Using an unsupervised GNN for the k-coloring problem

Student:
Dirk Jan de Lint (582523)

Supervisor:
dr. O. Kuryatnikova
Second assessor:
dr. P.C. Bouman

April 7, 2023

Abstract

In this thesis, I have designed an unsupervised GNN for the k-coloring problem where the message-passing part of the design is inspired by [VCC$^+$17] and the loss function is based on [SBZK22]. Instead of training only on the usual neural network parameters, I have also allowed the algorithm to train on the input features, which has to my knowledge not yet been done before. A new set of trainable parameters needs to be trained for each graph. The best performing method of the algorithm is able to find optimal colorings for the *mycielski* graphs and is able to find some optimal colorings for the *non-mycielski* graphs but can not match GNN models recently proposed in the literature for the k-coloring problem. Next, training on the features improves the performance of the algorithm for most graphs. Lastly, I have shown that the algorithm can be initialized using either a heuristic or a relaxation and that the best performing method of the algorithm is able to improve on almost all the starting points provided by the heurisics and on all starting points provided by the relaxation.

Contents

1 Introduction

With the continuing improvements in computational power, machine learning has become an often-used tool to solve various problems in a range of different fields such as image recognition and natural language processing. In recent years machine learning has also been applied to combinatorial optimization problems with varying success. Machine learning can assist in two ways, it can either replace heavy computation by using some learned fast approximations, or it can explore the space of decisions and try to learn out of this experience the best-behaving policies [BLP21]. In this thesis, this first way is used to help solve the decision version of the graph coloring problem (GCP). Namely, I propose and test an unsupervised GNN that works as a heuristic and tries to find a feasible k-coloring of the graph.

The goal of the GCP is to color a graph, using the minimum number of colors, in such a way that no two adjacent vertices are colored with the same color. The k-coloring problem, where the question is whether a graph allows a feasible coloring using k colors, is the decision version of the GCP and can, by decreasing k until the decision problem returns a no-decision, be used to minimize the number of colors needed to color a graph. The GCP originates from an old map drawing problem where the question was how many colors are needed to assign all counties of England a color such that no two neighboring counties are assigned the same color. Nowadays the GCP remains very interesting as its applications are widespread, many problems such as scheduling problems [MHH08], register allocation problems [CAC$^+$81], and frequency assignment problems [PL96] can be formulated as a GCP. However, even with state-of-the-art methods finding optimal solutions for these problems remains very difficult, not the least because the GCP is an NP-hard problem [Kar72]. Given that the GCP is NP-hard, designing an algorithm that provides optimal solutions in a reasonable time is improbable. A fast heuristic, however, that nears the optimal solutions is more likely to be found. If this heuristic also improves on the results of the state-of-the-art methods, problems in many fields could be solved more efficiently and therefore save a lot of time and costs.

This thesis suggests the above described heuristic. I will use an unsupervised graph neural network (GNN), where the architecture of the GNN is based on a combination of two papers. The unsupervised loss function is based on [SBZK22] and the rest of the design is based on the architecture of [VCC$^+$17]. Instead of using existing features for the initialization as is done in [VCC$^+$17], the features are initialized randomly as is done in [SBZK22], [LPAL19], and[LLM$^+$22]. However, where these papers do not allow their GNNs to train on these randomly initialized features and instead train on the neural network parameters, I will train my GNN both on the randomly initialized features and on the neural network parameters. I do not allow the algorithm to train on the features and on the parameters at the same time but instead switch between training on the features and training on the parameters.

I use an unsupervised approach for two reasons. First, an unsupervised approach does not need labeled training graphs. Second, the loss function of an unsupervised approach is more intuitive, we can simply derive the loss by summing the (partly) conflicting colors for adjacent nodes. In a supervised approach we are essentially training the algorithm to only find a specific solution thereby disregarding all the other solutions, which are either permutations of this solution or are entirely different solutions. I suggest to not only train on the neural network parameters but also on the features. Normally the starting point, i.e., the input of a GNN, is fixed. For unsupervised NNs the input features are usually chosen randomly, [LLM$^+$22], [SBZK22]. However, a better choice of the starting point could improve the performance of a NN, as [SBK21] remarks. That is why in this thesis I have decided to change the starting point in order to find potentially better solutions than a random starting point would provide. The training on the features is the novice part of this design and has to my knowledge not yet been tried for GNNs in application to the GCP.

The goal of this thesis is to examine how well such an extended unsupervised approach performs. In order to do this I will compare the outcomes of my proposed algorithm against several other algorithms. Another aspect of this thesis is to explore if the machine learning approach can be combined with already existing methods, I will therefore combine the algorithm with various heuristics, such as the Recursive Largest First algorithm [Lei79], the TABUCOL algorithm [HdW87], and with a linear programming relaxation [MDZ08], to see if the algorithm can improve upon the solution provided by these heuristics or relaxation. Next, I also try to turn the final colorings of the algorithm that are infeasible into feasible colorings by either using the TABUCOL algorithm, or by a self-designed color correction heuristic based on the RLF algorithm [Lei79]. To sum up, this thesis has three main research goals.

1. To construct an unsupervised NN that would work as a heuristic that repeatedly solves the decision version of the GCP
2. To improve the performance of the obtained NN by optimizing over the features vector
3. To combine the obtained NN with heuristics for the initialization and for turning infeasible solution into feasible solutions.

The thesis is structured as follows. In section 2, I discuss the relevant literature relating to the GCP and GNNs and provide necessary background information on (G)NNs. In section 3, I discuss the data used. Next, in section 4, the methodology is explained. Thereafter, in section 5, I present the results. And finally, section 6, concludes the findings and gives an outlook for possible future research.

2 Background & Literature Review

The graph coloring problem is a well-known problem in literature, and many different approaches have been tried to solve this problem. However, the use of machine learning for the GCP has not yet been deployed that often. In this section, both the more classical approaches and the more modern approaches using machine learning will be discussed. First, in subsection 2.1, I will give a general description and formulation of the problem. Next, in subsection 2.2, I will relate some heuristic approaches to this study. This will be followed by an introduction into (graph) neural networks and their specific architectures in subsection 2.3. And lastly, I will discuss a few machine learning approaches for the GCP in subsection 2.4.

2.1 Formulations

This thesis aims to solve the decision version of the GCP using a machine learning approach. The goal of the GCP is to find a coloring using the minimum number of colors. That is, to minimize the number of colors used to assign all vertices of a connected undirected graph a color such that no two adjacent vertices are assigned the same color. The smallest integer for which a feasible solution can be found is called the chromatic number (χ). The decision version, the k-coloring problem, of the GCP is: whether or not a graph G admits a feasible assignment of k colors. Here the goal is to check whether or not a graph can be colored using k-colors. Using the k-coloring problem the minimization version of the GCP can be solved. Starting from an upper bound k, obtained via a heuristic, or in the worst case by $k = |V|$, a series of k-coloring problems can be solved until the k-coloring problem returns a "no" result. To guarantee, however, that the chromatic number is found, the "no" result of the k-coloring problem either needs to be solved exact, or the k for the "yes" result should match a lower bound on the chromatic number. Since in this thesis a heuristic is used, unless some known lower bounds are met, it can not be guaranteed that the lowest value k, for which the algorithm is able to find a feasible solution, is indeed the chromatic number. The minimization version of the GCP can be formulated in many different ways. Next, I present the formulation obtained from [HLS09]. Let $G = (V, E)$ be a graph, and assume that the graph can be colored with at least n colors. Two vertices $v, w \in V$ are said to be adjacent if they are such that $(v, w) \in E$. Next, let $y_h = 1$ if color $h = 1, \ldots, n$ is used and let $x_{ih} = 1$ if vertex i receives color h. Then, the formulation is as follows:

$$\text{Min.} \sum_{h=1}^{n} y_h \tag{1}$$

$$\text{s.t.} \sum_{h=1}^{n} x_{ih} = 1 \qquad i = 1, \ldots, |V| \tag{2}$$

$$x_{ih} + x_{jh} \leq y_h \qquad (i,j) \in E, \qquad h = 1 \ldots, n \tag{3}$$

$$x_{ih} \in \{0,1\}\; i = 1, \ldots, |V|, \qquad h = 1 \ldots, n \tag{4}$$

$$y_h \in \{0,1\} \qquad h = 1 \ldots, n \tag{5}$$

As [MT10] notes, however, this formulation has two major drawbacks which render it practically useless. First, due to the unknown chromatic number χ beforehand, n has to equal an upper bound which will be, in almost all cases, higher than k. For a solution, then, that uses k colors there are $\binom{n}{k}$ equivalent solutions, since the model does not distinguish between colors. All these equivalent solutions can again be permutated in $k!$ ways, resulting in $\binom{n}{k}k!$ equivalent solutions for a solution of value k. Second, the linear programming relaxation of the model, obtained by (1)-(5) by replacing (4) and (5), respectively, with

$$x_{ih} \geq 0, \; i \in V, \qquad h \in 1, ..., n \tag{6}$$

$$y_h \geq 0, \qquad h \in 1, ..., n \tag{7}$$

is extremely weak. A feasible solution of value 2, valid for any graph, is simply $x_{i1} = x_{i2} = 1/2, i \in V; y_1 = y_2 = 1; x_{ih} = 0; y_h = 0, i \in V, h > 2$. To strengthen this formulation, a few extra constraints can be added as done by [MDZ08]. To avoid the symmetry of the classical model and eliminate equivalent solutions, they add the following two constraints to the equations (1)-(5).

$$x_{ih} = 0 \quad \forall i \quad \forall h \geq i + 1 \tag{8}$$

$$x_{ih} \leq \sum_{k=h-1}^{i-1} x_{kh-1} \qquad \forall i \in V \backslash \{1\} \quad \forall 2 \leq h \leq i - 1 \tag{9}$$

Here, none of the symmetrical k-colorings using colors with labels greater than k are considered. Furthermore, all permutations of sets of the same cardinality are eliminated. As the linear programming relaxation provides a non-integer solution of the GCP, it might be an interesting starting point for the GCP. Therefore, I will use the linear programming relaxation consisting of equations (1-3) and (6-9) as a starting point of the algorithm to see if this can help the performance of the algorithm. How the algorithm is initialized using this starting point is described in 4.3.1.

2.2 Heuristic approaches

Since finding the chromatic number is NP-Hard [Kar72], exactly solving the minimization problem of the GCP, equations (1-5), is practically infeasible for most instances. Therefore, through the years, many heuristics approaches have been developed to (successfully) color graphs. I have chosen to highlight three of these heuristics, DSatur [Bre79], Recursive Largest First (RLF) [Lei79], and TABUCUOL [HdW87]. DSatur, because of its simplicity and its efficiency $\mathcal{O}(n^2)$, which makes it useful as an upper bound on the chromatic number. RLF, because of the multiple ways it can be combined with the proposed algorithm of this thesis. And TABUCOL, because, first, it is one of the best performing heuristics and thus is useful as a comparison to the proposed algorithm of this thesis. And second, because it is an improvement heuristic and therefore can, without many adaptations, be used to try to turn infeasible colorings of the proposed algorithm into feasible colorings. In the paragraphs below, I will first explain the general workings of the algorithm and thereafter discuss how and where these heuristics are used in this thesis.

DSatur [Bre79] The DSatur-algoritm uses the saturation degree of the vertices, i.e. the number of different colors a vertex is adjacent to, to determine which vertex to color next. This algorithm has complexity $\mathcal{O}(n^2)$ and is proven optimal for bipartite graphs [Bre79]. The algorithm consists of five steps:

1. Arrange the vertices by decreasing order of degrees
2. Color a vertex of maximal degree with color 1
3. Choose a vertex with a maximal saturation, if there is a tie, choose any such vertex of maximal degree in the uncolored subgraph
4. Color the chosen vertex with the least possible (lowest numbered) color
5. If all the vertices are colored, stop. Otherwise, return to 3.

Since the DSatur-algorithm finds feasible solutions for every graph and is very efficient in doing so, I have chosen to use this algorithm as an upperbound for the GCP and use the number of colors the DSatur-algorithm needs as the starting value of k for the algorithm used in this thesis. The pseudcode of the Dsatur algorithm can be found in the appendix B pseudocode 4.

RLF [**Lei79**] Another influential algorithm is the Recursive Largest First (RLF) algorithm [Lei79]. This algorithm also colors the graph vertex for vertex but selects the vertex which needs to be colored next in a different way. The RLF algorithm aims to construct independent sets. The underlying concept behind this is that every vertex in an independent set can be assigned the same color, since a vertex in an independent set is not adjacent to any of the other vertices in the independent set. To color the graph with the least number of colors, the RLF algorithm tries to subdivide all the vertices of the graph in as few independent sets as possible, because the number of independent sets equals the number of colors needed to color the graph. To construct these independent sets, such that vertices are subdivided efficiently, the following should be done according to RLF algorithm:

1. Select the vertex of maximal degree in the graph and create a set S_1.

2. Next from all vertices not adjacent to S_1, pick the vertex such that the degree of the vertices in the set of uncolored nodes not adjacent to any colored node is maximal in the set of uncolored vertices adjacent to S_1. Ties are, if possible, broken by choosing the node that has minimal degree in the set of uncolored nodes not adjacent to the independent set.

3. If the set of uncolored nodes not adjacent to S_1 is empty, color all the vertices in S_1 with the same color and exclude these vertices S_1 from the graph, and start a new set S_2 and continue the recursion. Do this until all vertices are assigned a color

I will exploit the use of independent sets by the RLF-algorithm when turning an infeasible coloring into a feasible coloring and color the subgraph consisting of the improperly colored nodes and their edges, using an adapted version of the RLF-algorithm. In subsection 4.4.1 this is explained in more detail. I will also use the RLF-algorithm as a starting point of the algorithm proposed in this thesis. Since the colorings of the RLF-algorithm are always feasible, I will remove one color from the solution and provide this infeasible solution as a starting point of the proposed algorithm and aim to color the graph in question with one color less than the number of colors the RLF-algorithm used, see subsection 4.3 for more details.

TABUCOL [**HdW87**] Both of the previous two heuristics mentioned are constructive heuristics. TABUCOL [HdW87] on the other hand, uses an improvement heuristic in the form of tabu search. In contrast to the previous two heuristics, in this approach, a target chromatic number needs to be provided, for which the algorithm tries to find a feasible coloring. The TABUCOL algorithm, therefore, does not solve the minimization version of the GCP but rather answers the k-coloring problem, using a heuristic approach, just as is tried in this thesis. Similar to the RLF approach, the TABUCOL algorithm aims to generate independent sets of vertices. The TABUCOl algorithm is initialized by dividing the vertices over k sets, next the objective function is defined as the number of edges for which both endpoints are in the same set. If the objective value becomes zero, a feasible k-coloring is found. Now, until the objective reaches zero or the maximum number of iterations is met, the following steps are performed:

1. Generate a fixed number of neighbors as follows: select a random vertex from the set of edges for which both endpoints are in the same set, and move this vertex to a different set

2. Evaluate the objective functions for the neighbors and pick the best one and move to this neighbor. After the move is made, this move back becomes tabu for some iterations.

Since the TABUCOL algorithm is an improvement heuristic and is also used to solve the k-coloring problem, it can quite easily be used to improve the resulting infeasible colorings of my proposed algorithm, by initializing the TABUCOL algorithm using the infeasible colorings (subsection 4.4.2). Similarly, the infeasible colorings of the TABUCOL algorithm can also be used to initialize my proposed algorithm (subsection 4.3.). It will be quite interesting to see whether or not both methods, TABUCOL and my proposed algorithm, are able to improve the infeasible coloring of the other method, since this would give an indication of which method is stronger.

2.3 Introduction into (graph) neural networks

In this section, I will first introduce the basics of a Neural Network. After that, I will introduce a subgroup of NN architectures, Graph Neural Networks, and explain the architecture behind a GNN in more detail.

2.3.1 Neural Network

I retrieved the information of this section from the lecture slides of the course Machine Learning and I used the paper [GK11].

Let x describe the input features vector and let y describe the output vector. An artificial neural network (NN) is a function from x to y, that is inspired by the workings of a biological neural network. A biological neural network roughly works in the following way:

- The brain gets some stimulus from outside the brain
- The receptors translate this stimulus to various signals
- The neural net processes these signals
- The effectors translate these signals into a response

The neural net thus transforms some stimulus into a response. The same is done for an artificial neural network. An input, which can be various kinds of data, is transformed with a certain goal, the response, in mind. This goal can be classification, for example, what is pictured on an image, or the goal can be to predict, for example, who will win the premier league? Or the goal can be something else. The input x and the output y, thus can be of an entirely different class, the input can be a matrix, where every index corresponds to the color of a pixel, and the output then can be a description of the image. How does an artificial neural network do this? This is done by using processing elements. A processing element similar to a neuron in a biological neural network, can accumulate signals, aggregate these signals, transform this aggregate, and sent out the transformed aggregate. Multiple processing elements together form a layer. Processing elements of different layers are connected to each other and form a directed graph, where the nodes are the processing elements and the edges are the connections between the processing elements. Multiple layers together form a neural network. Usually signals from a previous layer first undergo an activation function, depending on the design this can be a function where a threshold value needs to be met in order for the signal to be processed, or the activation function only dampens/strengthens the signal. An activation function can be very useful to dampen noise in the network and ensure that only strong signals are processed. The processing elements make use of a few parameters between accumulating the information of the previous layer and sending the transformed signals to the next layer. These parameters are of key importance for the network as these can be adapted and therefore allow the neural network to perform better. These parameters usually consist of some weight parameters, which are multiplied with the incoming signals, and some bias parameters, which are added to the incoming signals. In Equation (10) a transformation is shown for one of the processing elements, j, of layer i. Here, k represents the connections to the previous layer, the $\beta's$ are the weight parameters, h represents the activation function, and $a_{j,k}(x_k^{i-1})$ represent the value of the activation.

$$x_j^i = \sum_k^K \beta_{j,k}^{i-1} + \beta_{j,k}^{i-1} h(a_{j,k}(x_k^{i-1})) \tag{10}$$

Usually, a neural network consists of three types of layers. An input layer, hidden layers, and an output layer. The input layer is just like the receptors in the biological neural network. It translates the data such that it can be processed by the neural network. The hidden layer processes and transforms the data using functions similar to Equation (10). The output layer is just like the effectors in the biological brain and translates the output of the hidden layers into the desired form of the output, either being a prediction, classification, or something else. An overview of a neural network is presented in Figure 1. In this figure, all the nodes of layer i are connected to all the nodes of layer $i+1$ this does, however, not need to be the case, depending on the goal of the problem it might be advantageous to only connect certain processing elements to each other.

A key component of a NN is the training. The various parameter of the network, such as the $\beta's$ described above, need to be trained, in other words, estimated, in order for the NN to function as intended. In order to update the parameters, first the loss, in other words, the objective function to minimize or maximize during the training, of the current parameters needs to be determined. There are three different ways that are normally used to determine the loss of a NN: supervised learning, unsupervised learning, and reinforcement learning. For supervised learning, the data used for training consists of both the input, x, and the output, y. The x is used as the input of the algorithm and the output of the algorithm $\hat{y}$ is used to calculate the loss with respect to y. Thus, in supervised learning, the loss function indicates how different the output of the algorithm is from the intended output. For unsupervised learning, the data only consists of the input, x. The x is again used as the input of the algorithm but the loss is calculated by evaluating the fitness of the output using a target expression tailored to the problem. For reinforcement learning, the goal of the network is to accurately predict the action space. The input is again x but the loss is calculated by comparing the prediction with the feedback of the environment of the reinforcement learning.

After calculating the loss, the NN parameters need to be updated. Usually, backpropagation is used for this. Here using partial derivation, from the loss to the parameters of the network, the specific loss for each of the parameters of the network can be determined. Next, the parameters are updated using some non-linear optimization algorithm.

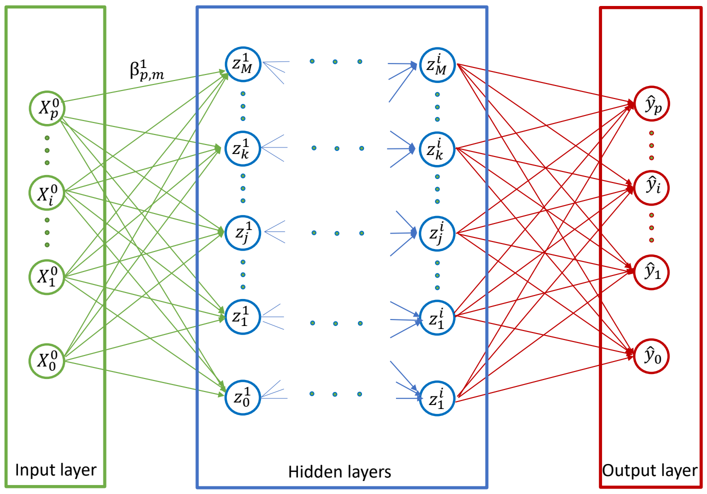


Figure 1: Overview neural network

2.3.2 GNN

When designing a NN many different architectures can be used. One of the options is Graph Neural Network (GNN). A problem that is solved using a GNN is beforehand already defined on a graph. The GNN architecture then exploits the graph structures in its design thereby distinguishing it from other architectures. In the subgroup of GNN's, there exist many different kinds of designs all tailored to different problems. In this part, I will give some key considerations to keep in mind when designing the three different parts of the NN (the input layer, the hidden layers, and the output layer).

Input layer The input layer is an important part of the algorithm as it determines the way in which the data can be used in the problem. In order to choose an effective way to input the data in the algorithm, it is important to get a good understanding of the characteristics of the problem. One of the first things to look at is, whether or not a graph has an implicit or explicit structure. If the problem is already defined using a graph, such as the subject of this thesis, the structure is called explicit on the other hand if the graph has to be constructed out of the problem the structure is called implicit. Next, the goal of the problem should be taken into consideration. Is the problem on vertex, edge, or graph level? And given the level of the problem, should the GNN classify, predict, regress, or do something else? In this thesis the goal is to assign one of the k colors to each vertex therefore the goal is classification and the problem is on node level. Other characteristics that need to be defined are whether the graph is directed or undirected, homogeneous or heterogeneous, whether the graph is static or dynamic, meaning that the input features of the graph vary with time (dynamic) or remain the same over time (static), also the scale of the problem should be considered. The GCP respectively is undirected, homogeneous, static, and ranges from small to large problems.

Hidden layers The hidden layers transform the inputted data such that the output layer can generate a solution for the problem. This is done by using a range of different mechanisms. In [ZCH+18] the authors differentiate three types of components: a propagation module, a sampling module, and a pooling module, of which one or more a generally used to transform the data in a GNN. They work as follows:

- **Propagation module**. The propagation module is used to exchange information between vertices of the graph in order to capture the information of the features of the vertices as well as the topological information.
- **Sampling module**. When the number of vertices and edges is very large, the cost of propagating the message for all the vertices in a neighborhood can become too expensive. To shrink these costs, sampling can be used. Depending on the method either the nodes, the layer, or whole parts of the graph can be sampled.
- **Pooling module**. The goal of a pooling module is to combine the features of multiple nodes to get more general features of the graph. This could be some hierarchical information but also topological information. For the k-coloring problem, for example, it could be very useful to be able to recognize some structures for which the number of colors is known, such as triangles.

Output layer and loss function Just as for the input layer the output of the algorithm is problem specific and strongly depends on the level and the goal of the problem. In the case of the GCP, depending on what precisely the goal is, already many different outputs can be used. One could try to predict χ and let the output be the prediction for the chromatic number. Or one could try to predict whether or not the graph is k-colorable, by letting the output equal a boolean representing whether or not the graph is k-colorable. One could also try to find a feasible k-coloring, as is done in this thesis, then the output should be a coloring of the graph. Another factor to account for in the output of the algorithm is the way the algorithm is trained. If the goal is to find a feasible k-coloring for example, a supervised training approach should be able to deal with the different permutations of the intended solution.

2.4 Some GNN desings

In this subsection I will relate GNN desings to the GCP. I will do this by first introducing three papers that are strongly related to the goal of this thesis, to use an unsupervised GNN for the GCP, and therefore are very relevant. I will discuss the designs of these papers separately in subsection 2.4.1, focussing on their initialization, the computational modules used, and the way in which they determine their loss. Thereafter, in subsection 2.4.2, I will compare the different components of the architectures of the papers in 2.4.1 in separate paragraphs. In these separate paragraphs, I will also mention papers where only some parts of the design are relevant to this thesis.

2.4.1 GNNs for GCP

Rethinking Graph Neural Networks for the Graph Coloring Problem [LLM+22] In this paper, the authors first establish four rules that make a GNN, denoted by A, powerful in the coloring problem.

1. The input graph contains no equivalent node pairs. Here an equivalent node pair is a pair of two nodes for which their neighborhoods are isomorphic and the features of the nodes in both neighborhoods are the same, see Figure 2. If the graph contains equivalent node pairs they advise to assign the nodes different node attributes. The authors establish this rule because they argue that all aggregation combination GNN, AC-GNNs, cannot discriminate any equivalent node pair, which in turn means that the output for such nodes will be very similar and therefore will cause problems when coloring these nodes.
2. A does not integrate the aggregation and combination function. An aggregation function is a function that aggregates features from the neighborhood and a combination function is a function that combines its own features with the features of the neighborhood. If the aggregation function aggregates features from the neighborhood and the node itself simultaneously, they call an aggregation and combination function integrated. The authors establish this rule because they say that if two nodes u and v share the same neighborhood except for each other, that is $N(u)\backslash\{v\} = N(v)\backslash\{u\}$, then these two nodes cannot be discriminated by an integrated AC-GNNs.

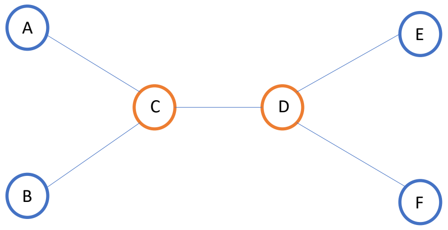


Figure 2: Equivalent node pair. In this figure nodes C and D are an equivalent node pair.

3. A should be as deep as possible, that is, A should have as many hidden layers as possible, because as they argue, due to the local nature of an aggregation function, an AC-GNN of L layers cannot detect the structure of nodes more than L steps away. Given that the depth of a graph, the minimum number of steps from the two nodes farthest apart from each other, differs from graph to graph, the depth of the A should be as deep as possible in order to ensure that for every graph the structures of as many as possible steps away can be detected.

4. Layers should be injective, meaning that a layer maps distinct elements of the input to distinct elements of the output of a layer. The authors establish this rule because if layer L^{i-1} discriminates $\{u, v\}$ layer L^{i} should also discriminate $\{u, v\}$.

Next, the authors try to design a GNN that can solve the k-coloring problem. They do this based on the four rules described above. For the initialization of the node attributes, they use a centered probability distribution of k colors, which is initialized randomly in order to eliminate the equivalent node pairs (rule 1). For the hidden layers, they first derive the message of a node by summing the node embeddings of the previous layer for the neighborhood of the node. Next, they combine the message of this node, multiplied with a learnable parameter, with the previous embedding of this node, multiplied with a different learnable parameter, to find the new node embedding of this node. In the output layer, they use an argmax function to round the node embedding of a node and obtain the final node embedding. The learnable parameters of this design are the learnable parameters used in the combination function. To train these learnable parameters they minimize an unsupervised loss function where the loss is derived as follows. For each pair of vertices that are adjacent, they evaluate the euclidean distance of two vectors and subtract this from a predetermined margin. They motivate their loss function by stating that the final node embeddings of adjacent nodes should be as different as possible. In the paper, the authors do not specify if they train new trainable parameters for each graph or that they train their trainable parameters using some graphs and then use these trained parameters for the graphs they use when testing the performance of their algorithm. However, when looking at the time spent on each graph during their testing, one graph, 2_Insertions_4 to be specific, takes way more time than the other graphs of the same size, which would be illogical if they only need to do one forward propagation. This indicates to me that they train new parameters for each graph.

Graph coloring meets deep learning: Effective Graph Neural Networks [**LPAL19**] In this paper, the authors do not directly build a feasible solution for the GCP but let the algorithm answer the question of whether a graph is k-colorable? The algorithm uses a color embedding as well as a node embedding as inputs. The color embedding is a vector of length of the number of colors, and the vertex embedding is a vector of length of the number of nodes. Both are initialized randomly, the color embedding using the uniform distribution between 0 and 1, and the vertex embedding using the normal distribution also between

0 and 1. The algorithm also uses a vertex to vertex adjacency matrix, and a vertex to color adjacency matrix where every color is connected to every vertex. Next, for the hidden layers the algorithm uses a message passing scheme, where the adjacent vertices and the colors communicate and update their embeddings for a number of layers using four learnable message functions:

- color message function, which translates colors embeddings into messages that are intelligible to a vertex update function;
- vertex message function, which translates vertex embedding into messages;
- vertex updating function, responsible for updating vertices;
- color updating function, responsible for updating colors.

The final embeddings are then used to compute a probability corresponding to the model's prediction of the answer to the decision problem: does graph G accept a k-coloring? The model is trained using a supervised loss function, where the output of the model is compared to the label of the graph. They train the algorithm using self-made graphs. The authors generate these graphs in pairs where one graph is just k-colorable and the other, by adding another edge, is $k+1$-colorable. The authors chose to generate these graphs which are borderline k-colorable because this should result in graphs that are very hard to classify.

Graph coloring using physics inspired graphs [SBZK22] The authors of this paper try to find a k-coloring of a graph. The node embeddings of the first layer are initialized randomly. Then the model follows (multiple) layers of a recursive aggregation scheme, where each node collects information of its neighbors in order to compute its new node embeddings. In order to output a coloring they use a softmax function. The softmax function transforms the real values, in this case the node embeddings, to an output where all the values sum to 1. The main contribution of this paper is its loss function. They take the dot product of every node pair and multiply each pair with a penalty. This allows them to train the network unsupervised. The network parameters are trained for every instance, and when the training is finished, they use a projection heuristic to map the soft assignments of colors to hard assignments.

2.4.2 Componentwise discussion of GNN literature

Initialization In the three methods above, the initial node embeddings are all initialized randomly. However, in most GNNs the model uses the node features for the initial embedding. These node features are often derived from real-world data and therefore can provide various kinds of information about both the node itself and its relation to other nodes, which in turn can both speed up and improve the accuracy of the training of the neural network. There are various ways in which node features are used and initialized. In [DHD+19] the authors point out that one of the advantages of using a GNN is the ability to incorporate node features into its learning process and show that a GNN works well when there is a strong connection between node features and the label the node needs to be classified as. They prove this by showing that there is a performance decrease when they shuffle the node features whilst keeping the labels intact. Furthermore, they show that for graphs where there are no node features available, one can artificially create them and that these artificially created node features can perform rather well. Since the initialization for the three approaches above is random, the node features most likely do not have a strong connection to the node labels. Adapting these node features such that the node features have a stronger connection to the node labels might improve the power of the GNNs from the three papers mentioned above. Therefore in this thesis, I will try to increase the connection between the node features and the final coloring by allowing the network to treat the initial node embeddings as trainable parameters.

Computation modules There is a vast number of different options for this part of the architecture of the neural network. When looking at the three papers discussed above, they all use some aggregation and combination function, which for the GCP seems logical. When we look at whether or not [SBZK22] and [LPAL19] adhere to the four rules for a powerful GNN for the GCP, we can observe that [LPAL19] adheres to all rules and that [SBZK22] adhere to three of the four rules. Both papers adhere to the first rule as they both initialize their node embeddings randomly, therefore, ensuring that no equivalent node pairs exist. Both papers also adhere to the second rule, as neither integrates their aggregation and combination functions. However, looking at the third rule, the GNN used in [SBZK22] only has one layer, thereby not being as deep as possible. The number of layers is a hyperparameter of their GNN and it can therefore be concluded that, at least in their case, one layer performed better than the other values they have tried. In [LPAL19], the authors allow for up to 32 layers and thereby adhere to rule number 3. Both the papers also use layers that are injective therefore adhering to rule number 4. From this, we can conclude that both papers, [SBZK22] and [LPAL19], seem to be powerful for the GCP.

In [SBK21], the prequel of [SBZK22], the authors mention in their recommendations for further research that trying out different architectures such as [VCC+17] might be useful. This architecture is not specifically designed for the GCP but can be easily adapted for this use. The authors of [VCC+17] initialize their algorithm using real-world features. Next, they use a learnable weight matrix to transform the size of the node embeddings from the real-world features size to the desired size for the problem at hand. This node embedding is then used to derive the attention coefficients. This attention coefficient is a normalized scalar, indicating the importance of node (B) to node (A). The attention coefficient is normalized in order to make the coefficients easily comparable across different nodes. Once the attention coefficients are obtained, the node embeddings are again used, this time to derive the updated node embedding. In the output layer, these node embeddings are again normalized using the softmax function. To use this as a design for the GCP, the final node embedding should represent a k-coloring of the graph. This can be done by letting the weight matrix map the input features from R^F to R^k. This then in turn also gives a clear interpretation of the node features after the weight matrix transformation, namely each feature represents a color. In [LLM+22], it is mentioned that the power of [VCC+17], is quite poor as the algorithm might not be able to color two adjacent nodes differently. However, by using the random initialization of the node features instead of the real world features, the problem with equivalent node pairs should be resolved, and by letting the algorithm train on the initial node embeddings the discriminatory power of the algorithm could be further strengthened. A great advantage of the [VCC+17] approach is that it first allows the network for assigning different importances to nodes of the same neighborhood and then uses these different importances assigned to these nodes when updating the node embedding. This allows the model to be more receptive to certain nodes, which in the graph coloring problem could be very useful. For example: Suppose that a node (A) has two nodes in its neighborhood, one node (B), that is adjacent to many other nodes, and one node (C), that is adjacent to no other node except for (A). Given that the coloring of (C) is much more flexible than the coloring of (B) we would like that the model gives (B) more importance than C. Given the right initial node embedding, the [VCC+17] model could facilitate this. The other papers aggregate the information of the neighborhood and then use this to update the node embedding and are therefore not able to facilitate this, for these reasons I have chosen to mainly use the architecture of [VCC+17] with the random features initialization.

Loss function Of the three approaches above, two use an unsupervised loss function [LLM+22], [SBZK22] and one uses a supervised loss function [LPAL19]. The supervised loss function has two main problems. First, a supervised loss function needs solutions in order for it to work. This, therefore, results in a 'chicken or the egg' situation. In order for the neural network to solve the most difficult GCP problems, it most likely needs to be trained on these types of problems, this training however requires the solutions for these problems. Second, even if these solutions are available, it is difficult to use them effectively for training. In a supervised learning approach, the loss is typically derived by checking if the nodes are classified correctly. A node thus has to be assigned a label. However, the question remains what should this label be? In the GCP it is not so much about the coloring of a node, but rather about the coloring of a node with respect to the coloring of the nodes in its neighborhood. When looking at Figure 3, two different feasible colorings are shown. Should we now label the middle node as blue as is done in coloring 1, or should we label the node red as done in coloring 2? When using labels, we are essentially training the algorithm to find exactly that one solution, whilst there are many other solutions.

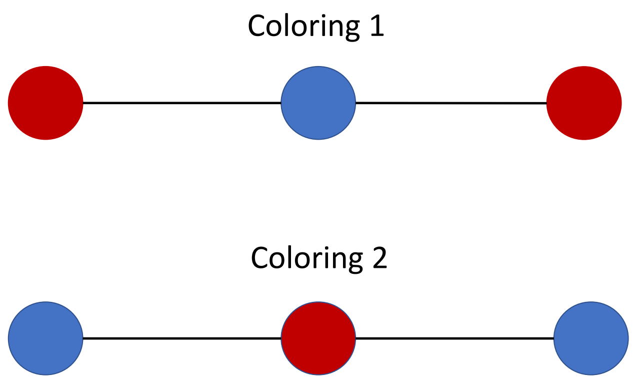


Figure 3: Two colorings for the same graph

In [LLM+22] the authors mention the second problem described above as one of the reasons why they have opted for an unsupervised approach. One way to use the solutions of known graphs is provided in [LPAL19] where the authors let the network only answer the question of whether k-coloring of the graph is possible. Here, the label provided to the network is not a coloring, but just a yes or no. The drawbacks of this approach are clear. First of all, the network does not provide a solution but only states whether or not a solution is possible. And besides that, the network as in [LPAL19] produces a distribution for multiple chromatic numbers, which means that for a certain graph, the chromatic number might be 4 with probability 0.1, 5 with probability 0.7 and 6 with probability 0.2. The two previously mentioned papers use an unsupervised loss function, [LLM+22] and [SBZK22] both use very intuitive approaches; the authors of [SBZK22] just assign penalties to adjacent nodes that are (partly) assigned the same coloring, and the authors of [LLM+22] do something very similar by deriving the euclidean distance between the node embeddings of two neighboring nodes. Given the intuitiveness of the unsupervised approach and the problems regarding the supervised approach, I have chosen to use an unsupervised loss function.

3 Data

In order to properly train and evaluate the algorithm various factors are important. First, there are requirements concerning the instances. The instances used to evaluate the performance of the algorithm are preferably often used in literature as this allows for a comparison with other approaches. Ideally, the instances used for evaluation would also have different properties regarding their structures, size, and connectedness, since this would give an indication as to how well the algorithm generalizes. A second factor that needs to be considered is the computational costs. Although the algorithm uses an unsupervised approach, various hyperparameters need to be trained which is computationally heavy. Keeping these two factors in mind, I have chosen to use the instances presented in Table 1. These instances consist of all instances used in the approaches from [LPAL19], [LLM+22] and [SBZK22], therefore the performance of the algorithm of this thesis can be compared with the performances of their respective algorithms. I have chosen to add the myciel2, myciel3, and myciel4 graphs in order to evaluate the performance of the algorithm on graphs that are considered easy because they do not contain triangles. The graphs used in training and evaluation are of various different types and therefore the performance of the algorithm on a set of different types of graphs can be evaluated thereby giving an indication of how well the algorithm generalizes. The following types of graphs are used.

- Book Graphs: Given a work of literature, a graph is created where each node represents a character. Two nodes are connected by an edge if the corresponding characters encounter each other in the book. These graphs are created for five classic works: Tolstoy's Anna Karenina (anna), Dicken's

David Copperfield (david), Homer's Iliad (homer), Twain's Huckleberry Finn (huck), and Hugo's Les Misérables (jean).

- Game Graphs. The games played in a college football season can be represented by a graph where the nodes represent each college team. Two teams are connected by an edge if they played each other during the season.
- Miles Graphs. The nodes represent a set of United States cities and two nodes are connected if the distance by road mileage, from 1947, between the two cities is smaller than some predetermined value.
- Queen Graphs. Given a n×n chessboard, a queen graph is a graph on n^2 nodes, each corresponding to a square of the board. Two nodes are connected by an edge if the corresponding squares are in the same row, column, or diagonal. Unlike some of the other graphs, the coloring problem on this graph has a natural interpretation: Given such a chessboard, is it possible to place n sets of n queens on the board so that no two queens of the same set are in the same row, column, or diagonal? The answer is yes if and only if the graph has coloring number n.
- MYC, Mycielski Graphs. Graphs based on the Mycielski transformation. These graphs are easy to solve because they are triangle free (clique number 2), the coloring number increases in problem size.
- MYC+, a generalization of myciel graphs with inserted nodes to increase graph size but not density
- MIZ, Graphs that are almost 3-colorable, but have a hard-to-find four clique embedded.

Besides the type, the following properties that influence the hardness of coloring are also provided in the table: the number of vertices, number of edges, and the density of the graph, calculated using Equation (11)

$$d = \frac{2|E|}{|V||V-1|} \tag{11}$$

and the chromatic number, obtained from [Gua15].

For all these graphs, I have removed the vertices which have no edges, and I have removed an edge B-A if the edge A-B already existed and vice versa. In the methodology, section 4, a detailed explanation is given which graphs are used for training subsection 4.2 and how the graphs are used for training subsection 4.1.

name	vertices	edges	density of the graph	χ	type
anna	138	493	0,052	11	Book
david	87	406	0,109	11	Book
homer	561	1628	0,010	13	Book
huck	74	301	0,111	11	Book
jean	80	256	0,081	10	Book
games120	120	638	0,089	9	Game
mug88_1	88	146	0,038	4	MIZ
queen5_5	25	160	0,533	5	Queen
queen6_6	36	290	0,460	7	Queen
queen7_7	49	476	0,405	7	Queen
queen8_8	64	728	0,361	9	Queen
queen8_12	96	1368	0,300	12	Queen
queen9_9	81	1056	0,326	10	Queen
queen11_11	121	1980	0,273	11	Queen
queen13_13	169	3328	0,234	13	Queen
miles250	128	389	0,048	8	Miles
myciel2	5	5	0,500	3	MYC
myciel3	11	20	0,364	4	MYC
myciel4	23	71	0,281	5	MYC
myciel5	47	236	0,218	6	MYC
myciel6	95	755	0,169	7	MYC
myciel7	191	2360	0,130	8	MYC
1_insertions_4	67	232	0,105	5	MYC+
2_insertions_4	149	541	0,049	5	MYC+

Table 1: Used instances and their properties

4 Methodology

In this thesis, I combine two different approaches for machine learning in order to design a machine learning approach for the k-coloring problem. In this subsection 4.1, I explain this combined design in detail. Thereafter, in subsection 4.3, I explain how a relaxation and two heuristics can be used as initialization and discuss how they can be incorporated into the combined design. Next, in subsection 4.4, I discuss how an infeasible coloring possibly can be transformed into a feasible coloring, by either using extra color or by using a heuristic.

4.1 The combined design

The design of the algorithm to solve the k-coloring problem for a large part consists of the elements of two papers: [VCC+17] for the message passing part of the algorithm and [SBZK22] for the unsupervised loss function. The general overview of the combined design is as follows. The input features, h, are randomly uniformly generated variables that undergo some linear transformations by matrix $\mathbf{W}$ and vector a, transforming the features into a node feature vector of length equal to the number of nodes and width k. Thereafter some non-linear transformation in the first-order neighborhood is performed on the node feature vector, which transforms it into a soft assignment of the colors of the nodes. The quality of this soft assignment solution is then evaluated using the loss function. The algorithm is first trained on a and $\mathbf{W}$, then using predetermined rules the trainable parameters are switched, and the algorithm is trained on h, depending on the predetermined rules the algorithm, thereafter, switches back and forth between training on a in combination with $\mathbf{W}$, and h. For each graph, a new set of trainable parameters is trained as in [LLM+22] and [SBZK22].

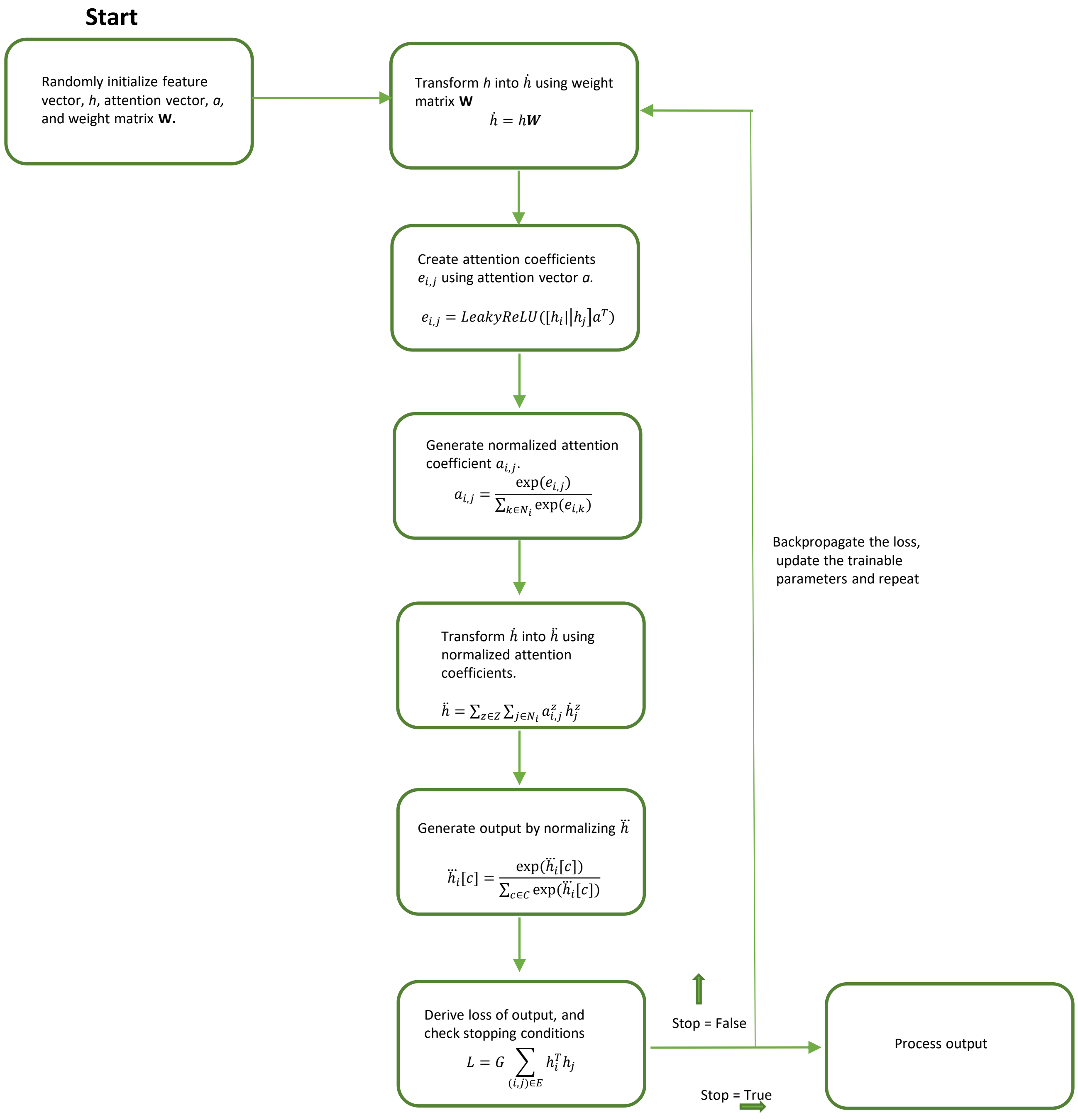

Start
Randomly initialize feature vector, h, attention vector, a, and weight matrix $\mathbf{W}$.
Transform h into $\dot{h}$ using weight matrix $\mathbf{W}$
$\dot{h} = h\boldsymbol{W}$
Create attention coefficients $e_{i,j}$ using attention vector a.
$e_{i,j} = LeakyReLU([h_i||h_j]a^T)$
Generate normalized attention coefficient $a_{i,j}$.
$a_{i,j} = \frac{\exp(e_{i,j})}{\sum_{k\in N_i}\exp(e_{i,k})}$
Transform $\dot{h}$ into $\ddot{h}$ using normalized attention coefficients.
$\ddot{h} = \sum_{z\in Z}\sum_{j\in N_i} a_{i,j}^z \dot{h}_j^z$
Generate output by normalizing $\ddot{h}$
$\dddot{h}_i[c] = \frac{\exp(\ddot{h}_i[c])}{\sum_{c\in C}\exp(\ddot{h}_i[c])}$
Derive loss of output, and check stopping conditions
$L = G\sum_{(i,j)\in E} h_i^T h_j$
Backpropagate the loss, update the trainable parameters and repeat
Stop = False
Stop = True
Process output

Figure 4: flow diagram algorithm

4.1.1 Initialization

The message passing part of the algorithm is mainly based on [VCC$^+$17], here the authors initialize their neural network with real-world features, and from these features, they try to build a function to the space of outcomes. They can do this because they use a supervised learning approach, hence they know the outcomes allowing them to try to build this function. In my design, the initial feature vector is generated randomly, as is done in [LLM$^+$22], [SBZK22], [LPAL19], and is a learnable parameter. The idea behind this is that I try to modify the starting point of the algorithm in order to give every node an indication of a coloring. If real world features would be used instead in a GNN with node embeddings, it is much more likely that two neighboring nodes would have very similar features whereas for the GCP it is important that two adjacent nodes are as different as possible in order to assign them different colors.

4.1.2 Forward Propagation

Let h, of the length equal to the number of nodes, and width equal to X, where X is the number of features of a node, be a feature vector. The first step in the network is to transform the feature vector, h, into $\dot{h}$. This is done by a matrix multiplication with weight matrix **W**. The weight matrix transforms the feature vector from size (N×X), where N is the number of nodes of the graph and X is the number of features of a node, to (N×k) where k is the number of colors for which the algorithm tries to find a feasible solution for the GCP. The weight matrix **W** is thus of size (X×k). In the [VCC+17] implementation, the function of the weight matrix **W** is to assign more importance to certain features of the node than to others, and also to reduce the dimensionality of the feature vector, making it of the same size as the k thus transforming some node properties into a vector containing the initial embedding of the color of that node. In the architecture of this thesis, the function of the weight matrix is mostly similar. The input vector is transformed from X to k, giving some indication of the coloring of the node. Here, however, the feature/properties of the initial feature vector, h, have no real-world meaning and are just randomly assigned values.

The next step is to apply some non-linear transformations on the initial node embeddings. These steps are all the same as in the [VCC+17] paper. To begin with, for every node in the first-order neighborhood, the attention coefficient is derived. The attention coefficients, $e_{i,j}$, indicate the importance of node j's features to node i's features. The attention coefficient $e_{i,j}$ is only calculated when node j is in the direct first-order neighborhood of node i, N_i. The attention coefficient is calculated by multiplying the concatenation of the feature vectors of the respective nodes by a weight vector a of the size $2k$ and then applying the LeakyRelu nonlinearity, see Equation (12) below. Depending on the structure of the graph, some colors are assigned to many nodes whilst other colors are assigned only to a very limited number of nodes. In the extreme case, for example, where one node is connected to all the nodes in a graph, the color assigned to this node can only be used once. The weight vector, a, offers the neural network the ability to make certain colors more important than other colors.

$$e_{i,j} = \mathit{LeakyRelu}([\dot{h_i}||\dot{h_j}]a^T) \tag{12}$$

Where || represents the horizontal concatenation-operation, $a \in R^{2k}$ and *LeakyRelu*, for some fixed parameter c, represents:

$$\begin{cases} x & if \;\; x > 0 \\ cx & otherwise \end{cases} \tag{13}$$

Thereafter the attention coefficients are normalized by applying the softmax function over $e_{i,j}$.

$$\alpha_{i,j} = softmax(e_{i,j}) = \frac{exp(e_{i,j})}{\sum_{k \in N_i} exp(e_{i,k})} \tag{14}$$

In the next step, the new feature vector for node i is calculated by multiplying the input feature vector of node j by its attention coefficients with respect to i, for all of the neighboring nodes and summing over these nodes see Equation (15). Thereafter, a softmax function, see Equation (16), is applied to every entry of the vector to ensure that the output vector sums to one, without having negative values for one of its entries. Multi-head attention, which allows the algorithm to perform the same part from multiple different starting positions, can also be applied by performing the same three steps, as mentioned above, for different initial inputs. This can be advantageous because, as [VSP+17] points out, multi-head attention allows the algorithm to start from different initial starting positions and therefore attend information from different representations at the same time. If multi-head attention is used, the feature vectors for node i are averaged, and thereafter the softmax function is applied.

$$\ddot{h_i} = \frac{1}{z}\sum_{z=1}^{Z}\sum_{j \in N_i} \alpha_{i,j}^z \dot{h_j^z} \tag{15}$$

$$\dddot{h}_{i,c} = \frac{exp(\ddot{h}_{i,c})}{\sum_{c \in k} exp(\ddot{h}_{i,c})} \tag{16}$$

Next, the output vector is evaluated. The loss function is derived from the [SBZK22] paper, which is one of the features which makes this thesis different from [VCC+17], that is, I use an unsupervised setup in contrast to that paper. In this thesis, for every edge in the graph, the respective output vectors of these endpoints are elementwise multiplied with each other and a penalty term G. The loss function here assures that if two adjacent nodes are assigned the same color, this elementwise product will be non-zero, and if two adjacent nodes are assigned different colors the elementwise product will be zero, which is exactly as desired for the GCP where two adjacent nodes cannot be assigned the same color. So, if this loss function is minimized and the total loss is 0, no two adjacent nodes are assigned the same color and the coloring assigned to the graph is feasible. The penalty term in my implementation and in the implementation of [SBZK22] is the same for every edge but could be used to assign a different importance to the loss of an edge.

$$L = G * \sum_{(i,j)\in E} \dddot{h}_i^T \dddot{h}_j \tag{17}$$

4.1.3 Trainable parameters

When designing a neural network, the design options are almost limitless. One would obviously like to pick the best option and hence find the optimal solution to the problem. However, finding this perfect design is very difficult. What one ideally would like to do, is not to choose between these design choices, but rather let this be a variable of the neural network. But when one would do this for all choices, the number of parameters would become very large, and the network training optimization problem would become too complex and therefore untrainable. Therefore often only the weights and biases of a network are used as trainable parameters. For most approaches, this is a logical choice, especially when the input of the neural network is fixed, for example, with some properties of a node. Yet, when this is not the case and the inputs of this network aren't fixed but instead are random starting points, it is perhaps not as clear what the trainable parameters should be.

The design above is, as mentioned before, a combination of two papers, [VCC+17] [SBZK22]. In the [VCC+17] paper, the authors use a supervised approach and their input feature vector consists of properties belonging to a node. They train on the weight matrix and the weight vector. Yet, in the [SBZK22] paper, the authors initialize with a random embedding and also train on their neural network. However, the same authors mention in this paper that finding a good starting point could be helpful. Therefore here the choice is made to first keep the feature vector, h, fixed and train the neural network parameters, $\mathbf{W}$ and a, for some time and then make a switch and keep the neural network parameters fixed and train on the feature vector. This switching between trainable parameters is the novice part of this design, and has, as far as I know, not been done before. When exactly the switch needs to be made is however not so clear. Below, three possible options for when to make the switch are outlined. All these three options will be tried in order to see which of those performs best.

- **Switch when the neural network parameters are trained for a fixed number of iterations and thereafter only train on the feature vector.** Here, the thought behind the switch is that we first train the neural network parameters such that they make some sense, thereafter the algorithm is allowed to change the random input vector in order to adapt this input vector as well as possible to the patterns learned by the neural network parameters.

- **Switch back and forth between neural network parameters and feature vector every time the loss function reaches a local minimum**. For this option, the thinking is somewhat similar to the first strategy, but here the switch occurs every time a local optimum is reached. The algorithm is said to be in a local minimum if the loss value has not changed for $\frac{1}{\%mbp}$ iterations, where $\%mbp$ is the percentage of nodes whose gradients are used during the training.

- **Switch back and forth every fixed number of iterations.** In this case, the intuition is to let the input vector and the neural network parameters consistently adapt to each other, instead of possibly overfitting the neural network parameters with regards to the input vector by training until the loss function does not decrease anymore as is the case with the above introduced local minimum switching.

4.1.4 Hyperparameters

Besides the trainable parameters, which are optimized in the loop of the algorithm, there are also hyperparameters, which are optimized outside the loop of the algorithm. In this section, all the hyperparameters are mentioned and explained. The first hyperparameter is the size of the feature matrix h, here the number of rows is fixed to be the number of vertices but the number of columns can be changed. The second hyperparameter is the value of c in the LeakyReLU equation(13). The third hyperparameter is the number of multi-head attention heads, which essentially performs the algorithm from a range of different starting positions and averages the soft assignment results to produce one soft assignment. The fourth hyperparameter is the percentage of nodes used to derive the loss for the stochastic gradient descent, from now on called the mini-batch percentage. And the fifth hyperparameter is the learning rate of the optimizer used in the gradient descent.

4.1.5 Complexity Analysis

In order to get a clear image of the time it takes to run certain instances, the time complexity is a key component. In this subsection the time complexity of the algorithm is derived. For every step of in Table 2 the complexity of that step is stated. The total complexity is the sum of all the steps, as denoted in the last row of the table. As can be observed from the table, the forward time complexity strongly depends on the size of the instance, the number of nodes, and the number of edges, as on k, with k being of a quadratic degree (step 1, since $k \leq F$). To get a feeling of which variables are most likely to increase the running time, it might be useful to look at some characteristics of the dataset, see Table 1, the average chromatic number is 8.08, whilst the average number of nodes is 104 and the average number of edges is 744. Looking at the total time complexity it cannot be concluded that one of the four factors, k, number of edges, number of nodes, and number of features, is dominating the time complexity. Here, k is at least quadratic but is way smaller than the number of nodes which again is way smaller than the number of edges.

step	number of basic operations needed	step description
1	N*k*F	$h\mathbf{W}$
2	2*k*E	$a(\dot{h}_i, \dot{h}_j)$
3	E	$LeakyREl u$
4	E*N (max when every node is connected to every node)	$softmax(e_{i,j})$
5	2*E*k	$\sum_{j \in N_i} \; a_{i,j} * \dot{h}_j$
6	N*k*k	$softmax(\ddot{h}_i)$
7	E*k	$(\dddot{h}_i)(\dddot{h}_j^{T})$
total	E*N + N*k*F + 5*E*k + N*k*k + E	total complexity

Table 2: Forward complexity, k = number of colors, E = number of edges, N = number of nodes and F = number of features for a node

4.1.6 Backward Propagation

After propagating forwards and calculating the loss, this loss has to be used to improve the neural network and hopefully decrease the loss in the next iteration. This is where the backward propagation comes in. The backward propagation calculates the losses corresponding to the trainable parameters of the neural network. This is done by continuously partially differentiating every step until the trainable parameters are reached. In the appendices, appendix C Table 17, the partial derivatives, for every step, are given. After the gradients of the trainable parameters are determined, the trainable parameters are updated using the Adam algorithm, pseudocode given in appendices, appendix B pseudocode 2. Regarding the backward time complexity, if the loss is derived using full gradient descent, then the backward time complexity almost equals the forward time complexity, going through the network forward or backward makes no difference because the network remains the same. The backward complexity is slightly higher because for the softmax 3 basic operations extra are needed. However, if stochastic gradient descent is used, where the batch size equals only a few or even just one node, the backward time complexity will be much lower than the forward time complexity. The overall time it takes the algorithm will most likely be determined by the number of iterations that are needed to find a feasible solution as the forward and backward propagation has to be performed each iteration.

4.1.7 Escaping local minima

During the training phase, the algorithm can get stuck in some local minima. As mentioned before, in this algorithm a solution is said to be a local minimum if the loss value after $\frac{1}{\%mbp}$ iterations has not changed. In order to escape this local minimum multiple strategies can be used. One previously discussed strategy is to switch the trainable parameters of the neural network. Another strategy is a very simple but often-used strategy, adding some noise to the neural network. It is difficult to estimate how much noise needs to be added to the neural network in order to change the solution. Therefore, in this algorithm, the choice is made to start with adding relatively small noise, and if the solution remains the same, increase the noise bit by bit. The noise is added to the features completely at the beginning. First, the features of one node are adapted, and if this results in no change in the loss the next iteration of the algorithm the features of two extra node nodes are adapted, if this again results in no change in the loss the next time 3 extra nodes are adapted. This process is repeated until there is a change in the loss when evaluating the the algorithm.

4.1.8 Obtaining a solution

The output vector of the algorithm has non-integer values for every node. In order to turn the output vector into feasible solutions for GCP, these need to be set to integer values. The algorithm can finish in two ways. Either the value of the loss function approaches 0 within a predetermined margin, such that a feasible solution is likely found, or the time limit is reached. In the first case, the output vector of the last iteration is used to create an integer solution. In the second case, the best-saved output vector is used to create an integer solution. Since the algorithm ideally improves at each iteration, it is computationally inefficient to save the best solution for each iteration. Therefore, the output vector is only saved if the corresponding loss of the output vector is a certain percentage below the corresponding loss value of the previously saved output vector. In both cases, the output vector is set to an integer value in the same way: the argmax over colors for the final embedding vector of each node is set to 1, while the other colors are set to 0.

4.2 Training Strategy

In order to get the best results out of this algorithm, decisions about settings and choices have to be made. In this subsection, the strategy used to train the hyperparameters and make the choices is laid out. Since the computational costs of hyperparameter training are high, and given that it is wise to split the dataset into a training and a testing part, the first choice needed to be made is the selection of the dataset. Although I understand that it is far from optimal to train on only a few graphs, I have made the choice to train only on 5 graphs and train only for one coloring number per graph. I have done this because I have limited computational resources and a full training needs to be performed for each graph. In selecting these graphs I have tried to select as many different types, outlined in the data section, of graphs as possible. From each type, except for the MIZ type since this type is very similar to the MYC type, I have selected a graph. Next, I have tried to ensure that the average number of nodes (116.6), edges (873.4), and chromatic number (8.8) of the training graphs is of the same order as the average number of nodes (104.4), edges (744.3), and chromatic number (8.08) of all the graphs in Table 1. The selected graphs are: games120, jean, miles250, queen8_8, and myciel7. The properties of the graphs can be seen below in the Table 3. When training the

name	#nodes	#edges	density	χ [Gua15]	type
jean	80	254	0.31	10	BOOK
games120	120	638	0.19	9	GAMES
miles250	128	387	0.33	8	MILES
queen8_8	64	728	0.09	9	MYC
myciel7	191	2360	0.08	8	QUEEN

Table 3: Training graphs

hyperparameters, there is a trade-off between computation time and the accuracy of the optimization. To find the best values for the hyperparameters, ideally, you would want to train all the hyperparameters in all their different combinations and see which performs best. However, given the number of hyperparameters, this is impossible, since the computational cost would simply be too high. To reduce the computational costs of the hyperparameters optimization, all the hyperparameters could be optimized sequentially. This means that we first determine an order in which we want to optimize the hyperparameters and next fix all but the first hyperparameter of that order and then optimize the first hyperparameter. Thereafter, we fix this first hyperparameter and optimize the second hyperparameter whilst keeping the other hyperparameters fixed. This is done until all hyperparameters are optimized. Various hyperparameters have a very big influence on each other. When these are sequentially optimized it could very well be that some good settings are missed. Therefore, the choice is made to semi-sequentially optimize the hyperparameters. Meaning that some hyperparameters, which have a strong influence on each other, will be optimized together. In this algorithm, the hyperparameters are: the size of h, c, #attention heads, %mbp, and learning rate optimizer. The last two of these hyperparameters probably strongly influence each other. Therefore it may be necessary to optimize these hyperparameters together. The other hyperparameters can be optimized sequentially. Since the performance of the optimizer is key in the performance of the algorithm, first the two hyperparameters which influence this performance are optimized using a grid-search, see Table 4.2. Next the size of h is optimized, after which α is optimized and lastly #attention heads is optimized.

%mbp	learning rate
0.01	0.001
0.031	0.0031
0.1	0.01
0.31	0.031
1	

Table 4: grid-search parameters

To pick the best performing hyperparameter I use the following procedure. First, I look at the minimum loss values obtained during the run of the instances for a certain hyperparameter and next sum these loss values to find the total loss of the instances for this hyperparameter. Second, I check whether or not (one of) the instances had very high volatility, a quite low loss value is reached somewhere during the run but the final loss value is way higher ($\times 2$). If (one of) the instances has very high volatility, then, when another hyperparameter its total loss value is similar, but has way less volatility, this one is chosen instead. Third, in the case of two or more loss values that are very similar, I factor in the 'potential' of the instances, meaning that I look at whether or not the change in the loss value at the end of an instance is zero or if there is still a (slight) decreasing trend. So the training procedure is the following: First for all 3 methods (*oneSwitch, switchWhenLM* and, *switchPeriodically*) I optimize the *%mbp* and the learning rate together, by running the neural network for all 5 training graphs (3) for all 20 combinations. Next, I pick the best combination for every method, keeping the factors discussed above in mind. Thereafter, for every method, I pick the best size of h using the best values for *%mbp* and the learning rate. Following that, I do the same for c with the best values for *%mbp* and the learning rate and size of h. Lastly, I pick the best performing value for the #attention heads. This results in the best performing values of the hyperparameters for the three different methods.

4.3 Working in combination with heuristics

The algorithm is described above as an isolated heuristic but it can also be used to (possibly) improve solutions from relaxations or heuristics. Instead of initializing the algorithm randomly, as happens in an isolated heuristic, it can also be initialized using a heuristic or a relaxation. In this subsection, I first discuss the way the algorithm is initialized when using a heuristic or relaxation. After this, I provide an explanation of how the different heuristics and the relaxation, used in this thesis, can be combined with the proposed algorithm.

4.3.1 Combining the algorithm with heuristics or relaxations in initialization

In order to use either a heuristic or a relaxation together with the algorithm, we can either use the solution of the algorithm as a starting point for those heuristics and relaxations, or use the solutions of a heuristic or a relaxation as a starting point for the algorithm. The latter is not straightforward to do since the input features do not correspond to a solution of the k-coloring problem. To use the heuristic or relaxation as a meaningful starting point of the algorithm, the output vector must be the same as the heuristic/relaxation solution. To achieve this, ideally, the corresponding input features and weights of the algorithm are determined. However, in practice, this is very difficult because there is no one-to-one mapping of input features and the output vector since the input of the algorithm is a matrix N×X and the output vector is a matrix of the size N×k. Another way to achieve that the output vector is the same as the heuristic/relaxation solution is to make use of the last step before the loss function and add a bias term to the input of the softmax function. By adding the bias every input of the softmax function can be transformed into the wanted heuristic/relaxation solution. This works as follows. The algorithm still initializes at a random input vector, and the first forward step proceeds exactly the same as when the algorithm is initialized randomly. However, instead of calculating the loss and propagating backward, the output is compared to the heuristic/relaxation solution, and the needed biases are calculated to ensure the output in the next iteration is the same as the heuristic/relaxation solution. Here for every node, the following derivation is done. For a problem where three colors are assigned a node has three features. Let's say the node has the following output vector before the softmax: [a, b, c] the output after the softmax is $[\frac{e^a}{e^a+e^b+e^c}, \frac{e^b}{e^a+e^b+e^c}, \frac{e^c}{e^a+e^b+e^c}]$, the wanted output of the softmax is however [d,e,f]. Now we add the bias vector [h,i,j] to the input then the equations which need to be solved become: $\frac{e^{a+h}}{e^{a+h}+e^{b+i}+e^{c+j}} = d$, $\frac{e^{b+i}}{e^{a+h}+e^{b+i}+e^{c+j}} = e$, $\frac{e^{c+j}}{e^{a+h}+e^{b+i}+e^{c+j}} = f$. I solve this system in the following way:

$$\begin{gathered}
\frac{e^{\mathrm{a+h}}}{e^{\mathrm{a+h}}+e^{\mathrm{b+i}}+e^{\mathrm{c+j}}} = \mathrm{d}, \frac{e^{\mathrm{b+i}}}{e^{\mathrm{a+h}}+e^{\mathrm{b+i}}+e^{\mathrm{c+j}}} = \mathrm{e}, \frac{e^{\mathrm{c+j}}}{e^{\mathrm{a+h}}+e^{\mathrm{b+i}}+e^{\mathrm{c+j}}} = \mathrm{f}. \\
e^{\mathrm{a+h}} = e^{\mathrm{a+h}} + e^{\mathrm{b+i}} + e^{\mathrm{c+j}} \times \mathrm{d} \\
\text{set } e^{\mathrm{a+h}} + e^{\mathrm{b+i}} + e^{\mathrm{c+j}} = e^{\mathrm{a}} + e^{\mathrm{b}} + e^{\mathrm{c}} \\
e^{\mathrm{a+h}} = \mathrm{d} \times e^{\mathrm{a}} + e^{\mathrm{b}} + e^{\mathrm{c}} \\
\ln e^{\mathrm{a+h}} = \ln(\mathrm{d} \times e^{20}) \\
\mathrm{h} = \ln(\mathrm{d} \times e^{\mathrm{a}} + e^{\mathrm{b}} + e^{\mathrm{c}}) - \mathrm{a} \\
\mathrm{i} = \ln(\mathrm{e} \times e^{\mathrm{a}} + e^{\mathrm{b}} + e^{\mathrm{c}}) - \mathrm{b} \\
\mathrm{j} = \ln(\mathrm{f} \times e^{\mathrm{a}} + e^{\mathrm{b}} + e^{\mathrm{c}}) - \mathrm{c}
\end{gathered} \tag{18}$$

In the next iteration, the bias is added to the output before the softmax function, and the loss of the solution (now the same as the heuristic/relaxation solution) is derived and backpropagated. The algorithm thereafter continuous in the same way as when it is initialized randomly, see Figure 5.

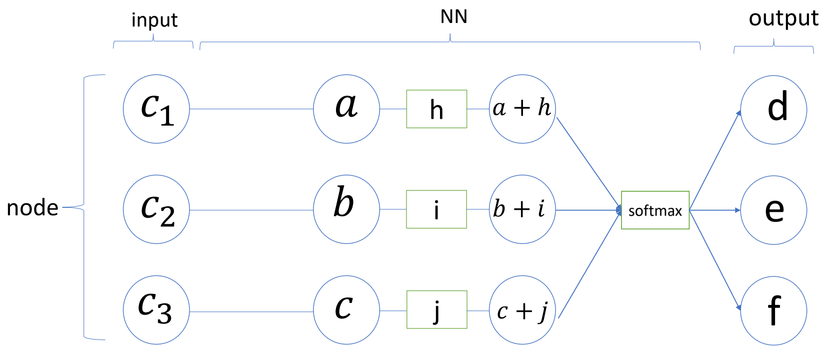


Figure 5: heuristic initialization

Relaxation The linear programming relaxation solution found, consisting of equations (1-3) and (6-9), is used as the initialization for the algorithm. If the number of colors used in the linear programming relaxation does not match k, some extra colors need to be added to the solution of the linear programming relaxation in order to match k.

Recursive Largest First (RLF) The recursive largest first algorithm tries to build independent sets in order to color the graph. The way in which this is done is described in 2.2. The RLF always produces a feasible coloring, which means that in order to use it, when initializing the algorithm, the solution has to be adapted. In this thesis I remove the color least used in the RLF solution and assign a random other used color to the nodes which were previously assigned the removed color. The pseudocode of this algorithm can be found in the appendix B pseudocode 3.

TABUCOL In contrast to the other two heuristics, the TABUCOL algorithm does not always produces a feasible coloring. Therefore, the only thing that needs to be done to use the TABUCOL as an initialization is to assign the TABUCOL algorithm a chromatic number for which it is not able to solve the GCP in a given time limit and then use the output of the TABUCOL algorithm for the initialization. The pseudocode of the TABUCOL algorithm can be found in the appendix B pseudocode 5.

4.4 Improving the output of the neural network

The neural network will not find a feasible solution for every value for k. It would obviously be beneficial if we could turn an infeasible solution into a feasible solution with or without using extra colors. In this thesis, I use one method, in subsection 4.4.1, that guarantees a feasible solution but most likely needs some extra colors, and I use one method, in subsection 4.4.2, that tries to find a feasible solution without needing extra colors.

4.4.1 Generating a feasible solution using color correction

The input of the color correction algorithm is the integer infeasible output of the algorithm. This output has a number of conflicting edges. The goal is to color one of the nodes of a conflicting edge with either a color already used in the graph or with an extra color. We can create a new graph, $\mathrm{G}_{\text{conflict}}$ ($\mathrm{N}_{\text{conflict}}$,$\mathrm{E}_{\text{conflict}}$+$\mathrm{E}_{\text{no conflict}}$), where the nodes are all the nodes of the conflicting edges, let call these nodes $\mathrm{N}_{\text{conflict}}$, and the edges are the conflicting edges, $\mathrm{E}_{\text{conflict}}$, and all the edges which begin at a node of a conflicting edge and end at a node of a different conflicting edge, $\mathrm{E}_{\text{no conflict}}$. For every $\mathrm{E}_{\text{conflict}}$ in $\mathrm{G}_{\text{conflict}}$ we need to change the coloring of one of the two nodes using the minimum amount of new colors. This is very similar to the GCP, however, here not all nodes need to be assigned a color. Let us call this new problem, $\mathrm{GCP}_{conflict}$. To solve the $\mathrm{GCP}_{\text{conflict}}$, we can first try to simplify the problem, by looking if we can color one of the nodes in the conflicting edges using colors already used in the infeasible solution. We do this by looking if one color is not used in the first order neighborhood of every node in $\mathrm{N}_{conflict}$. If we find one color that is not used, we can assign this color to the corresponding node of the conflicting edge, and we can thereafter remove the node pair corresponding to this conflicting edge and all the edges that are connected to this node pair from $\mathrm{G}_{\text{conflict}}$, thereby simplifying the $\mathrm{GCP}_{\text{conflict}}$. If there are still conflicting edges remaining we have to find a way to resolve those edges. Given the similarities between the GCP and the $\mathrm{GCP}_{\text{conflict}}$ we could try to solve the $\mathrm{GCP}_{\text{conflict}}$ using one of the heuristics used for the GCP, DSATUR, TABUCOL, and RLF. Of these three heuristics, the RLF, with some adaptations, probably works best. In the $\mathrm{GCP}_{\text{conflict}}$ once all the conflicting edges of a node are resolved, we can remove these conflicting edges and nodes from the graph thereby simplifying the remaining problem. The TABUCOL algorithm does not allow for removing nodes and edges during the running of the algorithm and is therefore probably not very useful in this problem. The DSATUR algorithm colors the graph by coloring the most saturated nodes first, thereby initially quickly increasing the number of colors used and coloring the nodes that are most connected. Which seems logical if you need to color every node of the graph, here, however, we have to color only half of the nodes or less, and we want to pick the half of the nodes which are least connected with each other. Therefore it seems logical to generate independent sets using an adapted version of the RLF algorithm, where in contrast to RLF algorithm, we do not pick the node of maximal degree, but the node of minimal degree and try to build an as big as possible independent set. After each independent set we can remove a lot of nodes and edges thereby greatly reducing the difficulty of the problem. The pseudocode of the color correction algorithm is given below.

Algorithm 1 Color correction algorithm

```
{C} = are colors used in infeasible solution
for conflicting edge in conflicting edges do        ▷ color nodes with existing color where possible
    for node in nodes conflicting edge do
        if c∈C is not used in first order neighborhood then
            color node = c
            remove conflicting edge
            break
        end if
    end for
end for
$N_{\text{conflict}}$ = node of a conflicting edge
$E_{\text{conflict}}$ = conflicting edges
$N_{\text{no conflict}}$ = edges between $N_{\text{conflict}}$ that are not $\in E_{\text{conflict}}$
create $G_{\text{conflict}}(N_{\text{conflict}}, E_{\text{conflict}}+E_{\text{no conflict}})$
colornumber = size {C}
while (number of conflicting edges> 0) do        ▷ introduce new color
    determine a vertex x of minimal degree in G
    colornumber = colornumber + 1
    F(x) = colornumber
    NN = set of non-neighbors of x
    while (cardinality of NN > 0) do
        maxcn = 1
        ydegree = 1
        for (every vertex z in NN) do
            cn = number of common neigbors of z and x
            if (degree(z) < ydegree or ( degree(z) == ydegree and maxcn(z) < cn)) then
                y = z
                ydegree = degree(y)
                maxcn = cn
            end if
        end for
        if (maxcn == 0) then
            y = vertex of maximal degree in NN
        end if
        F(y) = colornumber
        contract y into x
        update the set NN of non-neighbours of x
    end while
    for every node that is not connected to any conflicted edges do
        remove node and all edges connected to this node from $G_{\text{conflict}}$
    end for
end while
```

4.4.2 Generating a feasible solution using TABUCOL

For this approach, the input is again the integer output of the algorithm. Using the algorithm proposed in this thesis as a hotstart of the TABUCOL algorithm is very straightforward, instead of generating a random starting solution, we initialize the algorithm using the output of the algorithm. When TABUCOL is used to improve the infeasible solution it is, however, far from guaranteed that a feasible coloring arises, since, first, it could be possible that no better solution exists, and second, even though a better solution exists TABUCOL might not be able to find it.

5 Computational experiments and results

All the experiments were performed on one laptop. This laptop has an AMD RYZEN 5 4000 series processor and 16GB RAM memory and it is running on Windows 10. All methods were coded in spyder 5.3.1 using python 3.8.8. To speed up the time-intensive running process, I have performed the computational experiments with only one random seed. All three methods change their learnable parameters somewhere during their execution, the equations (19,20,21) indicate when this switch is made for each of the three methods. In these equations the *switchNumber*=$\frac{\#nodesInGraph}{(\#nodesInGraph*mbp)}$, where *mbp* is the percentage of nodes for which the loss is derived.

$$oneSwitch : switchNumber \times 15 \tag{19}$$

$$switchPeriodically : \frac{iterationCount}{25} \tag{20}$$

$$switchWhenLM : \text{if iterations since last switch} > switchNumber \ \& \ \frac{\text{loss}_i}{\text{loss}_{i-10}} > 0.99 \tag{21}$$

For the computational experiments where the $mbp < 1$, the loss of the whole solution is calculated once every *switchNumber* iterations. Furthermore, the penalty term in the loss function, eq (17), is set to 100. The goal of this section is to answer the three main questions of this thesis.

1. How does the combined design of [VCC+17] and [SBZK22] perform for the k-coloring problem?
2. Does training on the features improve the performance of a GNN for the k-coloring problem?
3. How can heuristics and relaxation be used to aid the algorithm and vice-versa?

In order to answer these questions, this section is laid out in the following way. First, I discuss the training of the hyperparameters and report the best hyperparameters in subsection 5.1. Next, I answer the first and second questions in subsection 5.2. In this subsection, I first compare the different methods, *oneSwitch*, *switchPeriodically* and *switchWhenLM*, of the algorithm with each other and with a method, called *onlyAW*, that is exactly the same as these methods except that this method is not allowed to train on the features. Thereafter, still in subsection 5.2, I compare the best performing method, of the three methods presented in this thesis, with three neural network approaches for the k-coloring problem: [SBZK22],[LLM+22],[LPAL19] and thereby try to answer the second question. Following this in subsection 5.3, I give an explanation as to why the methods introduced so far perform relatively well on one type of graphs, mycielski graphs, and perform substantially worse on the other types of graphs and introduce an adaptation to the design of the algorithm. After this, I test this adapted design and once again try to answer the second question. Lastly, in subsection 5.4, I show how the algorithm performs when it is initialized using a heuristic and show heuristics can be used to help the algorithm in turning infeasible solutions into feasible solutions.

5.1 Performance of oneSwitch, switchPeriodically, and switchWhenLM on training graphs

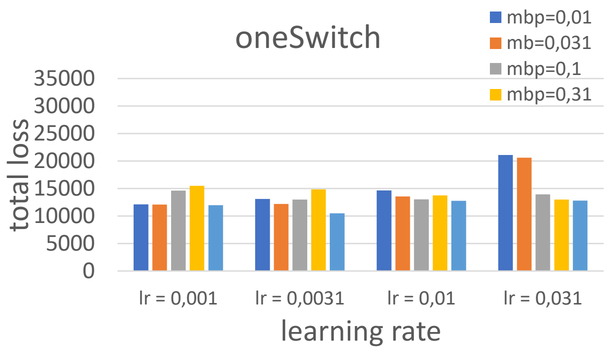
In this subsection, the final results of the hyperparameter optimization are presented. In order to maintain the balance between the total running time and the accuracy of the results, I have chosen a running time of 20 minutes per instance, since the preliminary results showed that after 20 minutes the loss of the instances did not change much more. I have also, as explained before, chosen to train the hyperparameters of the algorithm on only 5 graphs. This is a very small sample of graphs and will therefore most likely not yield the best settings for the hyperparameters. Interpreting the results of the hyperparameters will also be more difficult as patterns will be less clear since the patterns can more easily be disturbed by the randomness of the algorithm when using only a few graphs. I have, however, chosen to do this because including more graphs would increase the total computation time too much. During the first steps of the hyperparameter optimization, I noticed that the algorithm was only able to solve the myciel7 graph (for some hyperparameter settings). However, using the optimization procedure as explained in section 4.1.4 would result in choosing hyperparameters for which the algorithm was not able to solve the myciel7 graph. Therefore, I have chosen to optimize the algorithm for both the myciel7 graph and the other graphs, from here on *4trainingGraphs*. An outline of the hyperparameter settings used during hyperparameter optimization and their respective results can be found in appendices, appendix C Table 18.

5.1.1 Best performing hyperparameters *4trainingGraphs*

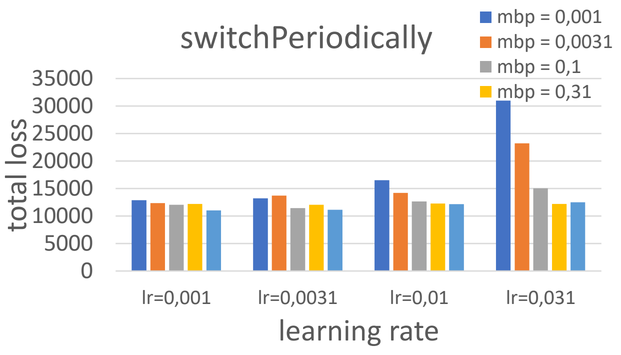
method	mbp	lr	ivs	leakyReLU	#attention heads
oneSwitch	1	0.0031	3	0.1	1
switchPeriodically	1	0.001	9	0.2	5
switchWhenLM	0.31	0.0031	3	0.3	1

Table 5: Best performing hyperparameters obtained for the *4trainingGraphs*. Where the *mbp* is the minibatch-percentage, *lr* is the learing rate of the optimizer, *ivs* is the integer by which k is multiplied to determine the number of features of a node, *leakyReLU* is the coefficient, c, in the leakyReLU Equation (12) and #attentionheads is the number of attention heads.

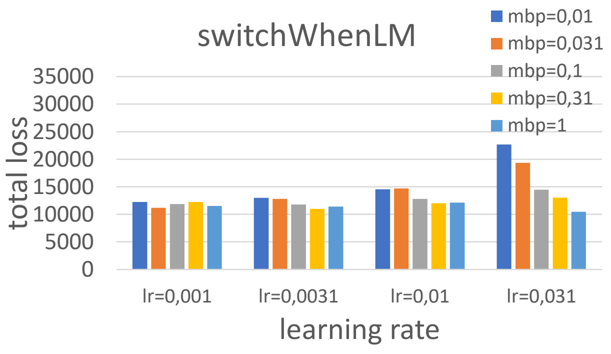
In Table 5 the best performing combination of hyperparameters for each of the three methods is given. In the appendices, appendix C Table 18, the losses of the training graphs for all used combinations of hyperparameters can be found. When looking at the mini-batch percentage (*mbp*), the fraction of nodes for which the loss is determined, it seems to be that an increase in *mbp*, independent of the learning rate, for all three methods improves –or at least does not worsen– the performance of the algorithm, see Figure 6. A possible explanation for this phenomenon might be that a larger *mbp* provides more information and therefore leads to a more accurate calculation of the gradient which in its place results in a more accurate updating of the trainable parameters.



(a) oneSwitch



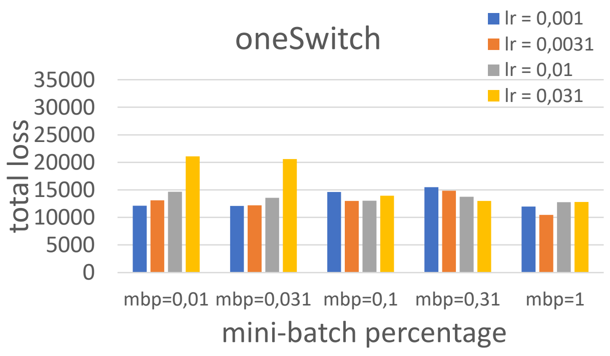
(b) switchPeriodically



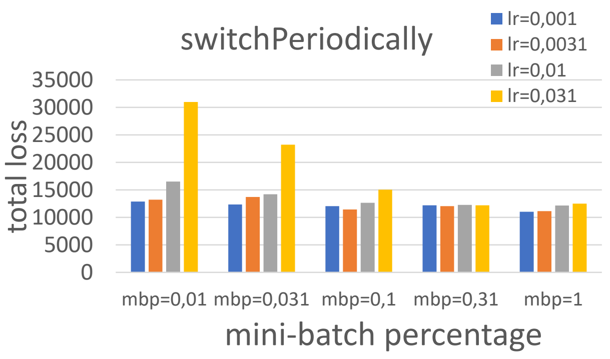
(c) switchWhenLM

Figure 6: The total loss, loss of the *4traininggraphs* added together, for each combination of the learning rate and mini-batch percentage, with the mini-batch percentage grouped together.

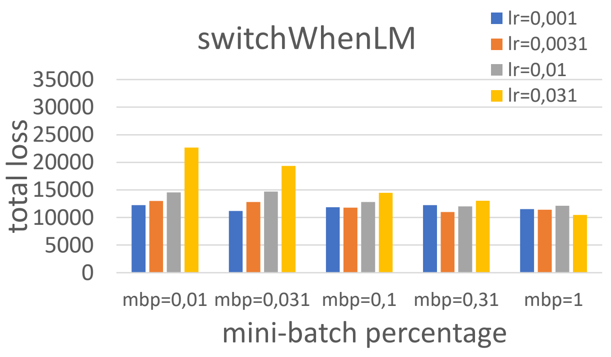
Next, when looking at the learning rate (*lr*), Figure 7, I expected a correlation between the *lr* and the *mbp*, where for a low *mbp* a low *lr* would work best, and a for a high *mbp* a high *lr* might work best because the higher the *mbp* the more certainty about the direction of the next step and thus the greater the *lr* can be. This seems to be only partially true. For the low *mbp*, in all three methods, it can be clearly observed that when the *lr* increase the performance worsens. However, the contrary does not seem to be the case; for the high mbp the high lr are far closer in performance, but the lower $lr's$ still perform better. Obviously, the choice for the lr plays a big part in this correlation. Since the problem is highly non-linear the best gradients at certain moments are the best direction only locally, a too high lr could make too big of a jump in this locally optimal direction which would not lead to a globally better solution. If I had chosen $lr's$ that would have been closer together, the correlation between the lr and the mbp might have been clearer.



(a) oneSwitch



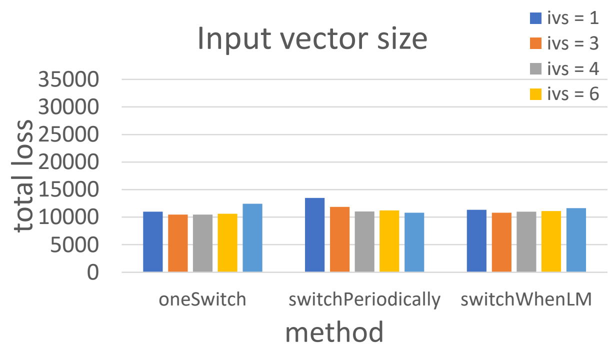
(b) switchPeriodically



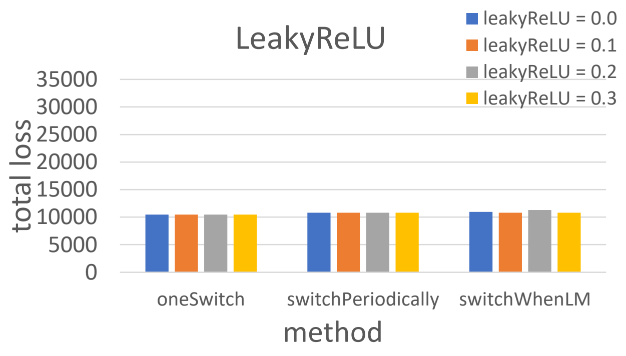
(c) switchWhenLM

Figure 7: The total loss, loss of the *4trainingraphs* added together, for each combination of the learning rate and mini-batch percentage, with the learning rate grouped together.

As for the input vector size (ivs) and for the leakyRelU parameter x, both do not seem to have a substantial effect on the performance of the algorithm. The ivs performance probably strongly depends on the random initialization of its features and the difference between the performances of the ivs for the different methods can be explained by this. For both the *oneSwitch* and *switchPeriodically* methods the leakyReLU parameter has almost identical results, but for the *switchWhenLM* method this is not the case. This probably has to do with the nature of the methods. Where the *oneSwitch* and the *switchPeriodically* method switch at a predetermined point, the *switchWhenLM* method switches when the local minimum criterion is reached. The influence of the leakyReLU parameter is very minimal and only changes the loss value a tiny bit. For the *switchWhenLM* method this tiny change can, however, result in switching between trainable parameters one step earlier or later, which in turn can lead to noticeable changes and therefore very different results.



(a) input vector size



(b) leakyReLU

Figure 8: The total loss, loss of the *4traininggraphs* added together, for the different values for the input vector size (left) and the leakyReLU parameter (right).

When we look at the #attention heads, we see the same pattern occurring for each method. #attention heads = 1, performs well. Thereafter for # attention heads = 2 the total loss increases quite drastically and then the total loss decreases for each extra attention head, until at #attention heads = 5, where the total loss reaches an almost similar value to #attention heads = 1. Increasing the number of attention heads obviously takes more time and, since the time limit is reached for all graphs used in training, thus worsens the performance of the algorithm because fewer iterations are performed. However, an increase of the number of attention heads could also improve the performance of the algorithm. The balance between these two opposite factors could be a possible explanation for the pattern observed for the number of attention heads. It could be that initially, the worsening factor outweighs the beneficial factor and that by increasing the number of attention heads this becomes more balanced.

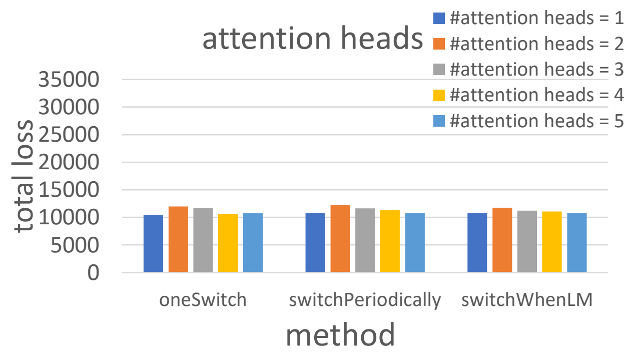


Figure 9: The total loss, loss of the *4traininggraphs* added together, for the different values for the #attention heads.

In the figure below, Figure 10, the loss of the graph (y-axis) is plotted against the iteration (x-axis). The maximum loss of a graph is 100× *numberOfEdges*, the begin-point and endpoint of every edge then have the same color. The minimum loss is 0, the begin-point and endpoint of every edge then have a different color. When we look closely at the figures, we can observe some similar patterns for all methods and graphs. The algorithm starts with a randomly initialized starting point, which can easily be improved, but quite quickly afterward the methods do not seem able to approve the solution anymore. When looking at the individual color values per node at the end of the computational experiment, it seems that most nodes have one color that has a value >0.97. The very slowly decreasing trend of the graphs at the end of the computational experiment might therefore be explained by the continuous approaching of a value of 1 for these nodes.

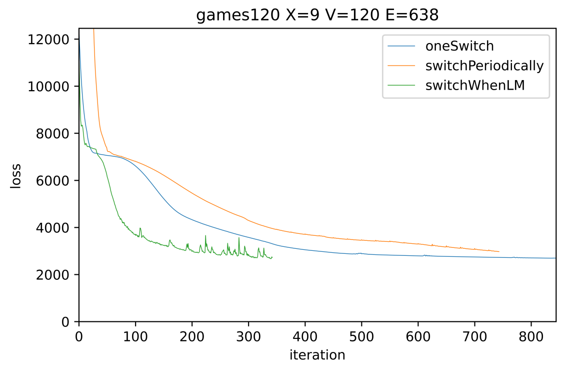
Another pattern visible in all graphs is that the *switchWhenLM* method, which has a lower *mbp* than the other methods, seems to be able to execute far fewer iterations in the time limit. This might seem contradictory as a lower *mbp* should result in more iterations since the loss of fewer vertices has to be computed.

When the loss of fewer vertices is computed the loss obviously is also lower, and might even be 0. Therefore the loss of the whole solution has to be computed sometimes in order to check whether a feasible solution is found, and also to evaluate the performance of the algorithm. As already explained above, in this algorithm the loss of the whole solution therefore is calculated $\frac{1}{mbp}$ times. When this loss is calculated this value is also added to the figure. The actual iterations of the algorithm are thus $\frac{1}{mbp}$ times the iterations in the figures.

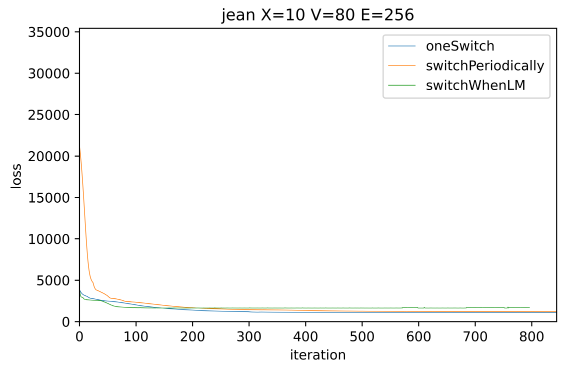
Another observation that can be made is that the loss of the *switchPeriodically* and *switchWhenLM* methods show quite capricious behavior, most clearly visible in the games120 graph for the *switchWhenLM* method. This happens when these methods switch between trainable parameters. In the Adam optimizer, Algorithm 2, there are four possible hyperparameters. The learning rate, the weight decay, and two coefficients β_1 and β_2, which are used in combination with the gradients to derive the first and second moments, which in turn is used to derive the stepsize. In the first iteration of the optimizer, the terms m_0 and v_0 are 0, therefore the first step the optimizer makes is not a running average of the first and second moment of the gradients, but just the first and second moment of the gradient derived from the first iteration multiplied with the $\beta's$, see Equation (22).

$$m_1 = \beta_1 m_0 + (1 - \beta_1) g_t \ v_1 = \beta_2 v_0 + (1 - \beta_2) g_t^2 \tag{22}$$

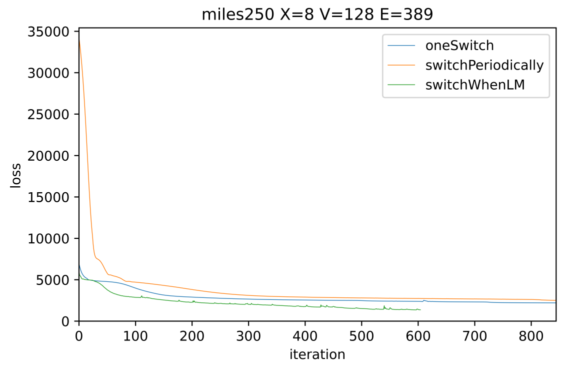
This, therefore, leads, I think to too big of a first step, resulting in an increase rather than a decrease of the loss. If both β_1 and β_2 are increased, this overshooting will happen less, however, this also affects the performance of the rest of the optimizer. Since this would require entirely new hyperparameter optimization, I have opted not to change the $\beta's$ and leave them at their default values.



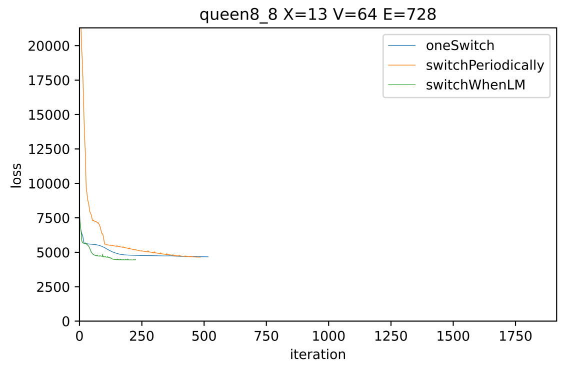
(a) games120



(b) jean



(c) miles250



(d) queen8_8

Figure 10: Loss value of best performing hyperparameters, where X = the inputted k. In appendix A, figure 19 the same figures can be found but more zoomed in on the final losses.

5.1.2 Training on myciel7 graph

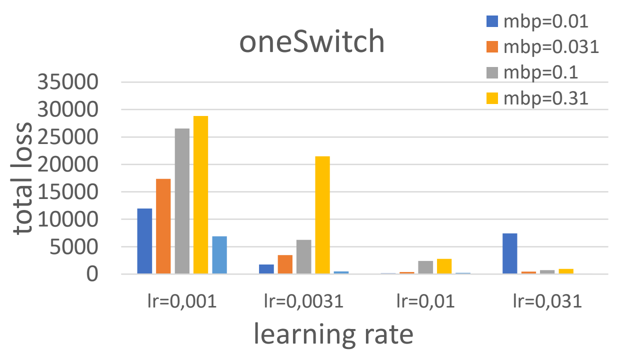
The best performing hyperparameters for the different methods for the myciel7 graph are in the table below, Table 6. To pick the best performing hyperparameters, the same rules were used as for the graphs used in the *4graphsGroup*. However, since for some hyperparameters combinations feasible solutions were found, the loss of these combinations was obviously the same. When this happened, I picked the hyperparameter that found the feasible solution the fastest.

method	mbp	lr	ivs	leakyReLU	#attention heads
oneSwitch	1	0.0031	4	0	3
switchPeriodically	0.031	0.01	6	0	1
switchWhenLM	1	0.01	4	0.2	1

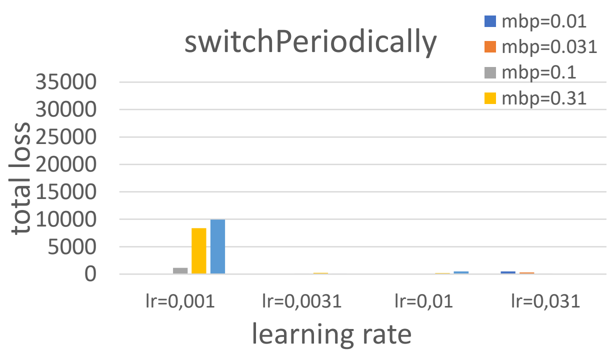
Table 6: The best performing hyperparameters for the myciel7 graph.

Interpreting the different performances of the various hyperparameters for this single graph is not entirely straightforward. It obviously does not help that there is only one graph which due to the random nature of the algorithm produces somewhat capricious results. The optimal hyperparameters are also quite different from the *4traingGrpahs*. One of the explanations could be that fine-tuning the hyperparameters on graphs for which the algorithm in its current state is unable to find solutions is very different than fine-tuning the algorithm for graphs it is able to find solutions for.

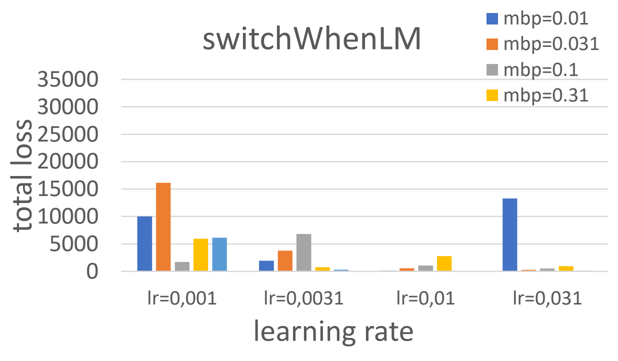
One of the patterns that did became clear for me was that the higher (0.01,0.031) learning rates for all three methods seem to perform better as can be seen in Figure 11. In this figure and for the other figures in this subsubsection the y-axis for the switchPeriodically graph is not the total loss but the time it took for the myciel7 graph to run. If the time here is 1200s, no feasible solution has been found.



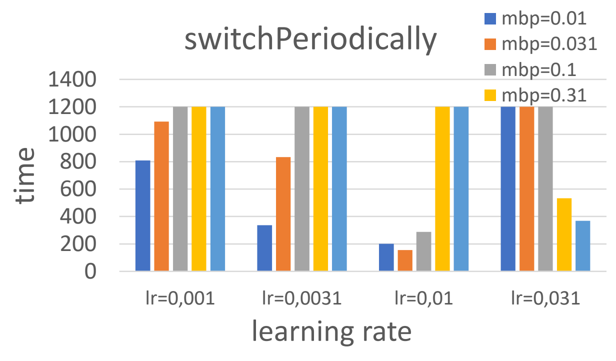
(a) oneSwitch



(b) switchPeriodically



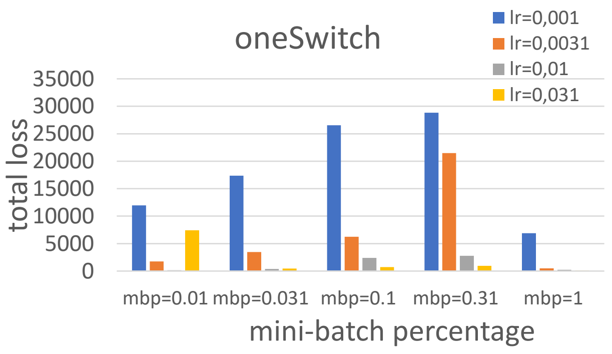
(c) switchWhenLM



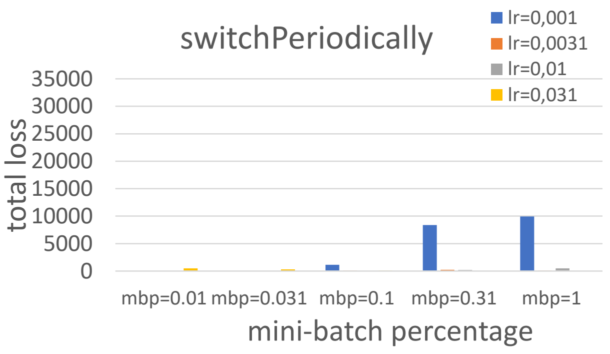
(d) switchPeriodically time

Figure 11: For the *oneSwitch* and *switchWhenLM* methods I only show the loss when the time limit is reached. For the *switchPeriodically* method I show the loss as well as the time it took to find a feasible solution for the myciel7, if the time for a combination is 1200, no feasible solution was found for this combination. For all three methods, the respective loss or time is shown for each combination of learning rate and mini-batch percentage, with the combinations with the same mini-batch percentage grouped together.

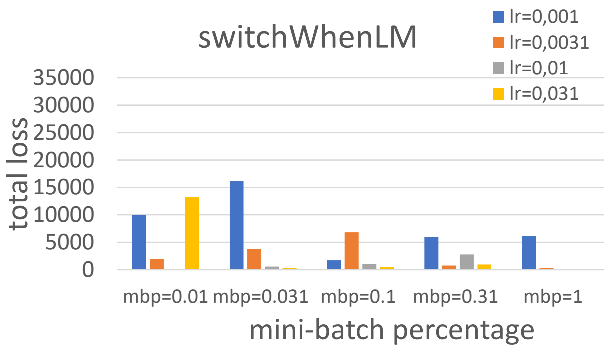
Next, for the mini-batch percentage, Figure 12, it seems that it does not matter so much since, for all three methods, across the different mini-batch percentages, the best loss value is very similar.



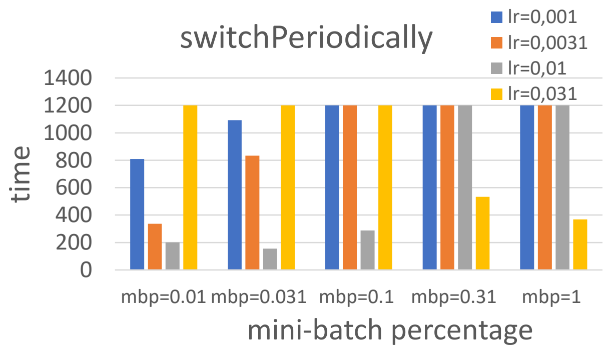
(a) oneSwitch



(b) switchPeriodically



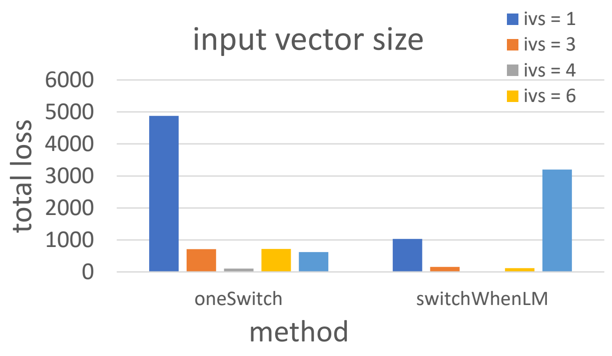
(c) switchWhenLM



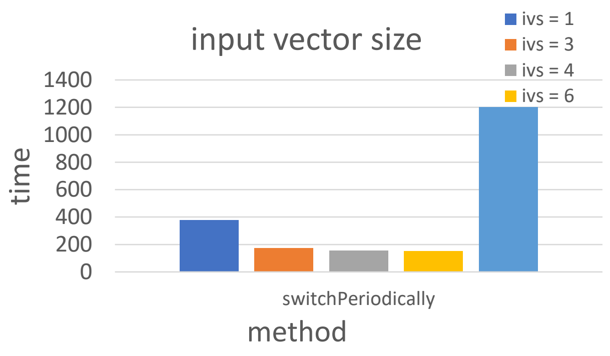
(d) switchPeriodically time

Figure 12: For the *oneSwitch* and *switchWhenLM* methods I show the loss when the time limit is reached. For the *switchPeriodically* method I show the loss as well as the time it took to find a feasible solution for the myciel7, if the time for a combination is 1200, no feasible solution was found for this combination. For all three methods the respective loss or time is shown for each combination of learning rate and mini-batch percentage, with the combinations with the same learning rate grouped together.

When looking at the *ivs* in Figure 13, the middle (3,4,6) values seem to perform better than the high and low values (1,9) for all three methods. The *ivs* largely determines the number of parameters of the network, the number of parameters is roughly *targetNumber*×number of nodes×*ivs*. A bigger *ivs* might allow the network to more delicately change its direction to the solution. However, since the inputvector itself is randomly initialized, it also adds more chaos to the network, making it more difficult to find the solution. Hence, it is not strange to say that balance is key.



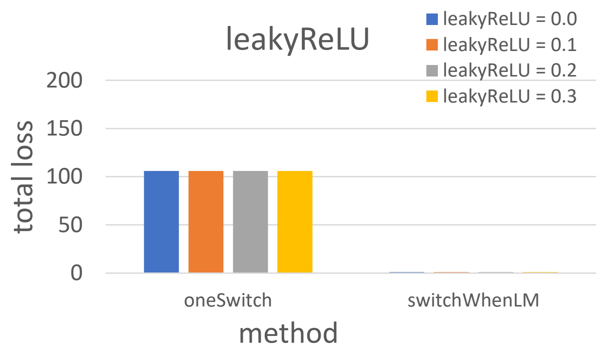
(a) oneSwitch and swichtWhenLM



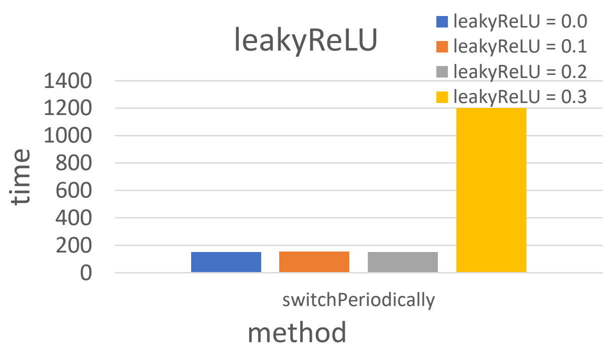
(b) switchPeriodically

Figure 13: For the *oneSwitch*(left) and *switchWhenLM*(left) methods I show the loss when the time limit is reached for each of the different input vector sizes. For the *switchPeriodically* method (right) I show the time it took to find a feasible solution for the myciel7 for each of the different input vector sizes, if the time for a combination is 1200, no feasible solution was found for this combination.

If we look at the *leakyReLU* value in Figure 14, all values are almost identical, as was also the case for the other *4trainigGraphs*. Only for the *switchPeriodically* method, the *leakyReLU* value of 0.3 has a significantly different value to the other values. I do not have a clear interpretation for this result.



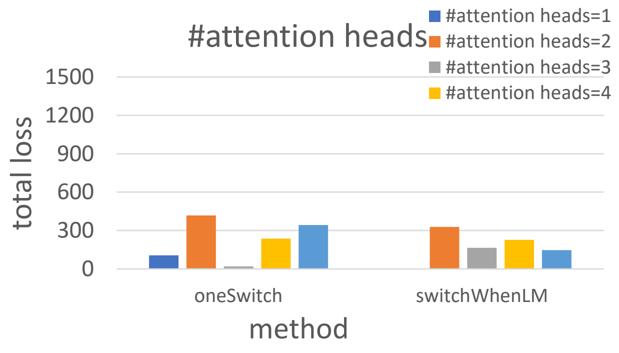
(a) oneSwitch and switchWhenLM



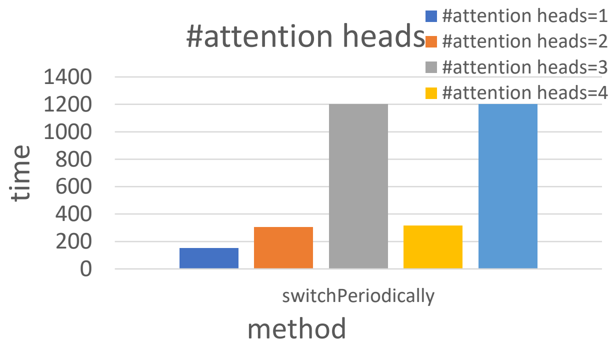
(b) switchPeriodically

Figure 14: For the *oneSwitch*(left) and *switchWhenLM*(left) methods I show the loss when the time limit is reached for each of the different leakyReLU parameters. For the *switchPeriodically* method (right) I show the time it took to find a feasible solution for the myciel7 for each of the different leakyReLU parameters, if the time for a combination is 1200, no feasible solution was found for this combination.

Lastly, if we look at the #attention heads, Figure 15, it seems that for the *switchPeriodically* and *switchWhenLM* methods #attention heads = 1, performs best. For the *oneSwitch* method this is not the case, here the #attention heads = 3, performs best.



(a) oneSwitch and switchWhenLM



(b) switchPeriodically

Figure 15: For the *oneSwitch*(left) and *switchWhenLM*(left) methods I show the loss when the time limit is reached for each of the different #attention heads. For the *switchPeriodically* method (right) I show the time it took to find a feasible solution for the myciel7 for each of the different #attention heads, if the time for a combination is 1200, no feasible solution was found for this combination.

5.2 Performance of oneSwitch, swichtPeriodically, and switchWhenLM on test graphs

In this subsection, I present the result of the different methods on the test graphs, which are all the graphs in Table 1. These are graphs of various types. In the hyperparameter optimization, I noticed a difference between the *mycielski* graphs and the *non-mycielski* graphs and have therefore chosen to train two different models. The best hyperparameters settings for the different methods can be found in Table 7. I have used the *mycielski* settings only to test the *mycielski* graphs, Table 8, and I have used the *non-mycielski* settings for both the *mycielski* graphs and the *non-mycielski* graphs, Table 10. In Table 1, also the hyperparameter settings for a method not mentioned before, *onlyAW* can be found. This method is completely similar to the other methods except that I do not allow this method to train on the features. In doing so I can evaluate whether or not it is beneficial to train on the features. The hyperparameters of the *onlyAW* method are optimized using exactly the same procedure and settings as the other methods. An overview of the results can be found in the Appendices, appendix C Table 18.

mycielski						
method	mbp	lr	ivs	leakyReLU	#attention heads	
oneSwitch	1	0.0031	4	0	3	
switchPeriodically	0.031	0.01	6	0	1	
switchWhenLM	1	0.01	4	0.2	1	
onlyAW	0.031	0.031	6	0	1	
non-mycielski						
method	mbp	lr	ivs	leakyReLU	#attention heads	
oneSwitch	1	0.0031	3	0.1	1	
switchPeriodically	1	0.001	9	0.2	5	
switchWhenLM	0.31	0.0031	3	0.3	1	
onlyAW	0.031	0.01	4	0.1	1	

Table 7: Hyperpameter settings for all the methods

In Table 8 the results of the different methods on the *mycielski* graphs are presented. Since almost all methods found feasible colorings for the graphs, the time it took to find these k-colorings is presented instead of the loss. If the time equals the time limit of 1200s, the method did not find a feasible coloring for the graph, in this case the loss at the time limit can be found between the brackets. For these cases where no feasible k-colorings were found before the time limit, the number of conflicting edges after rounding, ce, and the number of conflicting edges after a first-order neighborhood color correction, cn, can be found as well as the chromatic number, kcc, found after the whole color correction algorithm. The first-order neighborhood color correction consists of checking whether or not, for a conflicting edge, one of the nodes can be assigned an existing color that is not yet used in its neighborhood.

Now, if we look at the performances of the different methods we notice that both the *switchPeriodically* and *switchWhenLM* methods are able to find feasible solutions for all the problems within the time limit. The *oneSwitch* and *onlyAW* methods, however, were not able to find feasible solutions for all graphs, the *oneSwitch* method was not able to solve the 2_insertions_4 graph, the myciel6 and myciel7 graph, and the *onlyAW* method was not able to solve the myciel7 graph. For all graph-method combinations, where no feasible solution was found before the time limit, a feasible k-coloring was found either after rounding or after a first-order neighborhood color correction with the target k number of colors. Next if we look at the influence of training on the feature vector we see that the *switchPeriodically* method performs much better, *switchWhenLM* substantially better, and the *oneSwitch* method worse, than the *onlyAW* method. Clearly, we can not conclude that training on the features is always better than not training on the features. However, when used properly we can conclude that training on the features can greatly enhance the performance of the algorithm on the *mycielksi* graphs. This seems particularly true when we keep switching between training on the features and training on the other parameters, as both the methods where this continuous switching happens, *switchPeriodically* and *switchWhenLM*, clearly outperform the methods where this continuous switching does not happen *oneSwitch* and *onlyAW*. An explanation for this result could be that when continuous switching is used the set of parameters that is fixed is better because the fixed parameters have been recently trained on and therefore maybe better adapted to provide a lower coloring, which in term allows a more accurate training of the current trainable parameters.

	graph properties				*oneSwitch*				*switchPeriodically*				*switchWhenLM*				*onlyAW*			
graph	nodes	edges	χ	k	loss	ce	cn	k-cc	time	ce	cn	k-cc	time	ce	cn	k-cc	time	ce	cn	k-cc
1_insertions_4	67	232	5	5	269s	-	-	-	23s	-	-	-	66s	-	-	-	118s	-	-	-
2_insertions_4	149	541	5	5	1200s (106)	1	0	5	56s	-	-	-	179s	-	-	-	496s	-	-	-
myciel2	5	5	3	3	0.94s	-	-	-	0.29s	-	-	-	0.88s	-	-	-	0.39s	-	-	-
myciel3	11	20	4	4	6.95s	-	-	-	0.72s	-	-	-	1.58s	-	-	-	3.55s	-	-	-
myciel4	23	71	5	5	48.4s	-	-	-	2.28s	-	-	-	9.91s	-	-	-	9.86s	-	-	-
myciel5	47	236	6	6	777s	-	-	-	11.31s	-	-	-	63s	-	-	-	54s	-	-	-
myciel6	95	755	7	7	1200s (206)	2	0	7	33.9s	-	-	-	290s	-	-	-	460s	-	-	-
myciel7	191	2360	8	8	1200s (18)	0	0	8	149s	-	-	-	1198s	-	-	-	1200s (24)	0	0	8

Table 8: Performance on myciel graphs. For a given k, I report the results of the algorithm. If the algorithm found a feasible solution for the given *targetNumber*, I report only the time it took to find this solution. If the time limit of 1200s was reached I report the time and between brackets the final loss. For the graph method combinations where no feasible solutions are found I also report, the number of conflicting edges after rounding, *ce*, the number of conflicting edges after first-order neighborhood color correction, *cn*, and the chromatic number after the whole color correction algorithm, $k-cc$. The chromatic number χ is obtained from [Gua15].

Next, it stands out that the *switchPeriodically* method clearly outperforms the other methods and is substantially faster on all graphs. As mentioned above, the methods using continuous switching perform better than the methods that do not use continuous switching. The performance gap between the *switchPeriodically* method and the *switchWhenLM* could possibly be explained by the fact that the *switchWhenLM* method first needs to get stuck into a local minimum in order for it to switch and therefore might not profit as much from the performance-benefits of switching. In Table 9, this method is compared with the methods from the authors of [SBZK22], [LLM+22] and [LPAL19]. In [SBZK22] and [LPAL19] no time is mentioned, so comparing the time it took to find the k-colorings is difficult. But it can be concluded that at least the algorithm is able to find colorings for the same k as these two methods. Next, when comparing with [LLM+22], they are much faster in solving these graphs for their respective target k. Comparing the running times is, however, difficult since a lot of factors affect the running time, such as the computational power and the way the neural network is implemented.

	graph properties				switchPeriodically	[SBZK22]			[LLM+22]		[LPAL19]
graph	nodes	edges	chi	k	time	k	ce	X_greedy	k	ce	X_greedy
1_insertions_4	67	232	5	5	23s	-	-	-	-	-	5
2_insertions_4	149	541	5	5	56s	-	-	-	4	1 (101s)	5
myciel5	47	236	6	6	11.31s	6	0	6	6	0 (0.01s)	6
myciel6	95	755	7	7	33.9s	7	0	7	7	0 (0.01s)	7
myciel7	191	2360	8	8	149s	-	-	-	-	-	8

Table 9: Comparison with other methods. I compare the best performing method with three neural network approaches for the GCP. For the [SBZK22] approach, I report their target k, the number of conflicting edges and the chromatic number after their color correction algorithm, X_{greedy}. For [LLM+22] approach, I report their target k, and their number of conflicting edges for this k and between brackets the time it took to find these solutions. For the [LPAL19] approach, I report their chromatic number after their color correction algorithm, X_{greedy}. If for an approach-graph combination, a ' - ' is reported this approach did not consider this graph.

In Table 10, the performance of the different methods on the *mycielski* as well as on the *non-mycielski* graphs are presented, using the hyperparameters settings of the *non-mycielski* training graphs. The table shows the loss of the algorithm at the time limit. If the loss is 0, a feasible k-coloring is found, the time it took to find this feasible k-coloring can then be found between brackets. The rest of the notation is similar to Table 8: the conflicting edges after rounding, *ce*, the conflicting edges after first-order neighborhood color correction, *cn*, and the chromatic number after the whole color correction algorithm, $k - cc$.

A few things stand out when observing the results of these methods in general. First, for all methods solutions are found for *mycielski* graphs only. For all the other graphs the loss remains far from zero and the number of conflicting edges after rounding is also quite far from desired. Second, the difference between *ce* and *cn* is striking. The only difference between these two solutions is that for *cn* the color of some wrongly colored nodes is changed if in their first-order neighborhood no node was colored using that color. This procedure is very simplistic and changing this color influences no other edges since this color is not used in this neighborhood. Ideally, the algorithm should be able to change the color of these nodes quite easily since this lowers the loss function. This big difference between *ce* and *cn* indicates that the algorithm is not very good at finding these 'easy' changes of colors.

Now, if we start to compare these methods with each other, we can observe that when we look at the loss and the number of conflicting edges after rounding, the three methods, where training on the features is allowed, *oneSwitch, switchPeriodically, switchWhenLM* outperform the method, where this is not allowed, *onlAW*. For almost all graphs the *oneSwitch*, *switchPeriodically* and *switchWhenLM* have a substantially lower loss and number of conflicting edges. This is especially true for the *book, games* and *miles* graphs for which the losses and number of conflicting edges for the *oneSwitch*, *switchPeriodically* and *switchWhenLM* methods are much lower than the losses and number of conflicting edges for the *onlyAW* method. The *onlyAW* method performs better on the myciel5, myciel6, and myciel7 graphs than the *oneSwitch, switchPeriodically* and *switchWhenLM* graph, but this is not the case when the hyperparameter settings for the *mycielski* graphs are used, Table 8. However, when we look at the total number of colors needed and the total number of conflicting edges after first-order neighborhood correction, the performance gap between the *onlyAW* method and the other methods shrinks. The total number of colors needed is not an ideal indicator since it does not reflect potential advantages/disadvantages for particular types of graphs, but it allows for a comparison of the performance of the methods in a more general way and therefore is still used here. The total number of colors needed is only slightly lower for the *oneSwitch* and *switchWhenLM* method, and even a bit higher for the *switchPeriodically* method. However, some optimal coloring solutions, respectively the 1_insertions_4, 2_insertions_4, david, anna and mugg88_1 graphs using the *oneSwitch* method, 1_insertions_4, 2_insertions_4 and mugg88_1 graphs using the *switchPeriodically* method, and the 1_insertions_4, 2_insertions_4, games120 graph and the myciel7 graph, are found that were not found for the *onlyAW* method. The metrics *cn* and the $k - cc$ are, however, maybe not the best metrics to judge the performance of the methods, since these are not direct results of the algorithm but indirect results. Taking the above into account, we can conclude that training on features seems beneficial for the loss, the number of conflicting edges and in most cases also the final coloring number.

Next, if we compare the *oneSwitch, switchPeriodically* and *switchWhenLM* methods with each other, it seems that the oneSwitch method performs the best. It has the lowest total loss, the lowest ce the lowest cn, and the lowest $k - cc$. The performance of the *switchWhenLM* method is, however, not far from the performance of the *oneSwitch* method. Which makes it difficult to conclude if switching only once is the best choice or if switching more often is the best choice.

	graph properties				*oneSwitch*				*switchPeriodically*				*switchWhenLM*				*onlyAW*			
	nodes	edges	chi	k	loss	ce	cn	k-cc	loss	ce	cn	k-cc	loss	ce	cn	k-cc	loss	ce	cn	k-cc
1_insertions_4	67	232	5	5	**0 (283s)**	-	-	**5**	**0 (153s)**	-	-	**5**	**0 (157s)**	-	-	**5**	100	1	0	**5**
2_insertions_4	149	541	5	5	101	1	0	**5**	**0 (385s)**	-	-	**5**	**0 (496s)**	-	-	**5**	311	3	2	6
anna	138	493	11	11	1276	30	3	13	1376	29	4	13	1064	28	3	13	2230	57	4	13
david	87	406	11	11	2115	74	0	**11**	1616	37	3	13	2413	117	1	12	2446	146	1	12
games120	120	638	9	9	2696	66	0	**9**	2973	62	1	10	2926	54	0	**9**	4787	169	3	10
homer	561	1628	13	13	5187	146	9	15	8176	156	9	17	5922	153	7	15	10465	300	12	17
huck	74	301	11	11	1511	82	0	**11**	1759	100	1	12	1502	57	0	**11**	1938	125	0	**11**
jean	80	256	10	10	1107	45	1	11	1003	49	0	**10**	875	48	1	11	1796	86	0	**10**
miles250	128	389	8	8	1981	57	3	10	2151	56	7	11	2110	60	3	10	3367	137	3	10
mug88_1	88	146	4	4	267	6	0	**4**	383	8	0	**4**	**0 (873s)**	-	-	**4**	1136	17	1	5
myciel2	5	5	3	3	**0 (0.83s)**	-	-	**3**	**0 (2.2s)**	-	-	**3**	**0 (0.94s)**	-	-	**3**	**0 (0.76s)**	-	-	**3**
myciel3	11	20	4	4	**0 (5.5s)**	-	-	**4**	**0 (11s)**	-	-	**4**	**0 (5.34s)**	-	-	**4**	**0 (6.8s)**	-	-	**4**
myciel4	23	71	5	5	**0 (696s)**	-	-	**5**	**0 (45s)**	-	-	**5**	**0 (27s)**	-	-	**5**	**0 (29s)**	-	-	**5**
myciel5	47	236	6	6	100	1	0	**6**	100	1	1	7	**0 (241s)**	-	-	**6**	**0 (125s)**	-	-	**6**
myciel6	95	755	7	7	12	-	-	**7**	**0 (1004s)**	-	-	**7**	**0 (1083s)**	-	-	**7**	**0 (676s)**	-	-	**7**
myciel7	191	2360	8	8	710	3	1	9	2274	3	1	9	588	1	0	**8**	1206	10	7	10
queen5_5	25	160	5	5	1232	12	11	8	1734	21	18	8	1004	10	10	7	2099	51	14	8
queen6_6	36	290	7	9	2353	89	3	11	2418	74	6	12	2175	75	5	11	2402	75	3	11
queen7_7	49	476	7	10	3534	128	9	12	3092	37	11	13	4012	128	7	13	3447	136	5	12
queen8_8	64	728	9	13	4669	200	3	14	4636	173	6	15	4885	226	3	15	4970	310	23	17
queen8_12	96	1368	12	14	8703	331	25	17	9187	271	39	18	8578	402	23	17	9469	153	3	15
queen9_9	81	1056	10	12	7471	234	26	16	7606	232	17	16	7450	257	32	16	7895	356	23	15
queen11_11	121	1980	11	14	12574	442	57	19	13868	636	66	19	13202	427	52	19	13005	557	42	20
queen13_13	169	3328	13	17	19586	549	44	22	24533	986	35	22	19657	778	51	21	19469	733	49	22
sum	2505	17863	194	214	77185	2496	195	247	88885	2931	225	258	78363	2821	198	247	92538	3422	195	254

Table 10: Performance methods in comparison with *onlyAW* method. For a given k, I report the results of the algorithm. I report the loss value at the moment the time limit is reached, if the loss value is zero, the algorithm has found a feasible k-coloring, and the time it took to find this k-coloring is reported between brackets. For the graph method combinations where no feasible solutions are found I also report, the number of conflicting edges after rounding, ce, the number of conflicting edges after first-order neighborhood color correction, cn, and the chromatic number after the whole color correction algorithm, $k - cc$. The chromatic number χ is obtained from [Gua15]. Lastly, a number is bold when the optimal coloring is obtained.

In Table 11, the results of the *oneSwitch* method are compared with the methods from the authors of [LLM+22], [SBZK22] and [LPAL19]. Here similar to Table 9 the target k for the methods of the authors is provided as well as the number of conflicting edges and or their chromatic number after some color correction. The *oneSwitch* method is clearly outperformed by the algorithm of [LLM+22] and [SBZK22]. As a rough indicator of quality, the total number of colors found by each algorithm is used to compare the algorithms, as mentioned above this not a perfect indicator since it does not reflect the performance for particular types of graphs. For the graphs in common with [SBZK22] the sum of all the colors for the *oneSwitch* method is 156 whilst the sum of the algorithm of [SBZK22] is 123. For the graphs in common with [LLM+22], where [LLM+22] has no conflicting edges, the sum of all the colors for the *oneSwitch* method is 108 whilst the sum of the algorithm of [LLM+22] is 95. The sum of the *oneSwitch* method for the graphs in common with [LPAL19] is closer 216 and 206 respectively. It is clear that the other methods perform better, especially on the non-mycielski graphs.

	graph properties				oneSwitch				[SBZK22]			[LLM+22]		[LPAL19]	TABUCOL
	nodes	edges	chi	k	loss	ce	cn	cc	k	ce	χ_{greedy}	k	ce	χ_{greedy}	k
1_insertions_4	67	232	5	5	**0 (283s)**	0	0	**5**	-	-	-	-	-	**5**	**5**
2_insertions_4	149	541	5	5	101	1	0	**5**	-	-	-	4	1	**5**	**5**
anna	138	493	11	11	1276	30	3	13	11	0	**11**	11	**0**	12	**11**
david	87	406	11	11	2115	74	0	**11**	-	-	-	11	**0**	12	**11**
games120	120	638	9	9	2696	66	0	**9**	-	-	-	9	**0**	-	**9**
homer	561	1628	13	13	5187	146	9	15	-	-	-	13	**0**	15	**13**
huck	74	301	11	11	1511	82	0	**11**	-	-	-	11	**0**	**11**	**11**
jean	80	256	10	10	1107	45	1	11	10	0	**10**	10	**0**	**10**	**10**
miles250	128	389	8	8	1981	57	3	10	-	-	-	-	-	-	-
mug88_1	88	146	4	4	267	6	0	**4**	-	-	-	-	-	**4**	**4**
myciel2	5	5	3	3	**0 (0.83s)**	0	0	**3**	-	-	-	-	-	-	-
myciel3	11	20	4	4	**0 (5.5s)**	0	0	**4**	-	-	-	-	-	-	-
myciel4	23	71	5	5	**0 (696s)**	0	0	**5**	-	-	-	-	-	-	-
myciel5	47	236	6	6	100	1	0	**6**	6	0	**6**	6	**0**	**6**	**6**
myciel6	95	755	7	7	12.3	0	0	**7**	7	0	**7**	7	**0**	**7**	**7**
myciel7	191	2360	8	8	710	3	1	9	-	-	-	-	-	**8**	-
queen5_5	25	160	5	5	1232	12	11	8	5	0	**5**	5	**0**	8	**5**
queen6_6	36	290	7	9	2353	89	3	11	7	1	8	-	-	11	8
queen7_7	49	476	7	10	3534	128	9	12	7	4	9	7	9	10	8
queen8_8	64	728	9	13	4669	200	3	14	9	2	10	8	2	13	11
queen8_12	96	1368	12	14	8703	331	25	17	12	2	13	12	**0**	15	**12**
queen9_9	81	1056	10	12	7471	234	26	16	10	3	12	9	6	16	11
queen11_11	121	1980	11	14	12574	442	57	19	11	19	15	11	21	17	-
queen13_13	169	3328	13	17	19586	549	44	22	13	27	17	13	33	21	-
sum	2505	17863	194	214	77185.3	2496	195	247	108	58	123	147	72	206	147

Table 11: Comparing with other approaches. I compare the best performing method with three neural network approaches for the GCP. For the [SBZK22] approach, I report their target k, the number of conflicting edges, and the chromatic number after their color correction algorithm, X_{greedy}. For [LLM+22] approach, I report their target k, and their number of conflicting edges for this k and between brackets the time it took to find these solutions. For the [LPAL19] approach, I report their chromatic number after their color correction algorithm, X_{greedy}. A number is bold when the optimal coloring is obtained.

5.3 Adapted model

The difference between the performance of the mycielski graphs and the non-mycielski graphs is striking. The mycielksi graphs are graphs that have a maximum clique number of 2. The graphs therefore do not contain any triangles or even more connected structures. A reason for the poor performance on the non-mycielski graphs could be this structural difference between the graphs. To further investigate whether this is the problem, I made a few graphs myself consisting of very simple structures, see figure 17. The idea behind the triangle and crown graphs is that in a triangle two vertices have the same common neighbor, and that for a crown graph the two bottom vertices have multiple common neighbors. These common neighbors influence the node embedding of the two adjacent vertices. In the case of the triangle, if two nodes have an exactly similar feature matrix, these nodes will be colored exactly the same, despite their adjacency, because they have exactly the same neighborhood.

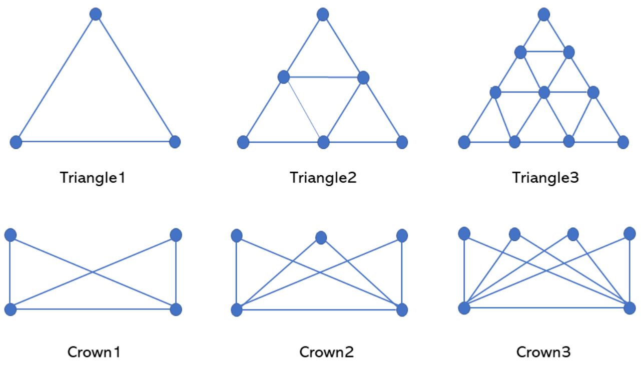
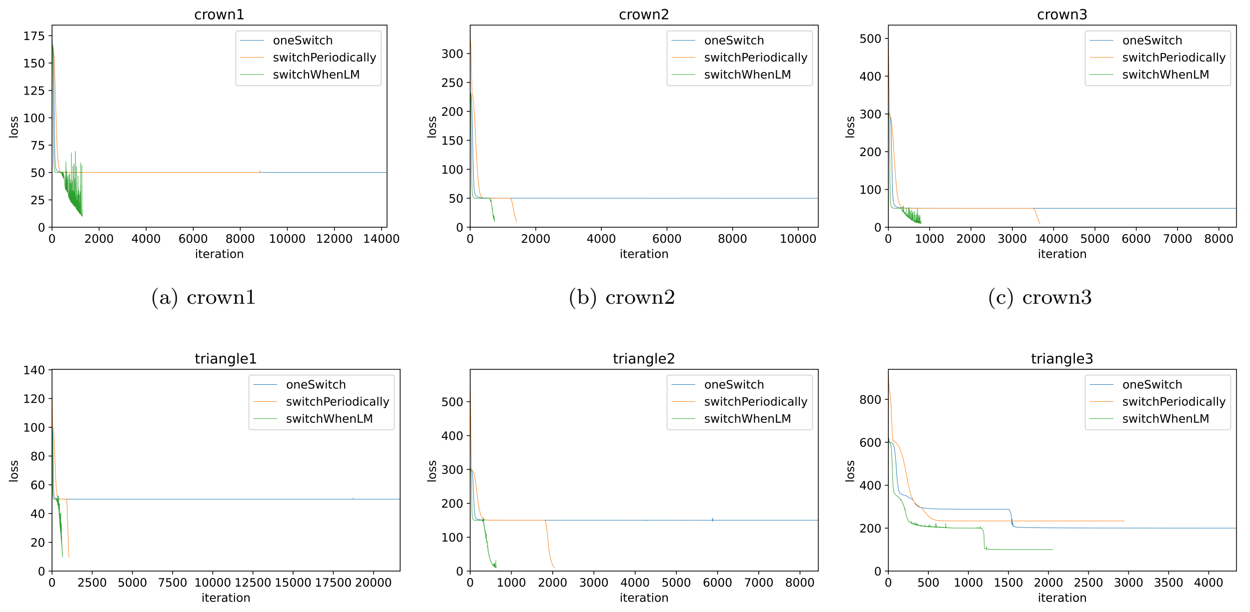


Figure 16: triangle and crown graphs

I have used the same settings as used for the *non-mycielski* graphs, 7, to generate the results visible in 5.3. A few things stand out when analyzing these graphs. The *oneSwitch* method is not able to find a feasible coloring for any of these graphs. The *switchPeriodically* method is able to find solutions for the crown2, crown3, triangle1, and triangle3 graphs but is unable to find solutions for the crown3 and triangle3 graphs. The *switchWhenLM* method is able to find solutions for all but the triangle3 graph. Although the *switchPeriodically* and *switchWhenLM* methods are able to solve some methods, the algorithm is stuck at the same solution value for a lot of iterations.



(a) crown1 (b) crown2 (c) crown3

(d) triangle1 (e) triangle2 (f) triangle3

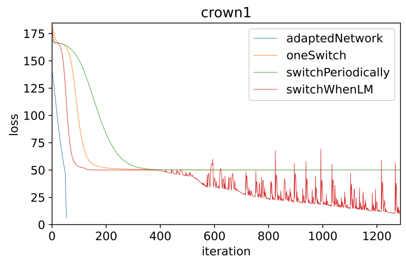
Figure 17: loss against iteration for crown and triangle graphs

For the *oneSwitch* the final coloring is the following:

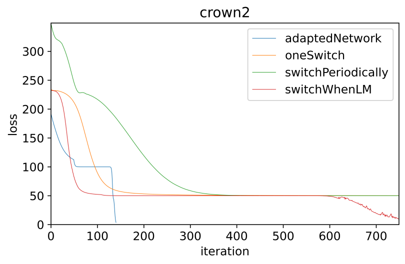
$$\begin{pmatrix} 0 & 0.5 & 0.5 \\ 0 & 0.5 & 0.5 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \tag{23}$$

Looking at the final coloring, it is clear how the coloring should change: if both the first and the second nodes changed their coloring of the second and third color a bit, the loss would be less. However, the algorithm cannot get out of this state. The fact that the neural network has only one layer and thus is too simplistic could be a reason for such inferior performance.

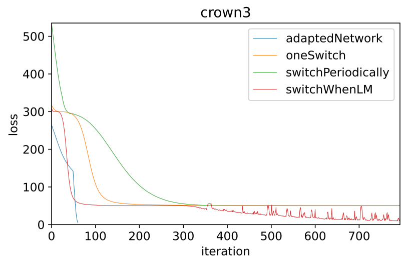
In order to solve this problem, I have tried to add an extra set of learnable parameters to the algorithm. This learnable parameter matrix **W2** is of the size (N $\times$*targetNumber*) and is added to the algorithm before applying the last softmax function. The output of the network before the softmax function is pointwise multiplied with this weight matrix. The weight matrix **W2** is a learnable parameter at the same moments as when the feature matrix is a learnable parameter. The results, using the *switchPeriodically* method with lr =0.2, *mpb* =1, *ivs*=4, *leakyReLU*=0.2 and #*attention heads*=1, on the crown and triangle graphs for this adapted model are presented in Figure 18. I have chosen to use the *switchPeriodically* method and hyperparameters based on some preliminary results. Here it can clearly be observed that the adapted algorithm is much better at coloring these simple graphs. Only for the crown2 graph, the loss has a plateau at some point, but the adapted algorithm is able to navigate its way out of this plateau quickly.



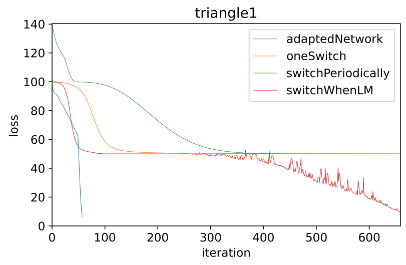
(a) crown1



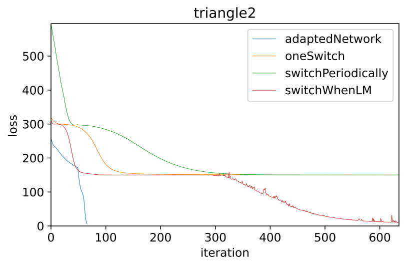
(b) crown2



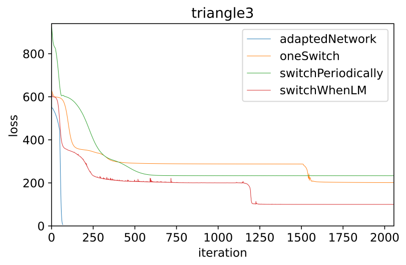
(c) crown3



(d) triangle1



(e) triangle2



(f) triangle3

Figure 18: loss against iteration for crown and triangle graphs

The results of the adapted algorithm are presented in Table 12. In this table, the *adapted* method is compared against the best performing method *oneSwitch* and the method where training on the features is not allowed. The adapted model clearly outperforms the *oneSwitch* and *onlyAW* methods. What is surprising though, is that the model is not able to solve the myciel6 and myciel7 graphs. The algorithm seems to be stuck in a local minimum and not able to move out of this local minimum. For most graphs where no feasible solution is found this seems to be the case. Looking at the loss of these graphs, where no feasible solutions are found, we can observe that the loss for all, but the queen13_13 graphs, is 100$\times$ the number of conflicting edges. This can only be the case when two adjacent nodes in the graph are assigned fully the same color. Now, if we look at the conflicting edges, after a first-order neighborhood color correction, we see that, for most graphs, the number of conflicting edges remains the same, this tells us that the nodes, which are assigned the wrong color, can not easily be assigned a different color without causing a new conflicting edge. This in term proves that indeed a local minimum is found since changing a color of a node will not immediately result in a lower loss. Contrary to the *oneSwitch* and *onlyAW* method, the *adapted* method is able to color a graph in such a way that almost no 'easy' color changes, color changes using a first-order neighborhood color correction, can be made. This seems to indicate that the *adapted* method is a (lot) more powerful than the other methods. Besides that the gap between *ce* and *cn* has substantially decreased, the overall number of *cn* has also decreased a lot. This indicates that the *adapted* method not only resolves these 'easy' color changes but also seems to be able to resolve some other color changes.

	graph properties				*oneSwitch*				*adapted*				*onlyAW*			
graphname	nodes	edges	chi	k	loss	ce	cn	cc	loss	ce	cn	cc	loss	ce	cn	cc
1_insertions_4	67	232	5	5	**0 (283s)**	0	0	**5**	**0 (26s)**	-	-	**5**	100	1	0	**5**
2_insertions_4	149	541	5	5	101	1	0	**5**	**0 (65s)**	-	-	**5**	311	3	2	6
anna	138	493	11	11	1276	30	3	13	100	1	1	12	2230	57	4	13
david	87	406	11	11	2115	74	0	**11**	200	2	2	13	2446	146	1	12
games120	120	638	9	9	2696	66	0	**9**	**0 (155s)**	-		**9**	4787	169	3	10
homer	561	1628	13	13	5187	146	9	15	400	4	4	16	10465	300	12	17
huck	74	301	11	11	1511	82	0	**11**	**0 (89s)**	-	-	**11**	1938	125	0	11
jean	80	256	10	10	1107	45	1	11	**0 (117s)**	-	-	**10**	1796	86	0	10
miles250	128	389	8	8	1981	57	3	10	100	1	1	9	3367	137	3	10
mug88_1	88	146	4	4	267	6	0	**4**	**0 (13.5s)**	-	-	**4**	1136	17	1	5
myciel2	5	5	3	3	**0 (0.83s)**	0	0	**3**	**0 (0.34s)**	-	-	**3**	**0 (0.76s)**	-	-	**3**
myciel3	11	20	4	4	**0 (5.5s)**	0	0	**4**	**0 (1.5s)**	-	-	**4**	**0 (6.8s)**	-	-	**4**
myciel4	23	71	5	5	**0 (696s)**	0	0	**5**	**0 (12s)**	-	-	**5**	**0 (29s)**	-	-	**5**
myciel5	47	236	6	6	100	1	0	**6**	**0 (46s)**	-	-	**6**	**0 (125s)**	-	-	**6**
myciel6	95	755	7	7	12.3	0	0	**7**	200	2	2	8	**0 (676s)**	-	-	**7**
myciel7	191	2360	8	8	710	3	1	9	200	2	2	9	1206	10	7	10
queen5_5	25	160	5	5	1232	12	11	8	600	6	6	7	2099	51	14	8
queen6_6	36	290	7	9	2353	89	3	11	0 (206s)	-	-	9	2402	75	3	11
queen7_7	49	476	7	10	3534	128	9	12	300	3	3	11	3447	136	5	12
queen8_8	64	728	9	13	4669	200	3	14	0 (340s)	0	0	13	4970	310	23	17
queen8_12	96	1368	12	14	8703	331	25	17	700	7	4	15	9469	153	3	15
queen9_9	81	1056	10	12	7471	234	26	16	800	8	6	14	7895	356	23	15
queen11_11	121	1980	11	14	12574	442	57	19	1300	13	11	18	13005	557	42	20
queen13_13	169	3328	13	17	19586	549	44	22	3958	39	23	20	19469	733	49	22
sum	2505	17863	194	214	77185.3	2496	195	247	8858	88	65	236	92538	3422	195	254

Table 12: *adapted* method compared with *oneSwitch* and *onlyAW*. For a given k, I report the results of the algorithm. I report the loss value at the moment the time limit is reached, if the loss value is zero, the algorithm has found a feasible k-coloring, and the time it took to find this k-coloring is reported between brackets. For the graph method combinations where no feasible solutions are found I also report, the number of conflicting edges after rounding, ce, the number of conflicting edges after first-order neighborhood color correction, cn, and the chromatic number after the whole color correction algorithm, $k - cc$. The chromatic number χ is obtained from [Gua15].

In Table 13, the *adapted* method is compared against the methods of the authors of [LLM+22], [SBZK22] and [LPAL19]. The *adapted* method performs worse than the [LLM+22] and [SBZK22] methods. For the graphs in common with [LLM+22], where the [LLM+22] method finds a feasible coloring, the sum of colors used in the *adapted* method is 107 whilst the sum for the [LLM+22] method is 95. For the graphs in common with [SBZK22] the sum of the colors used in the *adapted* method is 143 whilst the sum of the [SBZK22] method is 123. However, the sum of the colors used in the *adapted* method is equal to the sum of the colors used in the [LPAL19] method, for the graphs they have in common, 206 vs 206.

	graph properties				*adapted*				[SBZK22]			[LLM+22]		[LPAL19]
graphname	nodes	edges	chi	k	loss	ce	cn	k-cc	k	ce	X_greedy	k	ce	X_greedy
1_insertions_4	67	232	5	5	**0**	-	-	**5**	-	-	-	-	-	**5**
2_insertions_4	149	541	5	5	**0**	-	-	**5**	-	-	-	4	1	**5**
anna	138	493	11	11	100	1	1	12	11	0	**11**	11	**0**	12
david	87	406	11	11	200	2	2	13	-	-	-	11	**0**	12
games120	120	638	9	9	**0**	-		**9**	-	-	-	9	**0**	-
homer	561	1628	13	13	400	4	4	16	-	-	-	13	**0**	15
huck	74	301	11	11	**0**	-	-	**11**	-	-	-	11	**0**	**11**
jean	80	256	10	10	**0**	-	-	**10**	10	0	**10**	10	**0**	**10**
miles250	128	389	8	8	100	1	1	9	-	-	-	-	-	-
mug88_1	88	146	4	4	**0**	-	-	**4**	-	-	-	-	-	**4**
myciel2	5	5	3	3	**0**	-	-	**3**	-	-	-	-	-	-
myciel3	11	20	4	4	**0**	-	-	**4**	-	-	-	-	-	-
myciel4	23	71	5	5	**0**	-	-	**5**	-	-	-	-	-	-
myciel5	47	236	6	6	**0**	-	-	**6**	6	0	**6**	6	**0**	**6**
myciel6	95	755	7	7	200	2	2	8	7	0	**7**	7	**0**	**7**
myciel7	191	2360	8	8	200	2	2	9	-	-	-	-	-	**8**
queen5_5	25	160	5	5	600	6	6	7	5	0	**5**	5	**0**	8
queen6_6	36	290	7	9	**0**	-	-	9	7	1	8	-	-	11
queen7_7	49	476	7	10	300	3	3	11	7	4	9	7	9	10
queen8_8	64	728	9	13	**0**	0	0	13	9	2	10	8	2	13
queen8_12	96	1368	12	14	700	7	4	15	12	2	13	12	**0**	15
queen9_9	81	1056	10	12	800	8	6	14	10	3	12	9	6	16
queen11_11	121	1980	11	14	1300	13	11	18	11	19	15	11	21	17
queen13_13	169	3328	13	17	3958	39	23	20	13	27	17	13	33	21
sum	2505	17863	194	214	8858	88	65	236	108	58	123	147	72	206

Table 13: Adapted design in comparison with other neural network approaches for the k-coloring problem. I compare the *adapted* method with three neural network approaches for the GCP. For the [SBZK22] approach, I report their target k, the number of conflicting edges, and the chromatic number after their color correction algorithm, X_{greedy}. For [LLM+22] approach, I report their target k, and their number of conflicting edges for this k and between brackets the time it took to find these solutions. For the [LPAL19] approach, I report their chromatic number after their color correction algorithm, X_{greedy}.

5.4 Performance in combination with heuristic/relaxation

In Table 14 the results of the algorithm in combination with a hotstart are presented. I set the time limit for the TABUCOL algorithm to 100 seconds. Next, I have selected the graphs for which TABUCOL did not find the optimal solutions within 100 seconds, since these are the graphs for which a hotstart in combination with TABUCOL can be tried. These graphs are: homer, queen8_8, queen9_9, queen11_11, queen13_13.

If we look at the results in Table 14, we can observe a few things. First, we can see that the algorithm is not able to turn the infeasible solutions of the heuristics and relaxation into feasible solutions, except for the *adapted* method in combination with the starting point obtained via the relaxation. Second, we can observe that *oneSwitch* method is able to improve on 10 out of the 15 starting points, the *switchPeriodically* method on 9 out of the 15 starting points, *switchWhenLM* method on 10 out of the 15 starting points, and the *adapted* method on 14 out of the 15 starting points. All methods improve on all starting points of the relaxation as expected. More interestingly, all methods are also able to improve on some of the TABUCOL and RLF starting points. The *adapted* algorithm is particularly successful as it decreases almost all losses quite substantially. Another observation worth pointing out is that when using the relaxation as the initialization the losses of the algorithm are much lower than when using a random initialization, as can be seen in 15. Here, only the graph-method combination queen13_13-*adapted* method has a higher loss when using the relaxation as a starting point.

	TABUCOL		RLF		relaxation	
homer	k=11		k=12		k=13	
queen8_8	k =9		k=11		k=13	
queen9_9	k=10		k=12		k=12	
queen11_11	k=13		k=14		k=14	
queen13_13	k=15		k=17		k=17	
			oneSwitch			
	TABUCOL		RLF		relaxation	
graphname	begin	end	begin	end	begin	end
homer	900	900	100	100	31150	4276
queen8_8	700	536	300	203	7262	2666
queen9_9	1300	1000	500	300	9669	4438
queen11_11	600	600	1700	908	16605	8005
queen13_13	1000	1000	1300	1300	19508	13640
			switchPeriodically			
	TABUCOL		RLF		relaxation	
graphname	begin	end	begin	end	begin	end
homer	900	900	100	100	31150	6214
queen8_8	700	600	300	300	7262	2065
queen9_9	1300	1200	500	300	9669	3992
queen11_11	600	600	1700	1220	16605	9101
queen13_13	1000	1000	1300	1300	19508	16707
			switchWhenLM			
	TABUCOL		RLF		relaxation	
graphname	begin	end	begin	end	begin	end
homer	900	900	100	100	31150	7124
queen8_8	700	600	300	203	7262	2510
queen9_9	1300	1000	500	300	9669	4300
queen11_11	600	600	1700	1020	16605	11509
queen13_13	1000	1000	1300	1300	19508	17798
			adapted			
	TABUCOL		RLF		relaxation	
graphname	begin	end	begin	end	begin	end
homer	1000	800	100	100	31150	300
queen8_8	500	300	300	100	7262	0 (911s)
queen9_9	900	600	500	200	9669	500
queen11_11	700	300	1700	500	16605	1200
queen13_13	1300	700	1300	693	19508	4552

Table 14: Performance algorithm using a heuristic/relaxation as initialization. In this table, I report the target number, k, for the combination of the initialization method and graph. Next, I report the loss of the algorithm, when the time limit was reached, for each of the three methods in combination with a hotstart using two heuristics, TABUCOL [HdW87] and recursive largest first [Lei79] and one relaxation [MDZ08]

	oneSwitch		*switchPeriodically*		*switchWhenLM*		*adapted*	
graph	relaxation	random	relaxation	random	relaxation	random	relaxation	random
homer	4276	5187	6214	8176	7124	5922	300	400
queen8_8	2666	4669	2065	4636	2510	4885	0	0
queen9_9	4438	7471	3992	7606	4300	7450	500	800
queen11_11	8005	12574	9101	13868	11509	13202	1200	1300
queen13_13	13640	19586	16707	24533	17798	19657	4552	3985

Table 15: Comparison between initializing the starting position randomly (random) and initializing the starting position using the LP-relaxation of [MDZ08]

In Table 16, the results of using the algorithm as input for TABUCOL are presented. The loss of the method at the time limit is reported. Following this, the number of conflicting edges after a first-order neighborhood color correction *cn* is reported, and next the time it took TABUCOL to eliminate these conflicting edges and find a feasible solution is reported. Also, the chromatic number when using TABUCOL to solve the graph is reported with the time it took to solve the graph between brackets. I gave TABUCOL 20 minutes to solve a graph for a given color k, starting at χ, and if it not succeeded increased k by one and let it try again, until TABUCOL was able to find a feasible solution.

If we look at the results of the combination of TABUCOL and the methods, Table 16, we can observe that TABUCOL is able to solve every infeasible solution provided by the methods proposed in this thesis using very little time. The TABUCOL method alone, the last column of Table 16, however, is also very quick in solving the graphs and also solves graphs for a lower chromatic number than the k used when in combination with the methods of this thesis. From this, we can conclude that although TABUCOL is able to easily use and resolve the infeasible outputs of the methods proposed in this thesis, it is more beneficial to just use TABUCOL. In Table 10, we have already seen that the output of the algorithm can be used to find feasible colorings and in Table 16, we have seen that TABUCOL can also find feasible colorings using the output of the algorithm. Therefore we can conclude that the output of the algorithm can successfully be used to find feasible colorings.

	graph properties				*oneSwitch*			*switchPeriodically*			*switchWhenLM*			*adapted*			*onlyAW*			TABUCOL
graphname	nodes	edges	chi	k	loss	cn	TC	loss	cn	TC	loss	cn	TC	loss	cn	TC	loss	cn	TC	
1_insertions_4	67	232	5	5	0 (283s)	0	-	0 (153s)	-	-	0 (157s)		-	0 (26s)		-	100	0	0s	5 (0.14s)
2_insertions_4	149	541	5	5	101	0	0s	0 (385s)	-	-	0 (496s)		-	0 (65s)		-	311	2	0.19s	5 (1.78s)
anna	138	493	11	11	1276	3	2.0s	1376	4	4.8s	1064	3	4.4s	100	1	1.68s	2230	4	3.8s	11 (2.77s)
david	87	406	11	11	2115	0	0s	1616	3	1.03s	2413	1	0.93s	200	2	3.1s	2446	1	0.49s	11 (5.08s)
games120	120	638	9	9	2696	0	0s	2973	1	0.54s	2926	0	0s	0 (155s)		-	4787	3	0.31s	9 (0.64s)
homer	561	1628	13	13	5187	9	212s	8176	9	466s	5922	7	160s	400	4	160s	10465	12	328s	13 (304s)
huck	74	301	11	11	1511	0	0s	1759	1	0.20s	1502	0	0s	0 (89s)	-	-	1938	0	0s	11 (0.173s)
jean	80	256	10	10	1107	1	0.83s	1003	0	0s	875	1	6.7s	0 (117s)	-	-	1796	0	0s	10 (0.62s)
miles250	128	389	8	8	1981	3	1.2s	2151	7	17s	2110	3	1.6s	100	1	2.6s	3367	3	0.84s	8 (3.86s)
mug88_1	88	146	4	4	267	0	0s	383	0	0s	0 (873s)		-	0 (13.5s)	-	-	1136	1	0.04s	4 (0.12s)
myciel2	5	5	3	3	0 (0.83s)	0	0	0 (2.2s)	0	0	0 (0.94s)		-	0 (0.34s)	-	-	0 (0.76s)		-	3 (0.01s)
myciel3	11	20	4	4	0 (5.5s)	0	0	0 (11s)	0	0	0 (5.34s)		-	0 (1.5s)	-	-	0 (6.8s)		-	4 (0.01s)
myciel4	23	71	5	5	0 (696s)	0	0	0 (45s)	0	0	0 (27s)		-	0 (12s)	-	-	0 (29s)		-	5 (0.01s)
myciel5	47	236	6	6	100	0	0s	100	1	0.12s	0 (241s)		-	0 (46s)	-	-	0 (125s)		-	6 (0.17s)
myciel6	95	755	7	7	12.3	0	0s	0 (1004s)	-	-	0 (1083s)		-	200	2	0.49s	0 (676s)		-	7 (1.11s)
myciel7	191	2360	8	8	710	1	1.0s	2274	1	1.03s	588	0	0s	200	2	3.0s	1206	7	1.3s	8 (24.1s)
queen5_5	25	160	5	5	1232	11	0.06s	1734	18	0.10s	1004	10	0.35s	600	6	0.08s	2099	14	0.19s	5 (0.10s)
queen6_6	36	290	7	9	2353	3	0.15s	2418	6	0.23s	2175	5	0.19s	0 (206s)	-	-	2402	3	0.16s	7 (17.3s)
queen7_7	49	476	7	10	3534	9	0.31s	3092	11	0.57s	4012	7	0.46s	300	3	0.75s	3447	5	0.57s	7 (53.8s)
queen8_8	64	728	9	13	4669	3	0.55s	4636	6	0.63s	4885	3	0.63s	0 (340s)	0		4970	23	3.7s	9 (173s)
queen8_12	96	1368	12	14	8703	25	2.8s	9187	39	4.7s	8578	23	5.2s	700	4		9469	3	0.57s	12 (62.8s)
queen9_9	81	1056	10	12	7471	26	4.6s	7606	17	2.5s	7450	32	5s	800	6	3.2s	7895	23	2.6s	10 (1133s)
queen11_11	121	1980	11	14	12574	57	21s	13868	66	26s	13202	52	28s	1300	11	11s	13005	42	15s	13 (265s)
queen13_13	169	3328	13	17	19586	44	21s	24533	35	25s	19657	35	25s	3958	23	48s	19469	49	26s	16 (85.2s)
sum	2505	17863	194	214	77185.3	195		88885	225		78363	182		8858	65		92538	195		199 (2.134s)

Table 16: Performance of the color correction using TABUCOL. In this table I report the loss values at the time limit, if the loss value is 0 a feasible solution is found and the time it took to find this value is reported between brackets. Next, I report the number of conflicting edges after the first-order neighborhood color correction, *cn*, and I report the time it took TABUCOL to eliminate these conflicting edges and find a feasible solution, TC. Lastly, I report the results of running the TABUCOL algorithm alone where the number is k and the time it took to find the k-coloring is reported between brackets.

6 Conclusion and further recommendations

In this thesis, I have presented a new algorithm inspired by [SBZK22] and [VCC$^+$17]. I combined these two papers into one design and proposed to not only train on the normal neural network parameters but also on the input features. The goal of this thesis was to answer the following three research questions:

1. How does the combined design of [SBZK22] and [VCC$^+$17] perform for the k-coloring problem?
2. Does training on the features improve the performance of a GNN for the k-coloring problem
3. How can heuristics and relaxation be used to aid the algorithm and vice-versa?

To answer the first question, I have tested the methods *oneSwitch*, *switchPeriodically* and *switchWhenLM* on a range of different graph types and found that the methods performed well on the *mycielski* graphs, where optimal colorings for all *mycielski* graphs were found. On the *non-mycielski* graphs, however, the methods *oneSwitch*, *switchPeriodically* and *switchWhenLM* did not perform well, here the methods could not match the performance of the algorithms proposed by the authors of [SBZK22], [LLM$^+$22] and [LPAL19]. The adapted model, which resulted from an analysis as to why the algorithm did perform poorly on the *non-mycielski* graphs, performed a lot better than the methods *oneSwitch*, *switchPeriodically* and *switchWhenLM*, and managed to find some feasible solutions for *non-mycielski* graphs. The *adapted* method matched the performance of the algorithm of [LPAL19], but still performed worse than the algorithms of [SBZK22], [LLM$^+$22].

To answer the second research question, I compared the performance of the *oneSwitch*, *switchPeriodically* and *switchWhenLM* methods with a method where training on the features was not allowed, *onlyAW*. For both the *mycielski* and *non-mycielski* graphs training on features, when used properly, seemed to substantially improve the performance of the algorithm, for the *mycielski* graphs the feasible solutions were found faster, and for the *non-mycielski* graphs the loss at the time limit and the number of conflicting edges after rounding were lower.

In order to answer the last research question, I have used two heuristics RLF [Lei79], TABUCOL [HdW87] and one linear programming [MDZ08] as an initialization for the algorithm. All the starting points provided to the algorithm could be used and all methods, *oneSwitch*, *switchPeriodically*, *switchWhenLM* and *adapted* were able to improve on some of the starting points. Especially the *adapted* method was able to make some substantial improvements. Using the relaxation as a starting point of the algorithm improved the final loss for 19 out of 20 graph-method combinations. How much of this improvement is due to the lower loss of the starting point would require further experimentation. Next, I have also combined the output of the algorithm with TABUCOL and shown that TABUCOL was able to find feasible k-colorings for the outputs of the algorithm. However, TABUCOL alone was also able to find such coloring. Therefore the usefulness of combining the proposed algorithm with a heuristic would require further experimentation. I have also designed a color correction algorithm inspired by the RLF algorithm which was able to turn the infeasible colorings into feasible colorings.

Limitations The first set of limitations has to do with how the algorithm is implemented and trained. I trained the hyperparameters of the algorithm on a very small set of graphs and also tested the algorithm on a small set of graphs. This in turn makes it difficult to assess the true performance of the proposed algorithm for the k-coloring problem. Next, I have chosen to train a limited set of hyperparameters, the weight decay of the optimizer, the rules when to make a switch of trainable parameters, and the rules for the method to move out of a local minimum, were all not a hyperparameter but all greatly impact the performance of the algorithm. Another important factor that impacted the performance of the proposed algorithm is the computational power of the computer used to run the proposed algorithm. The second set of limitations has to do with how the algorithm is designed. Since the proposed algorithm has only one layer, in the case of the *oneSwitch, switchPeriodically* and *switchWhenLM* methods, or two layers, in the case of the *adapted* method, the algorithm is not able to gather information farther than nodes respectively one or two steps away. Especially in complex structures this can be a disadvantage. The loss of the algorithm often got stuck in a local minimum and getting out of this local minimum took often a lot of time, which limited the algorithm quite heavily. Next, the equation used to determine the loss during the execution of the algorithm only looks at whether or not two adjacent nodes are assigned the same color, but it does not look at the colors of nodes in the second order neighborhood of a node. Therefore there is no factor that pushes nodes that are not adjacent to each other to be assigned the same color. Lastly, the method used for the rounding of the outputs of the algorithm did not take the objective into account when rounding and therefore resulted in a probably far from optimal rounding.

Recommendations for further research To investigate whether training on the features indeed improves the performance for a NN for the k-coloring problem, different GNN designs, preferably designs that have known success for the GCP, could be used to see if it also improves the performance of these designs for the k-coloring problem. Following this, it would be interesting to see if a loss function can be designed that pushes adjacent nodes to be colored as differently as possible and also pushes second order neighbors to be assigned as similarly as possible. Next, it could be worth examining how much the performance of the algorithm improves when a rounding heuristic is used that takes the objective into account. Lastly, using the algorithm as an initialization for TABUCOL on graphs for which TABUCOL is not able to find feasible solutions could be tried in order to see if the output of the proposed algorithm provides proper starting points for TABUCOL.

References

[BLP21] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, April 2021.

[Bre79] Daniel Brelaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, apr 1979.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[DHD+19] Chi Thang Duong, Thanh Dat Hoang, Ha The Hien Dang, Quoc Viet Hung Nguyen, and Karl Aberer. On node features for graph neural networks, 2019.

[GK11] Erkam Guresen and Gulgun Kayakutlu. Definition of artificial neural networks with comparison to other networks. *Procedia Computer Science*, 3:426–433, 2011.

[Gua15] Stefano Gualandi. Graph coloring benchmarks, 2015.

[HdW87] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, December 1987.

[HLS09] P. Hansen, M. Labbé, and D. Schindl. Set covering and packing formulations of graph coloring: Algorithms and first polyhedral results. *Discrete Optimization*, 6(2):135–147, May 2009.

[Kar72] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer US, Boston, MA, 1972.

[Lei79] F.T. Leighton. A graph coloring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, 84(6):489, November 1979.

[LLM+22] Wei Li, Ruxuan Li, Yuzhe Ma, Siu On Chan, David Pan, and Bei Yu. Rethinking graph neural networks for the graph coloring problem, 2022.

[LPAL19] Henrique Lemos, Marcelo O. R. Prates, Pedro H. C. Avelar, and Luís C. Lamb. Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. *CoRR*, abs/1903.04598, 2019.

[MDZ08] Isabel Méndez-Díaz and Paula Zabala. A cutting plane algorithm for graph coloring. *Discrete Applied Mathematics*, 156(2):159–179, January 2008.

[MHH08] Mohammad Malkawi, Mohammad Al-Haj Hassan, and Osama Al-Haj Hassan. A new exam scheduling algorithm using graph coloring. *International Arab Journal of Information Technology (IAJIT)*, 5(1), 2008.

[MT10] Enrico Malaguti and Paolo Toth. A survey on vertex coloring problems. *International Transactions in Operational Research*, 17(1):1–34, January 2010.

[PL96] Taehoon Park and Chae Y. Lee. APPLICATION OF THE GRAPH COLORING ALGORITHM TO THE FREQUENCY ASSIGNMENT PROBLEM. *Journal of the Operations Research Society of Japan*, 39(2):258–265, 1996.

[SBK21] Martin J. A. Schuetz, J. Kyle Brubaker, and Helmut G. Katzgraber. Combinatorial optimization with physics-inspired graph neural networks. 2021.

[SBZK22] Martin J. A. Schuetz, J. Kyle Brubaker, Zhihuai Zhu, and Helmut G. Katzgraber. Graph coloring with physics-inspired graph neural networks, 2022.

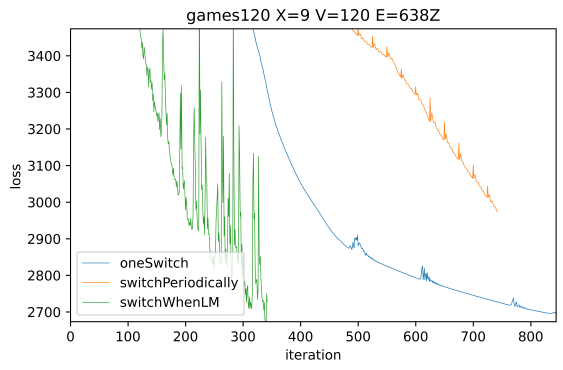
[VCC+17] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2017.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

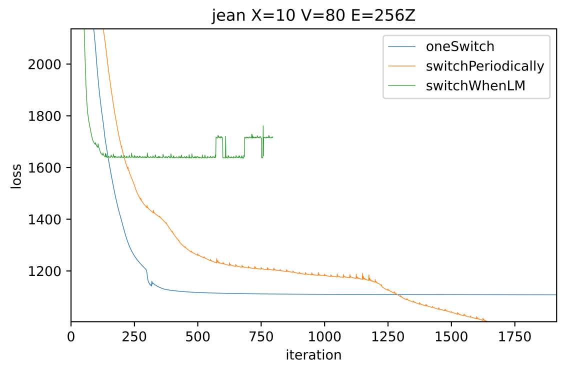
[ZCH+18] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications, 2018.

Appendices

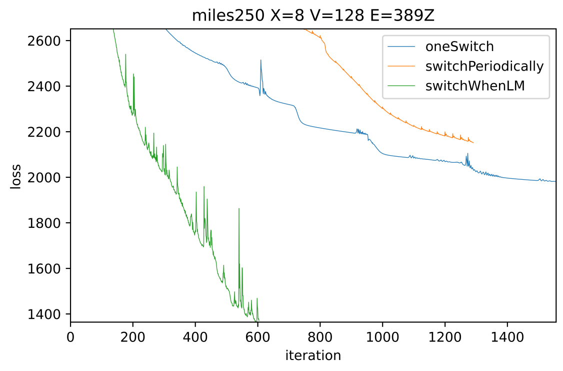
A Figures



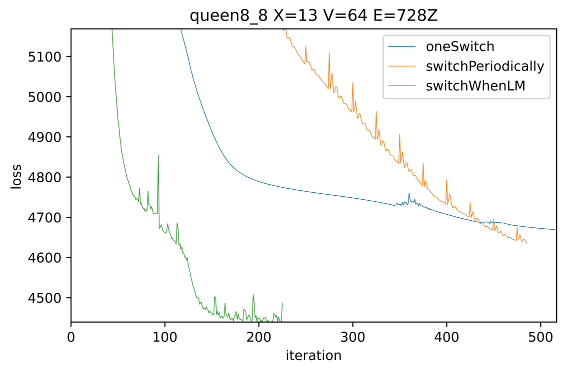
(a) games120



(b) jean



(c) miles250



(d) queen8_8

Figure 19: Loss value of best performing hyperparameters, where X = the inputted k

B Pseudocodes

Algorithm 2 Adam

```
  **Require** $\alpha$: stepsize
  **Require** $\beta_1$,$\beta_2 \in [0,1)$: Exponential decay rates for the moment estimates
  **Require** $f(\theta)$: Stochastic objective function with parameters $\theta$
  **Require** $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize the 1$^{st}$ moment vector)
  $v_0 \leftarrow 0$ (Initialize the 2$^{nd}$ moment vector)
  t $\leftarrow$ 0 (Initialize timestep)
  **while** ($\theta_t$ not converged) **do**
      t $\leftarrow$ t + 1
      g$_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
      m$_t \leftarrow \beta_1 m_{t-1}(1-\beta_1)g_t$ (Update biased first moment estimate)
      v$_t \leftarrow \beta_2 v_{t-1}(1-\beta_2)g_t^2$ (Update biased second raw moment estimate)
      $\hat{m}_t \leftarrow m_t/(1-\beta_1^t)$ (Compute bias-corrected first moment estimate)
      $\hat{v}_t \leftarrow v_t/(1-\beta_2^t)$ (Compute bias-corrected second raw moment estimate)
      $\theta_t \leftarrow \theta_{t-1} - \alpha\hat{m}_t/(\sqrt{\hat{v}_t}+\epsilon)$ (Update parameters)
  **end while**
  **Require** $\theta_t$ (Resulting parameters)
```

Algorithm 3 Recursive Largest First

```
  colornumber = 0
  **while** (number of uncolored vertices > 0) **do**
      determine a vertex x of maximal degree in G
      colornumber = colornumber + 1
      F(x) = colornumber
      NN = set of non-neighbors of x
      **while** (cardinality of NN > 0) **do**
          maxcn = 1
          ydegree = 1
          **for** (every vertex z in NN) **do**
              cn = number of common neigbors of z and x
              **if** (cn > maxcn **or** (cn == maxcn **and** degree(z) < ydegree)) **then**
                  y = z
                  ydegree = degree(y)
                  maxcn = cn
              **end if**
          **end for**
          **if** (maxcn == 0) **then**
              y = vertex of maximal degree in NN
          **end if**
          F(y) = colornumber
          contract y into x
          update the set NN of non-neighbours of x
      **end while**
      G = G x;
  **end while**
```

Algorithm 4 DSATUR

```
UV = set of uncolored vertices
saturation degree vertex (Sat)= the number of different colors to which it is adjacent (colored vertices).
Degree of vertex (Deg) = degree in the uncolored subgraph

Arrange the vertices by decreasing order of degrees
color a vertex of maximal degree with color 1
while ( UV is non empty) do
    maxSat = 0
    maxDeg = 0
    for (every vertex z in UV) do
        if (zSaturation > maxSaturation)) then
            maxSat = zSat
            maxDeg = zDeg
            y = z
        else if (zSat == maxSat and zDeg > maxDeg) then
            maxDeg = zDeg
            y = z
        end if
    end for
    F(y) = lowest possible number
    remove y from UV
end while
```

Algorithm 5 TABUCOL

```
k = number of colors
|T|= size of tabu list
rep = number of neighbours in sample
nbmax = maximum number of iterations
Initialization
Generate a random solution s = ($V_1$, ... ,$V_k$)
nbiter:=0
while (f(s) > 0 and nbiter < nbmax) do
    generate rep neighbors $s_i$ of $s$ with move $s \to s_i \notin T$ or $f(s_i) \leq A(f(s))$

    Let $s'$ be the best neighbor generated

    update tabu list $T$ (introduce move $s \to s'$ and remove the oldest tabu move)

    s:=$s'$
    nbiter:=nbiter + 1
end while
if ($f(s) = 0$) then
    feasible coloring of G found, with k colors: $V_1$, ... , $V_k$ are the colors sets
else
    No feasible coloring found
end if
```

C Tables

steps	Derivates
$\frac{\delta L}{\delta L_{i,j}}$	1
$\frac{\delta L_{i,j}}{\delta \dddot{h}_{k,c}}$	$\sum_{N_k}$ $\text{if}_{i=k}\ G * \dddot{h}_{j,c}\ \text{if}_{j=k}\ G * \dddot{h}_{i,c}$
$\frac{\delta \dddot{h}_{k,c}}{\delta \ddot{h}_{k,c}}$	$(\frac{\exp(\ddot{h}_{k,c})}{\sum_{i \in N_k} \exp(\ddot{h}_{i,c}))}) \times (1 - \frac{\exp(\ddot{h}_{k,c})}{\sum_{i \in N_k} \exp(\ddot{h}_{i,c})})$
$\frac{\delta \ddot{h}_{k,c}}{\delta \alpha_{i,j}}$	$\sum_{c \in C} \ddot{h}_{i,c}$
$\frac{\delta \alpha_{i,j}}{\delta \dot{e}_{i,j}}$	$(\frac{\exp(\dot{e}_{i,j})}{\sum_{k \in N_i} \exp(\dot{e}_{k,j})}) \times (1 - \frac{\exp(\dot{e}_{i,j})}{\sum_{k \in N_i} \exp(\dot{e}_{k,j})})$
$\frac{\delta \dot{e}_{i,j}}{\delta e_{i,j}}$	$\text{if}_{e_{i,j}>0}$ 1 $\text{if}_{e_{i,j}<0}$ LRconstant
$\frac{\delta e_{i,j}}{\delta \dot{h}_{k,c}}$	$\sum_{\text{i} \in \text{ N}_k}$ if $_{i \leq k} a_c$ if $_{i>k} a_{2*c}$
$\frac{\delta \dot{h}_{k,c}}{\delta h_{i,j}}$	$\sum_{\forall \text{j}} W_{i,j}$
$\frac{\delta e_{i,j}}{\delta a_c}$	$\sum_{\forall \text{i in V}}$ if $_{\text{c} \leq k} \dot{h}_{i,c}$ if $_{\text{c}>k} \dot{h}_{j,c}$
$\frac{\delta h_{k,c}}{\delta W_{i,j}}$	$\sum_{\forall \text{ i in V}} h_{i,j}$

Table 17: Backpropagation

Table 18: Results hyperparameters *oneSwitch, switchPeriodically, switchWhenLM* and *onlyAW* for *training4graphs* graphs.

mbp	lr	ivs	leakyReLU	nBB	games120	jean	miles250	myciel7	queen8_8	Sum	Sum
oneSwitch					$\chi = 9$	$\chi = 10$	$\chi = 8$	$\chi = 8$	$\chi = 13$	inc myciel	exc myciel
0,01	0,001	4	0,2	1	3736	1001	2745	11955	4630	24067	12112
0,01	0,0031	4	0,2	1	4146	1246	3143	1740	4589	14864	13124
0,01	0,01	4	0,2	1	4632	1434	3225	139	5382	14813	14673
0,01	0,031	4	0,2	1	7189	2613	4877	7430	6425	28534	21103
0,031	0,001	4	0,2	1	3677	1091	2770	17367	4562	29467	12099
0,031	0,0031	4	0,2	1	3540	1208	2840	3477	4623	15688	12211
0,031	0,01	4	0,2	1	4542	1073	3023	388	4920	13946	13558
0,031	0,031	4	0,2	1	7095	1892	4666	449	6959	21061	20612
0,1	0,001	4	0,2	1	3949	1135	4009	26567	5536	41196	14629
0,1	0,0031	4	0,2	1	3294	1341	3473	6245	4872	19225	12980
0,1	0,01	4	0,2	1	4183	1154	3204	2384	4494	15419	13035
0,1	0,031	4	0,2	1	4259	1407	3229	723	5056	14674	13951
0,31	0,001	4	0,2	1	7092	1220	2636	28826	4556	44331	15504
0,31	0,0031	4	0,2	1	6544	1219	2823	21479	4272	36337	14858
0,31	0,01	4	0,2	1	5065	1271	2828	2766	4599	16528	13762
0,31	0,031	4	0,2	1	4510	1081	3109	951	4311	13962	13011
1	0,001	4	0,2	1	3416	1135	2453	6902	4960	18866	11964
1	0,0031	4	0,2	1	2682	1443	2020	471	4334	10949	10478
1	0,01	4	0,2	1	3607	1384	2967	223	4822	13003	12779
1	0,031	4	0,2	1	3811	1299	2917	106	4781	12914	12808
1	0,0031	1	0,2	1	2936	1191	2134	4925	4747	15933	11008
1	0,0031	3	0,2	1	2697	1108	1984	718	4669	11175	10457
1	0,0031	6	0,2	1	2677	1370	2059	715	4509	11331	10616
1	0,0031	9	0,2	1	3667	1433	2849	617	4482	13048	12431
1	0,0031	3	0	1	2696	1108	1980	711	4669	11164	10453
1	0,0031	3	0,1	1	2696	1108	1981	711	4669	11164	10454
1	0,0031	3	0,3	1	2693	1108	1982	704	4668	11154	10451

1	0,0031	3	0,1	2	3903	1127	2333	7120	4599	19082	11962
1	0,0031	3	0,1	3	4426	963	1750	4444	4560	16143	11699
1	0,0031	3	0,1	4	3130	765	1981	2881	4784	13541	10660
1	0,0031	3	0,1	5	3047	792	1782	1256	5126	12003	10748
					games120	jean	miles250	myciel7	queen8_8	Sum	Sum
switchPeriodically					$\chi = 9$	$\chi = 10$	$\chi = 8$	$\chi = 8$	$\chi = 13$	inc myciel	exc myciel
0,01	0,001	4	0,2	1	4035	1297	2771	10	4782	12894	12884
0,01	0,0031	4	0,2	1	4439	1370	2825	5	4586	13225	13220
0,01	0,01	4	0,2	1	5536	1530	3467	1	5984	16517	16516
0,01	0,031	4	0,2	1	7249	5728	6185	500	11823	31484	30984
0,031	0,001	4	0,2	1	3682	1144	2766	8	4769	12368	12361
0,031	0,0031	4	0,2	1	4384	1412	2720	3	5186	13706	13703
0,031	0,01	4	0,2	1	4791	1252	3100	9	5072	14224	14215
0,031	0,031	4	0,2	1	7413	2389	4748	300	8660	23511	23211
0,1	0,001	4	0,2	1	3483	1099	2690	1154	4754	13179	12026
0,1	0,0031	4	0,2	1	3326	1111	2606	100	4378	11520	11420
0,1	0,01	4	0,2	1	3766	1193	2758	9	4919	12644	12635
0,1	0,031	4	0,2	1	4848	1436	3273	100	5489	15145	15045
0,31	0,001	4	0,2	1	3659	1102	2712	8376	4721	20570	12194
0,31	0,0031	4	0,2	1	3538	1135	2648	207	4717	12245	12037
0,31	0,01	4	0,2	1	3774	1063	2916	200	4507	12459	12259
0,31	0,031	4	0,2	1	3478	1108	2819	1	4776	12182	12181
1	0,001	4	0,2	1	3358	897	2198	9920	4570	20943	11023
1	0,0031	4	0,2	1	3222	972	2522	102	4427	11245	11143
1	0,01	4	0,2	1	3368	1365	2825	500	4616	12673	12173
1	0,031	4	0,2	1	3743	903	2831	2	5038	12518	12516
1	0,001	1	0,2	1	4764	1087	2697	26336	4947	39831	13494
1	0,001	3	0,2	1	3785	999	2375	13443	4715	25317	11874
1	0,001	6	0,2	1	3154	1241	2093	3528	4749	14765	11237
1	0,001	9	0,2	1	2858	987	2575	397	4388	11205	10808
1	0,001	9	0	1	2858	987	2573	397	4388	11203	10806
1	0,001	9	0,1	1	2857	987	2575	397	4388	11204	10806
1	0,001	9	0,3	1	2856	987	2574	397	4388	11201	10804
1	0,001	9	0,2	2	3671	1406	2760	8538	4391	20766	12228
1	0,001	9	0,2	3	3226	1141	2603	4378	4651	16000	11622
1	0,001	9	0,2	4	3233	982	2330	2175	4760	13479	11304
1	0,001	9	0,2	5	2974	1004	2151	2275	4636	13040	10765
					games120	jean	miles250	myciel7	queen8_8	Sum	Sum
switchWhenLM					$\chi = 9$	$\chi = 10$	$\chi = 8$	$\chi = 8$	$\chi = 13$	inc myciel	exc myciel
0,01	0,001	4	0,2	1	3852	1109	2710	9999	4561	22231	12232
0,01	0,0031	4	0,2	1	3922	1595	2805	1936	4688	14946	13011
0,01	0,01	4	0,2	1	4776	1683	3292	95	4799	14644	14549
0,01	0,031	4	0,2	1	7221	2536	4866	13286	8045	35954	22668
0,031	0,001	4	0,2	1	3469	901	2546	16160	4280	27356	11196
0,031	0,0031	4	0,2	1	3854	1216	3018	3757	4713	16558	12801
0,031	0,01	4	0,2	1	4604	1819	3251	555	5015	15244	14689
0,031	0,031	4	0,2	1	6663	1576	3618	265	7482	19604	19339
0,1	0,001	4	0,2	1	3762	1051	2670	1695	4396	13574	11879
0,1	0,0031	4	0,2	1	3231	1051	2702	6807	4821	18613	11805
0,1	0,01	4	0,2	1	3640	1049	3065	1043	5069	13866	12823
0,1	0,031	4	0,2	1	4104	1257	3330	529	5787	15008	14479
0,31	0,001	4	0,2	1	3524	1050	2838	5934	4848	18194	12259
0,31	0,0031	4	0,2	1	3162	1118	2551	747	4175	11754	11006

0,31	0,01	4	0,2	1	3683	861	2563	2788	4916	14811	12023
0,31	0,031	4	0,2	1	3373	1498	3155	932	5010	13967	13035
1	0,001	4	0,2	1	3166	1006	2651	6146	4714	17683	11537
1	0,0031	4	0,2	1	3316	932	2225	296	4928	11696	11400
1	0,01	4	0,2	1	3666	1076	2672	1	4708	12123	12121
1	0,031	4	0,2	1	3781	1181	877	100	4621	10560	10460
0,31	0,0031	1	0,2	1	4145	892	2070	6002	4230	17339	11336
0,31	0,0031	3	0,2	1	3092	1126	1940	936	4653	11746	10810
0,31	0,0031	6	0,2	1	3598	1227	1710	132	4582	11250	11118
0,31	0,0031	9	0,2	1	3483	1032	2579	7926	4554	19574	11648
0,31	0,0031	6	0	1	3575	1287	1850	127	4418	11257	11129
0,31	0,0031	6	0,1	1	3512	1668	1683	120	4559	11541	11422
0,31	0,0031	6	0,3	1	2673	1636	1364	131	4439	10243	10113
0,31	0,0031	6	0,3	2	3510	1184	2793	2766	4424	14677	11911
0,31	0,0031	6	0,3	3	2996	1088	2214	796	4759	11852	11056
0,31	0,0031	6	0,3	4	4037	720	1397	497	4542	11192	10695
0,31	0,0031	6	0,3	5	3025	970	2275	373	4332	10974	10601
0,31	0,0031	3	0	1	3043	1069	2249	642	4611	11614	10972
0,31	0,0031	3	0,1	1	3247	1078	2123	881	4837	12166	11285
0,31	0,0031	3	0,3	1	2927	876	2110	588	4873	11375	10786
0,31	0,0031	3	0,3	2	3817	1094	2291	5065	4564	16831	11766
0,31	0,0031	3	0,3	3	3212	692	2443	5000	4867	16215	11215
0,31	0,0031	3	0,3	4	3164	1144	2133	1989	4623	13053	11064
0,31	0,0031	3	0,3	5	2815	851	2249	2471	4886	13272	10801
					games120	jean	miles250	myciel7	queen8_8	Sum	Sum
onlyAW					$\chi = 9$	$\chi = 10$	$\chi = 8$	$\chi = 8$	$\chi = 13$	inc myciel	exc myciel
0,01	0,001	4	0,2	1	6837	1714	4095	24277	5418	42340	18064
0,01	0,0031	4	0,2	1	5501	1771	3489	8040	4938	23740	15700
0,01	0,01	4	0,2	1	4911	1381	3397	1844	5061	16594	14750
0,01	0,031	4	0,2	1	7093	3406	4953	5780	6209	27441	21661
0,031	0,001	4	0,2	1	7011	1986	4260	27262	5552	46072	18810
0,031	0,0031	4	0,2	1	5731	1836	3704	14444	5015	30730	16285
0,031	0,01	4	0,2	1	4934	1571	3391	964	4708	15568	14605
0,031	0,031	4	0,2	1	5276	1587	4833	733	7105	19534	18801
0,1	0,001	4	0,2	1	7097	2111	4667	28829	5589	48294	19464
0,1	0,0031	4	0,2	1	6790	1857	3772	24406	5291	42116	17710
0,1	0,01	4	0,2	1	5158	1581	3400	8801	4894	23834	15033
0,1	0,031	4	0,2	1	4728	1796	3361	1534	4907	16327	14793
0,31	0,001	4	0,2	1	7138	2397	4831	29139	5604	49109	19970
0,31	0,0031	4	0,2	1	7063	1787	4198	28463	5574	47085	18622
0,31	0,01	4	0,2	1	6023	1574	3444	16829	5057	32928	16099
0,31	0,031	4	0,2	1	5054	1341	3220	4065	5068	18747	14682
1	0,001	4	0,2	1	7153	2488	4952	29159	5607	49359	20201
1	0,0031	4	0,2	1	7081	1819	4524	28718	5587	47730	19012
1	0,01	4	0,2	1	6689	1665	3355	23158	5354	40220	17062
1	0,031	4	0,2	1	5036	1599	3173	6539	5215	21563	15023
0,031	0,01	1	0,2	1	7020	1995	4078		4863		17957
0,031	0,01	3	0,2	1	5148	1633	3163		5040		14984
0,031	0,01	6	0,2	1	5353	1794	3134		4698		14979
0,031	0,01	9	0,2	1	4932	1898	3267		5963		16060
0,031	0,01	4	0	1	4546	1825	3243		4965		14578

0,031	0,01	4	0,1	1	4940	1348	3447		4630		14365
0,031	0,01	4	0,3	1	4923	1548	3445		5109		15025
0,031	0,01	4	0,1	2	6495	1781	4358		5534		18168
0,031	0,01	4	0,1	3	6066	1886	4776		5567		18295
0,031	0,01	4	0,1	4	7006	2086	4650		5594		19337
0,031	0,01	4	0,1	5	7058	2246	4840		5651		19795

Table 19: results hyperparameters *oneSwitch, switchPeriodically, switchWhenLM* and *onlyAW* for *mycielski* graphs.

oneSwitch	myciel7	χ=8				
mbp	lr	ivs	leakyrelu	nbb	loss	time
0,01	0,001	4	0,2	1	11955	1200
0,01	0,0031	4	0,2	1	1740	1200
0,01	0,01	4	0,2	1	139	1200
0,01	0,031	4	0,2	1	7430	1200
0,031	0,001	4	0,2	1	17367	1200
0,031	0,0031	4	0,2	1	3477	1200
0,031	0,01	4	0,2	1	388	1200
0,031	0,031	4	0,2	1	449	1200
0,1	0,001	4	0,2	1	26567	1200
0,1	0,0031	4	0,2	1	6245	1200
0,1	0,01	4	0,2	1	2384	1200
0,1	0,031	4	0,2	1	723	1200
0,31	0,001	4	0,2	1	28826	1200
0,31	0,0031	4	0,2	1	21479	1200
0,31	0,01	4	0,2	1	2766	1200
0,31	0,031	4	0,2	1	951	1200
1	0,001	4	0,2	1	6902	1200
1	0,0031	4	0,2	1	471	1200
1	0,01	4	0,2	1	223	1200
1	0,031	4	0,2	1	106	1200
1	0,0031	1	0,2	1	4878	1200
1	0,0031	3	0,2	1	711	1200
1	0,0031	6	0,2	1	715	1200
1	0,0031	9	0,2	1	616	1200
1	0,031	4	0	1	106	1200
1	0,031	4	0,1	1	106	1200
1	0,031	4	0,3	1	106	1200
1	0,031	4	0	2	417	1200
1	0,031	4	0	3	19	1200

Continued on next page

Table 19: results hyperparameters *oneSwitch, switchPeriodically, switchWhenLM* and *onlyAW* for *mycielski* graphs. (Continued)

1	0,031	4	0	4	236	1200
1	0,031	4	0	5	342	1200
switchPeriodically						
0,01	0,001	4	0,2	1	10	809
0,01	0,0031	4	0,2	1	5	336
0,01	0,01	4	0,2	1	1	201
0,01	0,031	4	0,2	1	500	1200
0,031	0,001	4	0,2	1	8	1093
0,031	0,0031	4	0,2	1	3	833
0,031	0,01	4	0,2	1	9	155
0,031	0,031	4	0,2	1	300	1200
0,1	0,001	4	0,2	1	1154	1200
0,1	0,0031	4	0,2	1	100	1200
0,1	0,01	4	0,2	1	9	288
0,1	0,031	4	0,2	1	100	1200
0,31	0,001	4	0,2	1	8376	1200
0,31	0,0031	4	0,2	1	207	1200
0,31	0,01	4	0,2	1	200	1200
0,31	0,031	4	0,2	1	1	533
1	0,001	4	0,2	1	9920	1200
1	0,0031	4	0,2	1	102	1200
1	0,01	4	0,2	1	500	1200
1	0,031	4	0,2	1	2	368
0,031	0,01	1	0,2	1	9	376
0,031	0,01	3	0,2	1	3	174
0,031	0,01	6	0,2	1	1	150
0,031	0,01	9	0,2	1	300	1200
0,031	0,01	6	0	1	1	150
0,031	0,01	6	0,1	1	4	154
0,031	0,01	6	0,3	1	200	1200
0,031	0,01	6	0	2	1	305
0,031	0,01	6	0	3	200	1200
0,031	0,01	6	0	4	6	317
0,031	0,01	6	0	5	98	1200
switchWhenLM						
0,01	0,001	4	0,2	1	9999	1200

Continued on next page

Table 19: results hyperparameters *oneSwitch, switchPeriodically, switchWhenLM* and *onlyAW* for *mycielski* graphs. (Continued)

0,01	0,0031	4	0,2	1	1936	1200
0,01	0,01	4	0,2	1	95	1200
0,01	0,031	4	0,2	1	13286	1200
0,031	0,001	4	0,2	1	16160	1200
0,031	0,0031	4	0,2	1	3757	1200
0,031	0,01	4	0,2	1	555	1200
0,031	0,031	4	0,2	1	265	1200
0,1	0,001	4	0,2	1	1695	1200
0,1	0,0031	4	0,2	1	6807	1200
0,1	0,01	4	0,2	1	1043	1200
0,1	0,031	4	0,2	1	529	1200
0,31	0,001	4	0,2	1	5934	1200
0,31	0,0031	4	0,2	1	747	1200
0,31	0,01	4	0,2	1	2788	1200
0,31	0,031	4	0,2	1	932	1200
1	0,001	4	0,2	1	6146	1200
1	0,0031	4	0,2	1	296	1200
1	0,01	4	0,2	1	1	1198
1	0,031	4	0,2	1	100	1200
1	0,01	1	0,2	1	1031	1200
1	0,01	3	0,2	1	153	1200
1	0,01	6	0,2	1	116	1200
1	0,01	9	0,2	1	3201	1200
1	0,01	4	0	1	1	1198
1	0,01	4	0,1	1	1	1198
1	0,01	4	0,3	1	1	1198
1	0,01	4	0,2	2	328	1200
1	0,01	4	0,2	3	165	1200
1	0,01	4	0,2	4	226	1200
1	0,01	4	0,2	5	147	1200
onlyAW						
0,01	0,001	4	0,2	1	24277	600
0,01	0,0031	4	0,2	1	8040	600
0,01	0,01	4	0,2	1	1844	600
0,01	0,031	4	0,2	1	5780	600
0,031	0,001	4	0,2	1	27262	600
0,031	0,0031	4	0,2	1	14444	600

Continued on next page

Table 19: results hyperparameters *oneSwitch, switchPeriodically, switchWhenLM* and *onlyAW* for *mycielski* graphs. (Continued)

0,031	0,01	4	0,2	1	964	600
0,031	0,031	4	0,2	1	733	600
0,1	0,001	4	0,2	1	28829	600
0,1	0,0031	4	0,2	1	24406	600
0,1	0,01	4	0,2	1	8801	600
0,1	0,031	4	0,2	1	1534	600
0,31	0,001	4	0,2	1	29139	600
0,31	0,0031	4	0,2	1	28463	600
0,31	0,01	4	0,2	1	16829	600
0,31	0,031	4	0,2	1	4065	600
1	0,001	4	0,2	1	29159	600
1	0,0031	4	0,2	1	28718	600
1	0,01	4	0,2	1	23158	600
1	0,031	4	0,2	1	6539	600
0,031	0,031	1	0,2	1	2724	600
0,031	0,031	3	0,2	1	512	600
0,031	0,031	6	0,2	1	252	600
0,031	0,031	9	0,2	1	2562	600
0,031	0,031	6	0	1	188	600
0,031	0,031	6	0,1	1	1146	600
0,031	0,031	6	0,3	1	877	600
0,031	0,031	6	0	2	236000	600
0,031	0,031	6	0	3	41490	600
0,031	0,031	6	0	4	236000	600
0,031	0,031	6	0	5	236000	600