between 0 and 1 to scale future utility, and θ is the parameters of the neural network.

$$Loss = (\gamma \cdot max_{v'} Q(S_{t+1}, v'; \theta) + r(S_t, v_t) - Q(S_t, v_t; \theta))^2$$

- ***Q-Learning:***

Algorithm 1 Q-learning for the Greedy Algorithm

Initialize experience replay memory $\mathcal{M}$ to capacity N
for episode $e = 1$ **to** L **do**
  Draw graph G from distribution $\mathbb{D}$
  Initialize the state to empty $S_1 = ()$
  for step $t = 1$ **to** T **do**
    $v_t = \begin{cases} \text{random node } v \in \overline{S}_t, & \text{w.p. } \epsilon \\ \text{argmax}_{v \in \overline{S}_t} \widehat{Q}(h(S_t), v; \Theta), & \text{otherwise} \end{cases}$
    Add v_t to partial solution: $S_{t+1} := (S_t, v_t)$
    if $t \geq n$ **then**
      Add tuple $(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t)$ to $\mathcal{M}$
      Sample random batch from $B \overset{iid.}{\sim} \mathcal{M}$
      Update Θ by SGD over (6) for B
    end if
  end for
end for
return Θ

Figure 11: Pseudo-code for Q-Learning Model [8]

Q-Learning is a reinforcement learning algorithm that tries to decide the best action in each state with respect to experience (previous states, actions, and their rewards). It is an off-policy algorithm since it does not require any additional rule to guide its decision-making process. Therefore, an epsilon-greedy algorithm, where epsilon decides the random action probability, is used to first explore and then exploit the structure. In this model every action can be seen as a function of the current state.

- ***Deep Q-Learning:***

Deep Q-Learning is the novel technique for approximating value function by a non-linear mapping where each value shows how good an action is. When the state space is finite, it is suitable to keep a state-action table for storing the values (standard Q-Learning). However, when the state space is continuous, or too large to keep in the memory, a function that approximates the table is preferred (Deep Q-Learning).

Therefore, in order to implement this algorithm, we need to create a neural network that approximates our Q-values and for that reason we choose to use Pytorch, an open-source deep learning library that has built-in functions which enable us to use GPU acceleration, making the implementation of the network easier and faster [12]. However, using Python has some drawbacks considering that our graph and node embeddings are computationally expensive operations. We overcome that problem by using Ctypes [13] binding that enables Python to execute C++ code in the background.

- ***Double Q-Learning:***

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(terminal, \cdot) = 0$
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probability:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
    $S \leftarrow S'$
  until S is terminal

Figure 11: Pseudo-Code for Double Q-Learning Model [5]

In some stochastic MDPs, the naive Q-Learning algorithm can perform poorly due to large overestimations. For this reason we decided to use Double Q-Learning which overcomes this problem with two estimators rather than one [14]. The main difference in Double Q-Learning is that it separates the selection and evaluation of this selection.

We begin the implementation of this structure by writing the embedding functions like graph embedding, node embedding, batch normalization, and update functions, using ctypes in order to create a communication environment between C++ and Python. After that point, all of the graph related functions were in C++, and the network was left to Python. We created a Double Q Network model utilizing Pytorch. While training the network, we either generated a graph sample or used small SuiteSparse data. In both cases, our improvement in the model was not sufficient. Then we conducted an experiment by feeding a single Deep Q Network with 3 graphs and trained on them with 120 epochs, then tested our model's performance with the graphs that are used in the training phase in order to understand whether our model is improving. However, even if we tried to overfit, our Deep Q Learning model could not achieve to perform well compared to saturation.

One of the possible reasons it does not improve was that when the network always chooses the best node greedily, in terms of reward it gives, what it actually does is choosing the easiest node to color, meaning that it selects the node that does not increase maximum color. One can observe that this approach contradicts with our previous results. All the previous approaches show that the harder node is, the earlier we select it. With the current reward mechanism, our network neglects the delayed rewards and current reward always dominates. Another reason might be that our embedding approach does not satisfy the network's demands. Rather than using handcrafted features, it might be the case that we should have applied a feature extraction method to grasp graph and node features. Detailed performance figures of the DQN Model can be seen in the Results & Discussion section.

Results & Discussion

As a result of the first part of the project, which includes heuristic analysis and weighted model construction, one can see that the weighted ordering model dominates all the other heuristics, that were used for static greedy coloring, as well as colorings based on uniform and random node selection. If the heuristics' results are investigated for small size graphs in distance-1 coloring, since the ratio is taken with respect to the optimal coloring results, the improvement is ~4% compared to the best heuristic. For large graphs, improvement is only ~3% and for distance-2 colorings for both sizes, only ~1%.

Color Dist.	Graph Size	Deg-1	Deg-2	Deg-3	Close. Cent.	Clust. Coeff.	Page Rank	Weighted	Uniform	Random
1	Small	1.186831	1.210086	1.189724	1.184408	1.259610	1.287079	1.14	1.194594	1.578385
	Large	1.0	1.019877	1.026063	1.035645	1.029322	1.076855	0.968	1.048112	1.227984
2	Small	1.016742	1.0	0.998299	0.987175	1.030498	1.061261	0.9805	1.020671	1.075174
	Large	0.999686	1.0	1.004391	0.997327	1.015180	1.056862	0.974	1.018301	1.041926

Figure 12: Results of Static Coloring Heuristics

If dynamic coloring algorithms are also involved, then better results can be obtained compared to weighted ordering algorithms. Especially with saturation coloring, ~8-10% improvement on distance-1 coloring and ~4% improvement in distance-2 coloring is observed.

Color Dist.	Graph Size	Weighted	Inci-dence	Satur-ation
1	Small	1.14	1.108566	1.066864
	Large	0.968	0.979025	0.876218
2	Small	0.9805	0.996965	0.946578
	Large	0.974	0.979260	0.944194

Figure 13: Results of Dynamic Coloring Heuristics and Weighted Ordering Algorithm

These results above indicate that saturation ordering gives significantly better results for all coloring types and all sizes. However, when the solutions are filtered by selecting data over a threshold set by the minimum number of colors, after some value, saturation also tends to converge along with other heuristics (dynamic stands for incidence degree):

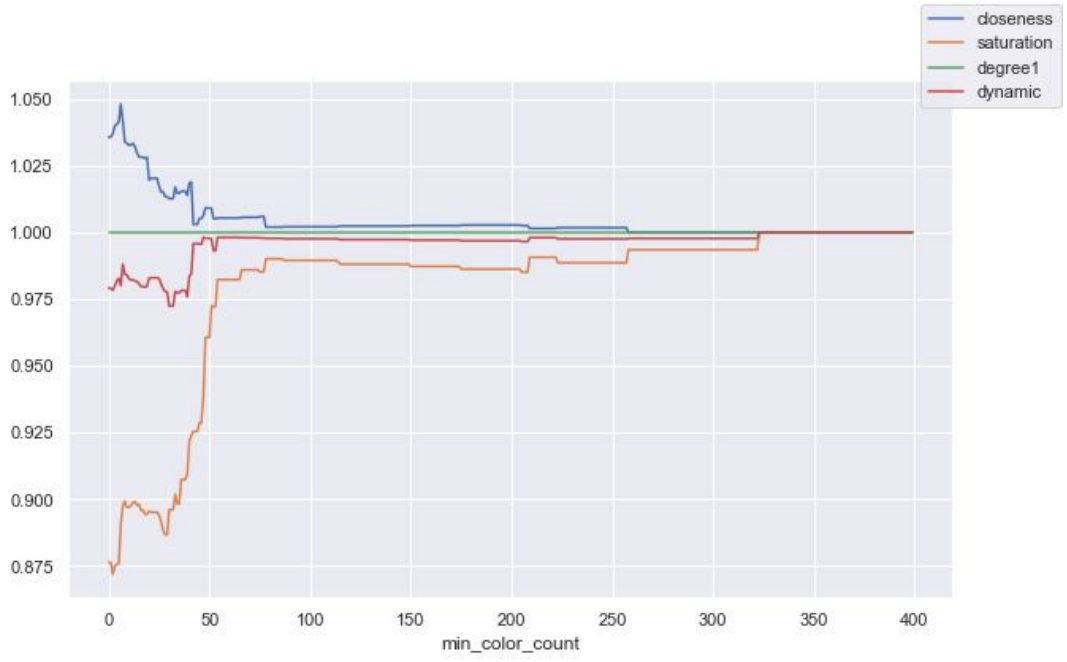

Figure 14: Distance-1 Coloring Results on Large Graphs with Thresholds

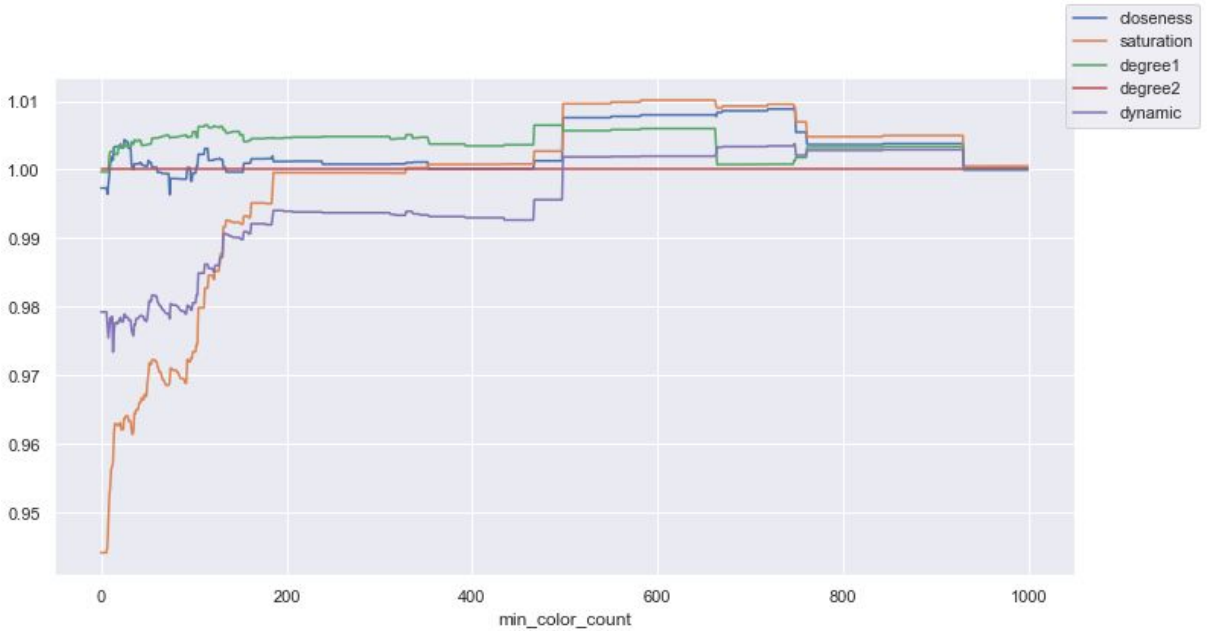

Figure 15: Distance-2 Coloring Results on Large Graphs with Thresholds

According to the first figure above, using saturation for distance-1 coloring is always more beneficial however, for distance-2, if graphs become denser or the minimum number of colors needed increases, it may be better to use degree-2 ordering for distance-2 coloring. Another drawback of saturation coloring is that it is not suitable for parallelization, because, for each step, values for vertices have to be updated. Therefore, if we seek for better time performance, then other heuristics become more favorable.

	degree1	degree2	degree3	closeness	clustering	pagerank	weighted	uniform	random	dynamic	saturation	double_qnet
count	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000
mean	1.186831	1.210086	1.189724	1.184408	1.259610	1.287079	1.194831	1.194594	1.578385	1.108566	1.066864	1.355061
std	0.222933	0.259865	0.219488	0.220671	0.276891	0.352060	0.249412	0.217872	0.470651	0.163620	0.112014	0.297713
min	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
25%	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.278750	1.000000	1.000000	1.142857
50%	1.133929	1.200000	1.166667	1.139610	1.250000	1.250000	1.166667	1.166667	1.476190	1.000000	1.000000	1.333333
75%	1.333333	1.333333	1.333333	1.333333	1.500000	1.333333	1.333333	1.333333	1.708333	1.200000	1.142857	1.500000
max	2.000000	3.333333	2.333333	2.000000	2.666667	3.000000	2.666667	2.000000	3.333333	1.750000	1.500000	3.000000

Figure 16: Double Q Learning Performance Table

Double Q Network is trained with graphs whose nodes between 100 and 500 and with average degree between 10 and N/4, where N is the number of nodes for a specific graph. We

create 10 graphs in each batch and after each coloring operation, the local network is updated with a learning rate of 0.005. In every 10 epochs we assign the local network's parameters to the target network's in order to gain a faster convergence over a total of 120 epochs. Moreover, to accelerate the training phase, we check Q-Values of top ten nodes with respect to their incidence degree.

As it can be seen from Figure 16, although the Double Q Learning model achieves to perform better than a random coloring, it fails to yield a better result than all other approaches. Since we underestimate the computational requirements of the deep learning model, the amount of time we spend on training may not be sufficient. Conducting further training is left to the future work in order to interpret the success of Double Q Network better.

Impact

By combining the exploitation of graph structures and reinforcement learning techniques, our solution to GCP provides a different viewpoint in the literature. Therefore, our approach forms a new baseline for developing more efficient models in the future. Moreover, this project is an example that shows how versatile RL can be when it comes to solving different kinds of complex problems.

Our complete dataset and source codes are available in our GitHub repository. By making this project open source, we allow other people to reproduce and improve our results.

Ethical Issues

The project does not present any danger for humans, animals, or the environment, however, it is equally important that the project should be conducted within the boundaries of academic integrity.

The complexity of the project requires extensive research. While it is being carried out, it is our utmost concern, to be honest with our findings. There are related papers written on this topic and more may appear during development. It is crucial to be aware of them and give credit where necessary.

Moreover, the experiments we performed have to be available for reproducibility. Through keeping all of our codes open-source and providing the datasets we have used for evaluating this algorithm, we aim to create an environment, where each step we follow can be observable and repeatable by others.

Making a significant contribution can only be achieved by being objective with our design and results. Being biased and altering results for a favor is substantial misconduct. Responsibility and respectfulness of the contributors are crucial to overcome hardships throughout the project.

Project Management

When starting this project, it was decided that the performance of the systems we were using along with the complexity of the problem would create trouble. Therefore we had to choose C++ as our language for graph related calculations and Python for our RL model, since C++ is more optimized and Python has more libraries in terms of Machine Learning. However we have underestimated the work that needs to be done for building a communication framework between 2 languages. Moreover, the time required to understand deep Q-learning structures on PyTorch also took more time than expected and this put us back behind the schedule. Hence, there was no time left for visualizing the process, although it was stated in the initial report.

Conclusion & Future Work

In the first part of our project, different heuristics were analyzed and orderings retrieved from these heuristics were used in the greedy graph coloring process. For both versions of graph coloring (distance-1 and distance-2 respectively) saturation based coloring is proven to be best among all the heuristics chosen in this project. Moreover, saturation based coloring also dominated our weighted ordering model.

In the second part, an RL model is developed based on deep Q-learning. For this purpose, embeddings are prepared for both graphs and individual nodes. In this way, a new solution in literature for GCP is proposed, which consists of exploiting graph structures. However, the results we collected do not conclude any improvement in solving GCP.

For the next steps, improving the existing RL model is aimed. If promising results will be reached, then further improvements may be made and visualization may also be added for better representation of the entire coloring process.

References

[1] Hasenplaugh William, Kaler Tim, Schardl Tao B., Leiserson Charles E.. Ordering Heuristics for Parallel Graph Coloring, 2014.

[2] Lloyd E.L., Ramanathan S.. On the Complexity of Distance-2 Coloring, 1992.

[3] Bellman, R. (1957). "A Markovian Decision Process". Journal of Mathematics and Mechanics.

[4] Dailey, D. P. (1980), "Uniqueness of colorability and colorability of planar 4-regular graphs are NP-complete," Discrete Mathematics, 30 (3): 289–293, doi:10.1016/0012-365X(80)90236-8

[5] Sutton, R.S. and Barto, A.G. Reinforcement Learning: An Introduction. MIT Press, 1998.

[6] Aşık A., Demir I. B., Demirel, B., Kaya K., Topal B. B., Vertex Ordering Algorithms for Graph Coloring Problem. 2020. Available at: http://www.pdf-express.org/Conf/49456XP/versions/1053055/PID6391619.pdf

[7] Bello, Irwan, Pham Hieu, Le, Quoc V, Norouzi, Mohammad, and Bengio, Samy. Neural Combinatorial Optimization with Reinforcement Learning. arXiv preprint arXiv:1611.09940, 2016

[8] Dai Hanjun, Khalil, Elias B. Zhang, Yuyu, Dilkina, Bistra, Song, Le. Learning Combinatorial Optimization Algorithms over Graphs. arXiv:1704.01665, 2018.

[9] Jiayi Huang, Mostofa Patwary Gregory Diamos. Coloring Big Graphs with AlphaGoZero. arXiv:1704.01665, 2018.

[10] Suite-Sparse 'A Suite of Sparse Matrix Software' [online]. Available at: http://faculty.cse.tamu.edu/davis/suitesparse.html

[11] IBM. CPLEX User's Manual, Version 12.6.1, 2014.

[12] Pytorch 'A Machine Learning Framework' [online]. Available at: https://pytorch.org/

[13] CTypes ‘A Foreign Function Library for Python’ [online]. Available at: https://docs.python.org/3/library/ctypes.html

[14] Hasselt H. V. (2015) ‘Double Q-Learning’. Available at: https://arxiv.org/pdf/1509.06461.pdf

[15] Watkins, C.J.C.H. (1989). Learning from delayed rewards. PhD Thesis, University of Cambridge, England.