Graph Coloring

Alexandrescu Nicolae,Apetria George,Ivan Remus

March 2024

1 Introduction

Graph coloring refers to the problem of coloring vertices of a graph in such a way that no two adjacent vertices have the same color. This is also called the vertex coloring problem. If coloring is done using at most m colors, it is called m-coloring. The first results about graph coloring deal almost exclusively with planar graphs in the form of map coloring. While trying to color a map of the counties of England, Francis Guthrie postulated the four color conjecture, noting that four colors were sufficient to color the map so that no regions sharing a common border received the same color. Guthrie's brother passed on the question to his mathematics teacher Augustus De Morgan at University College, who mentioned it in a letter to William Hamilton in 1852. Arthur Cayley raised the problem at a meeting of the London Mathematical Society in 1879.

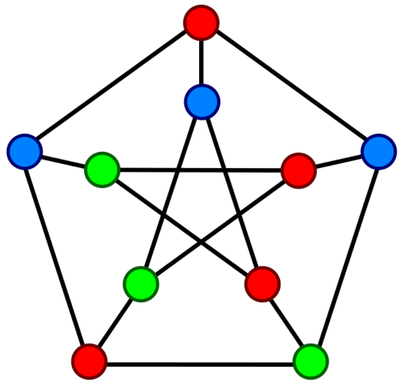


Figure 1: Petersen graph 3-coloring

More formally,we have a Graph Coloring Problem(GCP) Instance $I = (\mathcal{G}, C)$ composed of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $C \in \mathbb{N}, C > 2$.Find a function $f : \mathcal{V} - > \mathcal{C}$ such that $\forall u, v \in \mathcal{V}, u \neq v, f(u) \neq f(v)$.

2 Benchmark Instances

The selection of benchmark instances is an important factor in the empirical analysis of an algorithm's behaviour, and the use of inadequate benchmark sets can lead to questionable results and misleading conclusions. The criteria for benchmark selection depend significantly on the problem domain under consideration, on the hypotheses and goals of the empirical study, and on the algorithms being analysed.
Here is a list of Graph-Vertex Coloring instances used in the literature. For each instance the table gives:

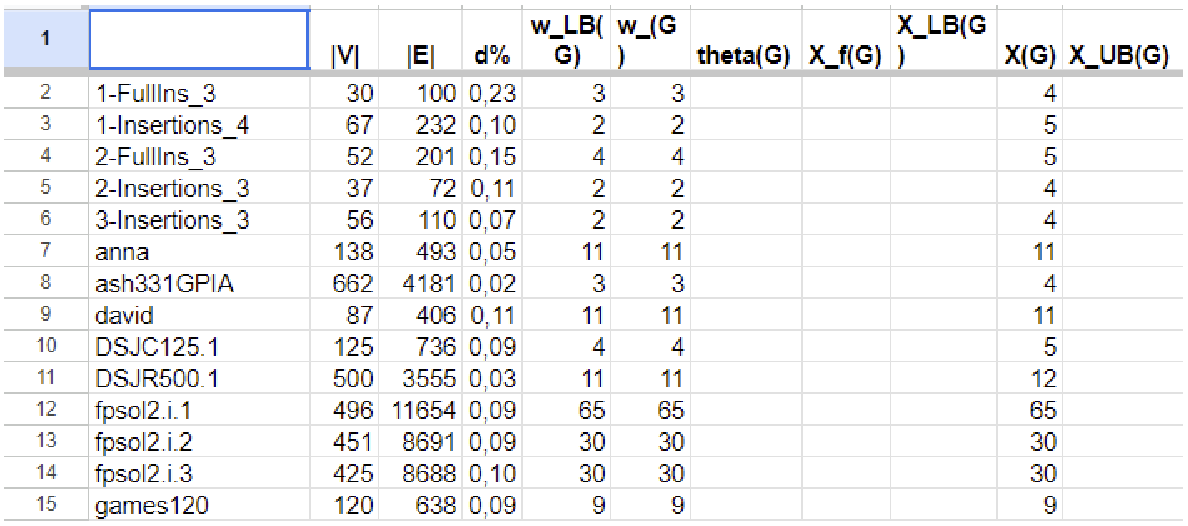
- $|V|$ number of vertices

- $|E|$ number of edges
- d% graph density, corresponding to the number of edges divided by ($|V|$ choose 2)
- w_LB(G) lower bound to the size of the maximum clique
- w_(G) the size of the maximum clique
- theta(G) the theta number computed via semi definite programming
- X_f(G) the fractional chromatic number
- X_LB(G) lower bound to the chromatic number if computed in a way different from the previous
- X(G) chromatic number
- X_UB(G) upper bound to the chromatic number, aka heuristic solution

The instances are classified according to their difficulty. We follow the SteinLib classification:
All instances for which no polynomial time algorithm is known are classified as NP.
The letter after this identifier indicates how long it takes to solve the problem using state-of-the-art soft- and hardware.

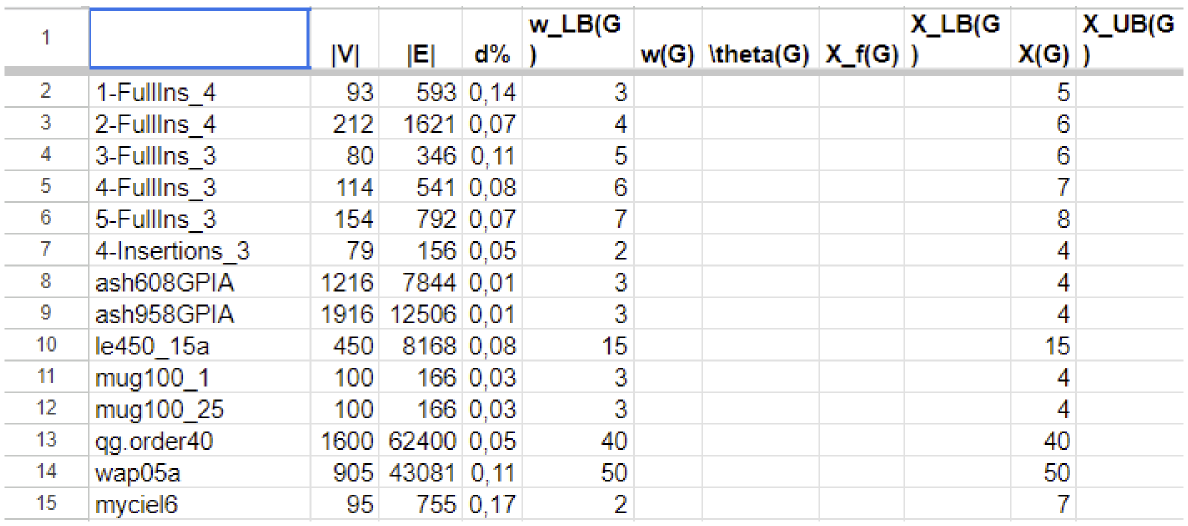
- s - seconds means less than a minute (this includes instances which can be solved in fractions of a second)
- m - minutes means less than an hour
- h - hours is less than a day
- d - days is less than a week
- w - weeks means it takes really a long time to solve this instance
- ? means the instance is not solved or the time is not known. If the chromatic number is given for this instances, then it is known by construction.

See Figure 2, Figure 3, Figure 4 and Figure 5.



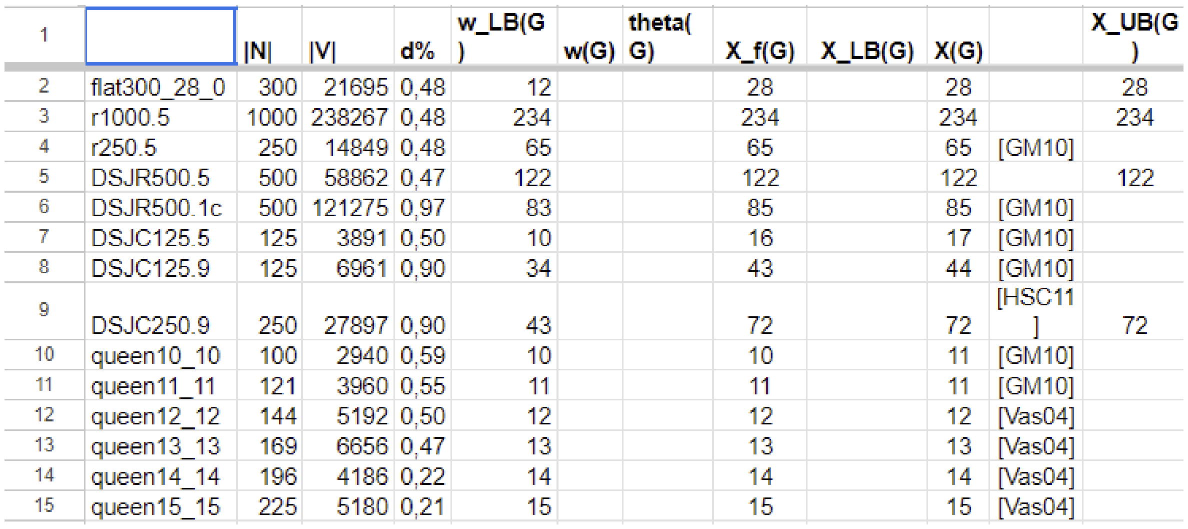
1		|V|	|E|	d%	w_LB(G)	w_(G)	theta(G)	X_f(G)	X_LB(G)	X(G)	X_UB(G)
2	1-FullIns_3	30	100	0,23	3	3				4	
3	1-Insertions_4	67	232	0,10	2	2				5	
4	2-FullIns_3	52	201	0,15	4	4				5	
5	2-Insertions_3	37	72	0,11	2	2				4	
6	3-Insertions_3	56	110	0,07	2	2				4	
7	anna	138	493	0,05	11	11				11	
8	ash331GPIA	662	4181	0,02	3	3				4	
9	david	87	406	0,11	11	11				11	
10	DSJC125.1	125	736	0,09	4	4				5	
11	DSJR500.1	500	3555	0,03	11	11				12	
12	fpsol2.i.1	496	11654	0,09	65	65				65	
13	fpsol2.i.2	451	8691	0,09	30	30				30	
14	fpsol2.i.3	425	8688	0,10	30	30				30	
15	games120	120	638	0,09	9	9				9	

Figure 2: NP-s instances



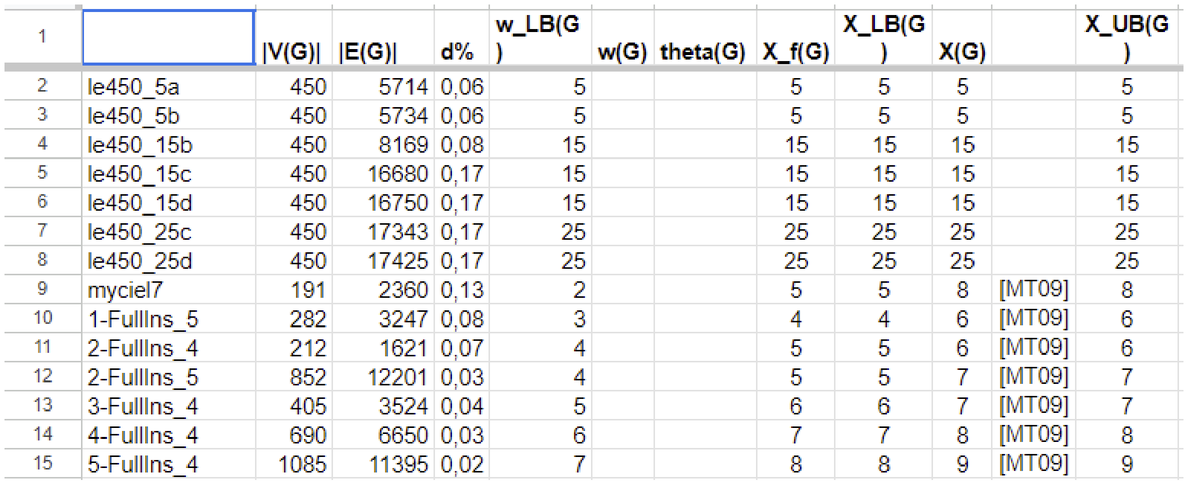
1		|V|	|E|	d%	w_LB(G)	w(G)	\theta(G)	X_f(G)	X_LB(G)	X(G)	X_UB(G)
2	1-FullIns_4	93	593	0,14	3					5	
3	2-FullIns_4	212	1621	0,07	4					6	
4	3-FullIns_3	80	346	0,11	5					6	
5	4-FullIns_3	114	541	0,08	6					7	
6	5-FullIns_3	154	792	0,07	7					8	
7	4-Insertions_3	79	156	0,05	2					4	
8	ash608GPIA	1216	7844	0,01	3					4	
9	ash958GPIA	1916	12506	0,01	3					4	
10	le450_15a	450	8168	0,08	15					15	
11	mug100_1	100	166	0,03	3					4	
12	mug100_25	100	166	0,03	3					4	
13	qg.order40	1600	62400	0,05	40					40	
14	wap05a	905	43081	0,11	50					50	
15	myciel6	95	755	0,17	2					7	

Figure 3: NP-m instances



1		|N|	|V|	d%	w_LB(G)	w(G)	theta(G)	X_f(G)	X_LB(G)	X(G)		X_UB(G)
2	flat300_28_0	300	21695	0,48	12			28		28		28
3	r1000.5	1000	238267	0,48	234			234		234		234
4	r250.5	250	14849	0,48	65			65		65	[GM10]	
5	DSJR500.5	500	58862	0,47	122			122		122		122
6	DSJR500.1c	500	121275	0,97	83			85		85	[GM10]	
7	DSJC125.5	125	3891	0,50	10			16		17	[GM10]	
8	DSJC125.9	125	6961	0,90	34			43		44	[GM10]	
9	DSJC250.9	250	27897	0,90	43			72		72	[HSC11]	72
10	queen10_10	100	2940	0,59	10			10		11	[GM10]	
11	queen11_11	121	3960	0,55	11			11		11	[GM10]	
12	queen12_12	144	5192	0,50	12			12		12	[Vas04]	
13	queen13_13	169	6656	0,47	13			13		13	[Vas04]	
14	queen14_14	196	4186	0,22	14			14		14	[Vas04]	
15	queen15_15	225	5180	0,21	15			15		15	[Vas04]	

Figure 4: NP-h instances



1		|V(G)|	|E(G)|	d%	w_LB(G)	w(G)	theta(G)	X_f(G)	X_LB(G)	X(G)		X_UB(G)
2	le450_5a	450	5714	0,06	5			5	5	5		5
3	le450_5b	450	5734	0,06	5			5	5	5		5
4	le450_15b	450	8169	0,08	15			15	15	15		15
5	le450_15c	450	16680	0,17	15			15	15	15		15
6	le450_15d	450	16750	0,17	15			15	15	15		15
7	le450_25c	450	17343	0,17	25			25	25	25		25
8	le450_25d	450	17425	0,17	25			25	25	25		25
9	myciel7	191	2360	0,13	2			5	5	8	[MT09]	8
10	1-FullIns_5	282	3247	0,08	3			4	4	6	[MT09]	6
11	2-FullIns_4	212	1621	0,07	4			5	5	6	[MT09]	6
12	2-FullIns_5	852	12201	0,03	4			5	5	7	[MT09]	7
13	3-FullIns_4	405	3524	0,04	5			6	6	7	[MT09]	7
14	4-FullIns_4	690	6650	0,03	6			7	7	8	[MT09]	8
15	5-FullIns_4	1085	11395	0,02	7			8	8	9	[MT09]	9

Figure 5: NP-? instances

3 Baseline Algorithms

Previous algorithms that show good results are the Greedy Algorithm(choose to assign colors to uncolored vertexes in a particular order),DSATUR, Tabu Search(starts with invalid k colorings and attempts to reduce the number of total conflicts by finding better solutions through exploring the neighborhood via tabu search),Quantum Simulated Annealing(add the forces of attraction and repulsion as a way of visiting the neighborhood of a solution to the Simulated Annealing).

3.1 Particle Swarm Optimization

Particle Swarm Optimization (PSO) stands as a noteworthy paradigm in the realm of optimization algorithms, harnessing inspiration from the collective behaviors observed in nature to tackle complex problem spaces. Originating in the late 1990s through the seminal work of Eberhart and Kennedy, PSO has evolved into a widely recognized metaheuristic, admired for its simplicity, computational efficiency, and efficacy in navigating high-dimensional solution spaces [2].

The quest for an improved version of Particle Swarm Optimization (PSO) stems from the inherent desire to enhance the algorithm's robustness and efficiency in addressing complex optimization challenges. While the standard PSO exhibits admirable simplicity and effectiveness, its performance can be influenced by factors such as parameter settings and convergence speed. An improved iteration seeks to overcome these limitations by introducing refinements that enhance convergence rates, promote a more thorough exploration of the solution space, and mitigate premature convergence.

The framework of the conventional PSO algorithm is derived from [2]. The provided pseudocode outlines the structure of the Particle Swarm Optimization (PSO) algorithm. The algorithm initializes a population of particles with random positions and velocities. It then defines the personal best positions for each particle and initializes the global best position. The fitness function, used to evaluate the quality of solutions, is also specified. The core of the algorithm lies within a loop, where particles are iteratively evaluated, and their positions and velocities are updated based on the PSO update equations. The process continues until certain termination conditions are met.

The variant of the Particle Swarm Optimization presented in the lines above is global also called (g_{best}). Unlike the local version (l_{best}) in which several neighborhoods are considered, in this case, all particles are in the same neighborhood, as can be seen in fig. 1.

This variant was also used in the implementation carried out in order to execute the proposed experiment - determining the optimum values for graphs provided as inputs in terms of number of colors used for graph coloring.

An important aspect that should be mentioned is the fact that the l_{best} approach leads to greater diversity, but this approach is slower in terms of execution time required to determine the optimal value compared to the g_{best} approach.

The pseudocode in **Algorithm 1** serves as a high-level representation of the key steps and operations involved in the standard PSO algorithm.
The velocity update equation for particle i is given by:

$$v_i = wv_i + c_1 r_1(pbest_i - p_i) + c_2 r_2(gbest - p_i) \tag{1}$$

The position update equation for particle i is given by:

$$p_i = p_i + v_i \tag{2}$$

Here, w represents the inertia weight, c_1 and c_2 are the acceleration coefficients, r_1 and r_2 are random numbers, $pbest_i$ is the personal best position for particle i, $gbest$ is the

Algorithm 1 Particle Swarm Optimization (PSO)

```
Initialize particles with random positions and velocities
Initialize personal best positions for each particle
Initialize global best position
Define fitness function for evaluation termination conditions not met each particle
Evaluate fitness using the fitness function
Update personal best if current position is better
Update global best if any particle has a better personal best
Update velocity and position using the PSO update equations
```

global best position, and p_i and v_i are the current position and velocity of particle i, respectively.

4 SOTA

4.1 Graph Colouring Meets Deep Learning: Effective Graph Neural Network Models for Combinatorial Problems

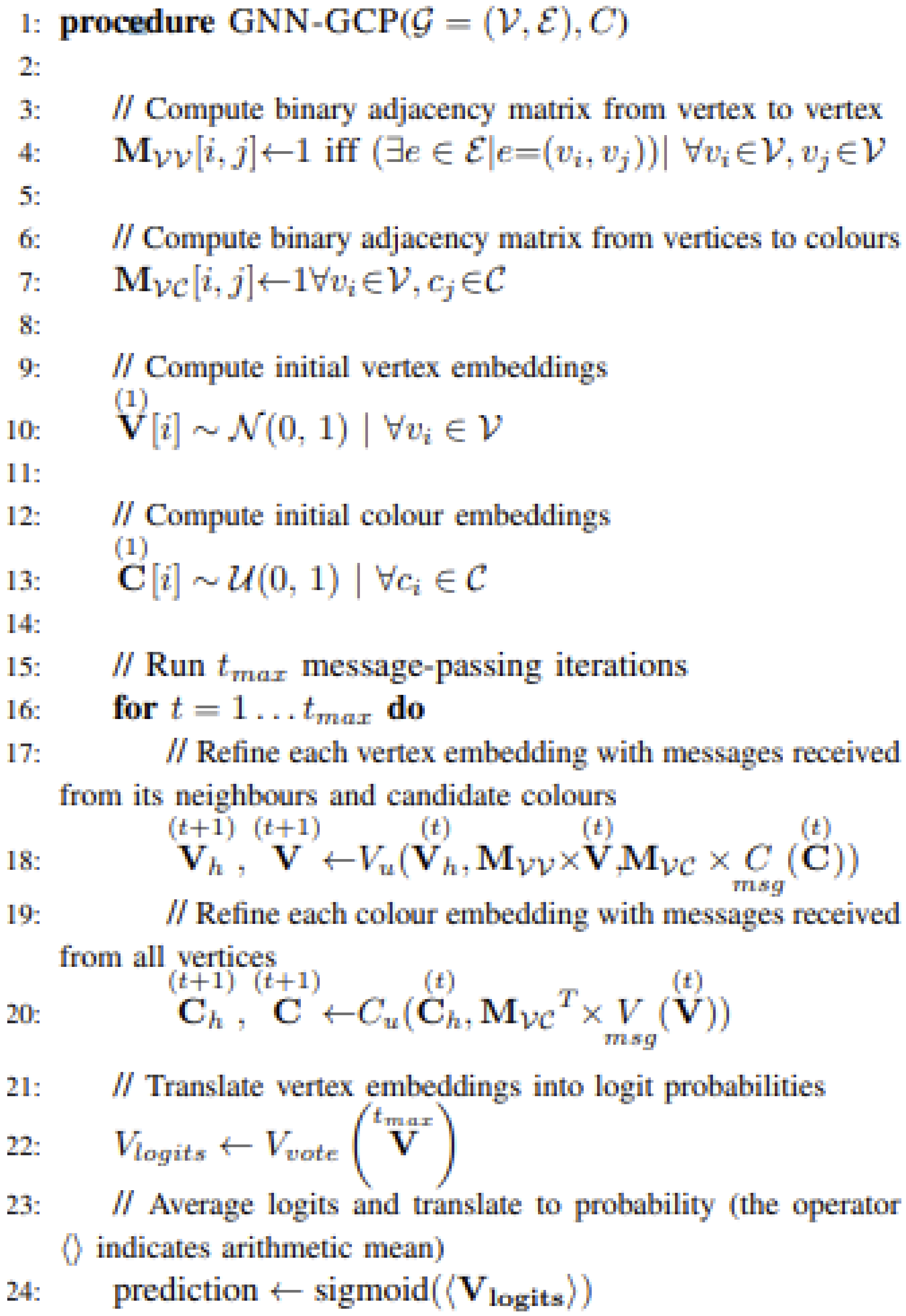
The abstract of the paper [4] discusses the recent advancements in deep learning, particularly in symbolic domains, and introduces the concept of employing Graph Neural Networks (GNNs) for solving combinatorial problems. It highlights the success of GNNs in training on complex relational data, including NP-Complete problems like SAT and TSP. The study focuses on how a simple GNN architecture can effectively tackle the fundamental combinatorial problem of graph coloring. The results demonstrate high accuracy on random instances and generalization to unseen graph distributions, outperforming several baseline methods. Additionally, the study shows how vertex embeddings can be clustered to provide constructive solutions, even though the model is trained solely as a binary classifier. Overall, the findings contribute to bridging the gap in understanding the algorithms learned by GNNs and provide empirical evidence of their capability in solving hard combinatorial problems, furthering the integration of robust learning and symbolic reasoning in deep learning systems.

One approach to incorporating relational structures into neural models involves ensuring permutation invariance through neural modules with parameter sharing. This leads to the development of various models such as message-passing neural networks, recurrent relational networks, and Graph Neural Networks (GNNs). The paragraph also discusses recent work on GNNs tackling NP-Complete problems like boolean satisfiability (SAT) and introduces a new model to solve the graph coloring problem (GCP) without prior reductions. The GNN framework is employed for this purpose, leveraging its ability to handle various types of edges. The authors aim to promote the adoption and further research on GNN-like models integrating deep learning and combinatorial optimization.

4.1.1 A GNN model for decision GCP

In typical GNN models, vertices and edges in the graph are assigned multidimensional representations or embeddings $\in \mathbb{R}^d$, which are refined based on adjacency information through message-passing iterations. The adjacency information filters valid incoming messages for each vertex or edge, aggregates these messages, and computes updates to the embeddings using a Recurrent Neural Network (RNN). The Graph Network model also allows for global graph attributes, which is suitable for the k-colorability problem. However, since each color needs its own embedding, the authors chose to model the k-colorability problem using a Graph Neural Network framework, which can handle multiple types of nodes. Given a GCO instance $I = (\zeta, C)$ composed of a graph $\zeta = (\vartheta, \varepsilon)$ and a number of colours $C \in \mathbb{N}|C > 2$, each colour is assigned to a random initial embedding over an uniform distribution $C[i] \sim \; \upsilon(0,1)|\forall c_i \in C$ and the model initially assigns the same embedding $\in \mathbb{R}^d$ to all V vertices: this embedding is randomly initialised and then it becomes a trained parameter learned by the model.

To allow the communication between neighbouring vertices and between vertices and colours, besides the vertex-to-vertex adjacency matrix $M_{\mathcal{V}\mathcal{V}} \in \{0,1\}^{|\mathcal{V}|\times|\mathcal{V}|}$, the model also requires a vertex to-colour adjacency matrix $M_{\mathcal{V}\mathcal{C}} \in \{0,1\}^{|\mathcal{V}|\times|\mathcal{C}|}$, that connects each colour to all vertices since we chose to give no prior information to the model; i.e., a priori any vertex can be assigned to any colour. After this initialisation adjacent vertices and colours communicate and update their embeddings during a given number of iterations. Then the resulting vertex embeddings are fed into a MLP which computes a logit probability corresponding to the model's prediction of the answer to the decision problem: "does the graph G accept a C-coloration?". This procedure is summarised in Algorithm 1 (see Figure 6).



```
procedure GNN-GCP($\mathcal{G} = (\mathcal{V}, \mathcal{E}), C$)

    // Compute binary adjacency matrix from vertex to vertex
    $\mathbf{M}_{\mathcal{VV}}[i,j] \leftarrow 1$ iff $(\exists e \in \mathcal{E} | e = (v_i, v_j)) | \; \forall v_i \in \mathcal{V}, v_j \in \mathcal{V}$

    // Compute binary adjacency matrix from vertices to colours
    $\mathbf{M}_{\mathcal{VC}}[i,j] \leftarrow 1 \forall v_i \in \mathcal{V}, c_j \in \mathcal{C}$

    // Compute initial vertex embeddings
    $\overset{(1)}{\mathbf{V}}[i] \sim \mathcal{N}(0, 1) \; | \; \forall v_i \in \mathcal{V}$

    // Compute initial colour embeddings
    $\overset{(1)}{\mathbf{C}}[i] \sim \mathcal{U}(0, 1) \; | \; \forall c_i \in \mathcal{C}$

    // Run $t_{max}$ message-passing iterations
    for $t = 1 \ldots t_{max}$ do
        // Refine each vertex embedding with messages received from its neighbours and candidate colours
        $\overset{(t+1)}{\mathbf{V}_h}, \overset{(t+1)}{\mathbf{V}} \leftarrow V_u(\overset{(t)}{\mathbf{V}_h}, \mathbf{M}_{\mathcal{VV}} \times \overset{(t)}{\mathbf{V}}, \mathbf{M}_{\mathcal{VC}} \times \underset{msg}{C}(\overset{(t)}{\mathbf{C}}))$
        // Refine each colour embedding with messages received from all vertices
        $\overset{(t+1)}{\mathbf{C}_h}, \overset{(t+1)}{\mathbf{C}} \leftarrow C_u(\overset{(t)}{\mathbf{C}_h}, {\mathbf{M}_{\mathcal{VC}}}^T \times \underset{msg}{V}(\overset{(t)}{\mathbf{V}}))$
    // Translate vertex embeddings into logit probabilities
    $V_{logits} \leftarrow V_{vote}\left(\overset{t_{max}}{\mathbf{V}}\right)$
    // Average logits and translate to probability (the operator $\langle\rangle$ indicates arithmetic mean)
    prediction $\leftarrow$ sigmoid($\langle \mathbf{V_{logits}} \rangle$)
```

Figure 6: Algorithm 1 Graph Neural Network Model for GCP

The GNN-based algorithm updates vertex and colour embeddings, along with their respective hidden states, according to the following equations:

$$V^{(t+1)}, V_h^{(t+1)} \leftarrow \vartheta_u(V_h^{(t)}, M_{\vartheta\vartheta} \times (V^{(t)}), M_{\vartheta C} \times C_{msg}(C^{(t)})(\ 1)$$

$$C^{(t+1)}, C_h^{(t+1)} \leftarrow C_u(C_h^{(t)}, M_{\vartheta C} \times V_{msg}(V^{(t)})(\ 2)$$

In this case, the GNN model needs to learn the message functions (implemented via MLPs) $C_{msg} : \mathbb{R}^d \to \mathbb{R}^d$, which will translate colours embeddings into messages that are intelligible to a vertex update function, and $V_m sg : \mathbb{R}^d \to \mathbb{R}^d$, responsible for translating vertices embeddings into messages. Also, it learns a function (RNN) responsible for updating vertices $V_u : \mathbb{R}^d \to \mathbb{R}^d$ given its hidden state and received messages and another RNN to do the analogous procedure to the colors $V_u : \mathbb{R}^d \to \mathbb{R}^d$.

4.1.2 Training Metodology

With this embedding as a graph problem as well as the given number of colors $\mathcal{C}$,we then let the RNN simulate the message passing(passing of the information from nodes adjacent to each other).This network will work as a binary classifier in order to predict the chromatic number χ of the GCP. In order to train this network,we can use randomly generated instances I with $\chi(I) = \mathcal{C}$ and create an adversarial instance I' that has the chromatic number $\chi(I') = \mathcal{C} + 1$.

The MLPs responsible for message computing are three-layered (64,64,64) with ReLU nonlinearities as the activations for all layers except for the linear activation on the output layer. The RNN cells are basic LSTM cells with layer normalisation and ReLu activation.

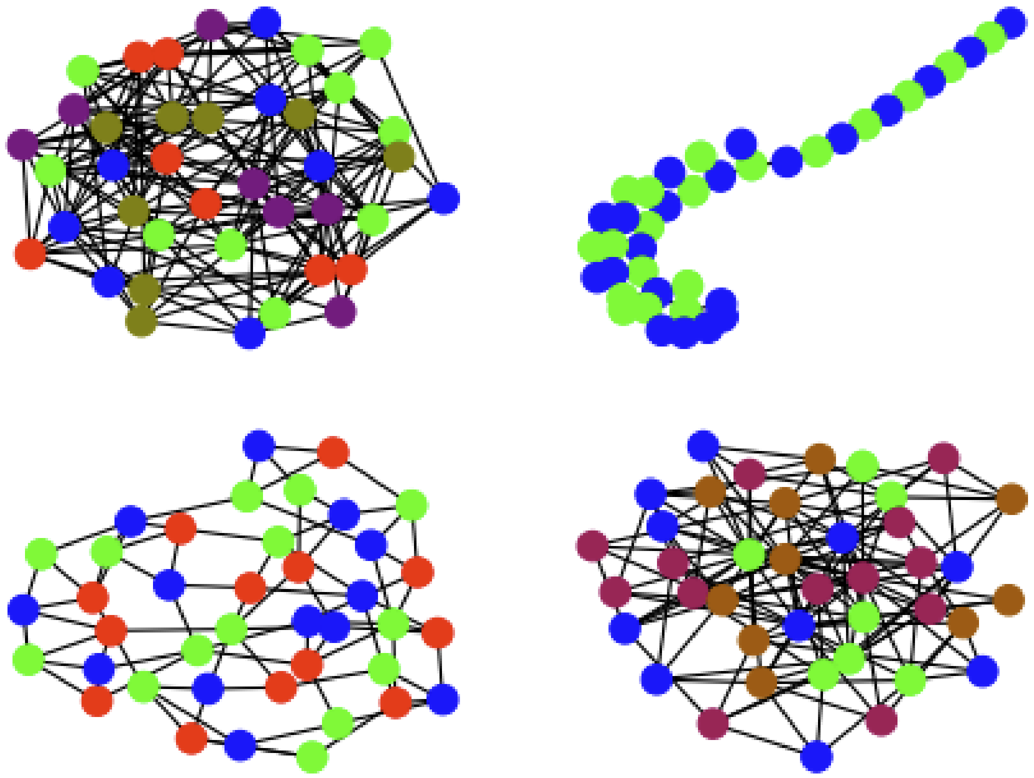


Figure 7: Pictorial representation of random training instances (top left) and some of the structured test instances (clockwise: power-law tree, power-law cluster, small-world). All instances are coloured with a number of colours equal to their chromatic number.

4.1.3 Experimental results and analyses

The authors stopped the training procedure when the model achieved 82% accuracy and 0.35 Binary Cross Entropy loss averaged over 128 batches containing 16 instances at the end of 5300 epochs. In Figure 8 you can see a comparison between GNN and other algorithms. Figure 9 and Figure 10 present more results of the GNN in comparison with the others.

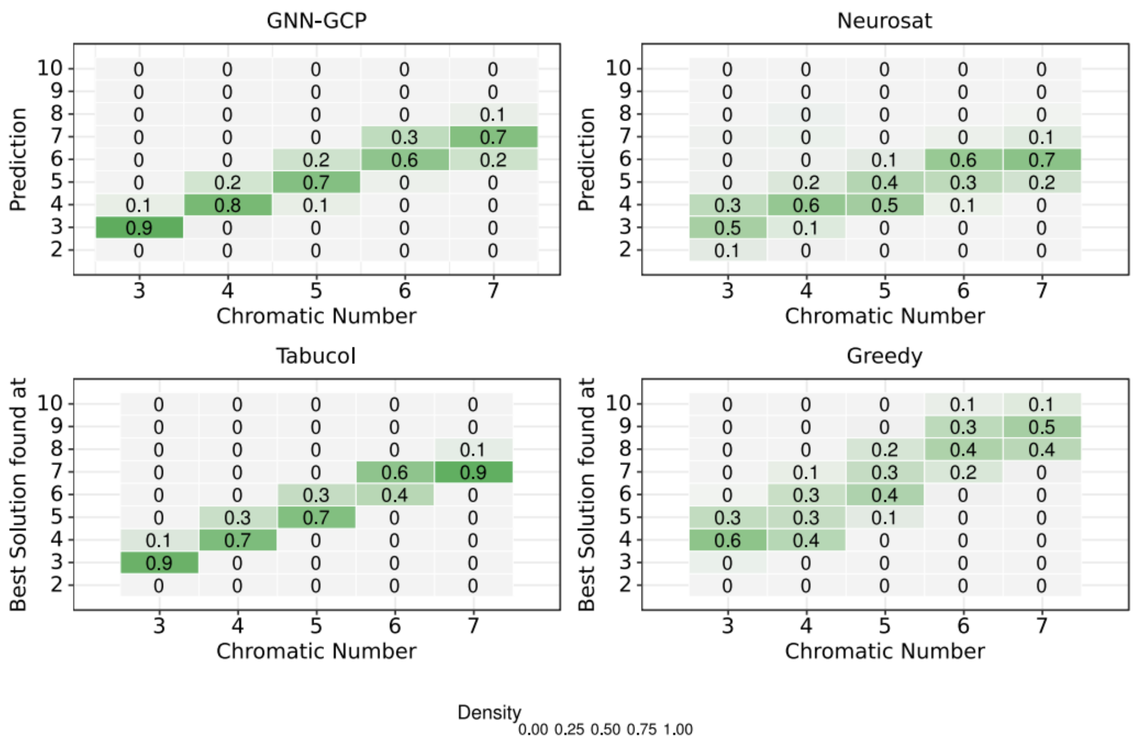
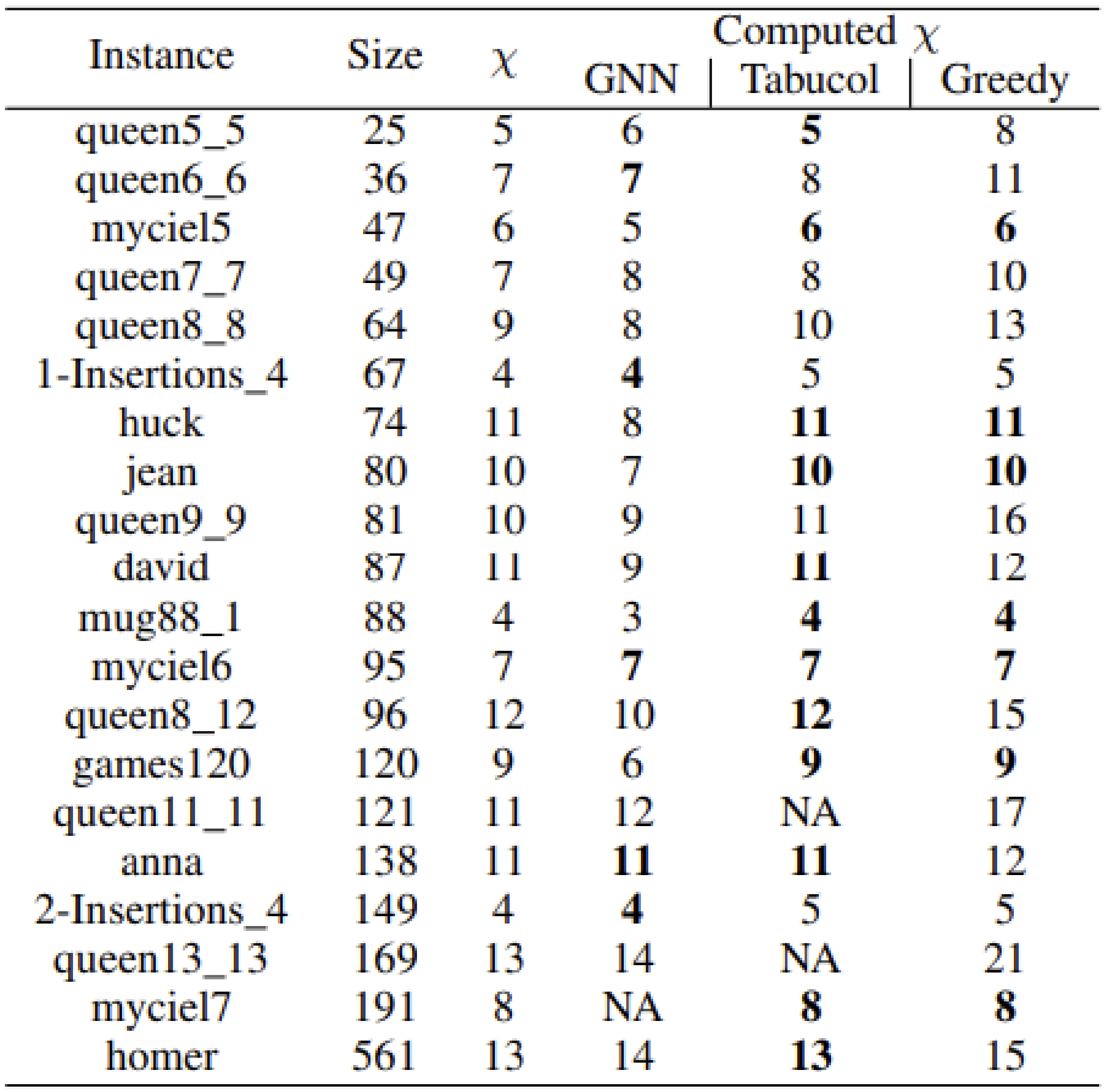
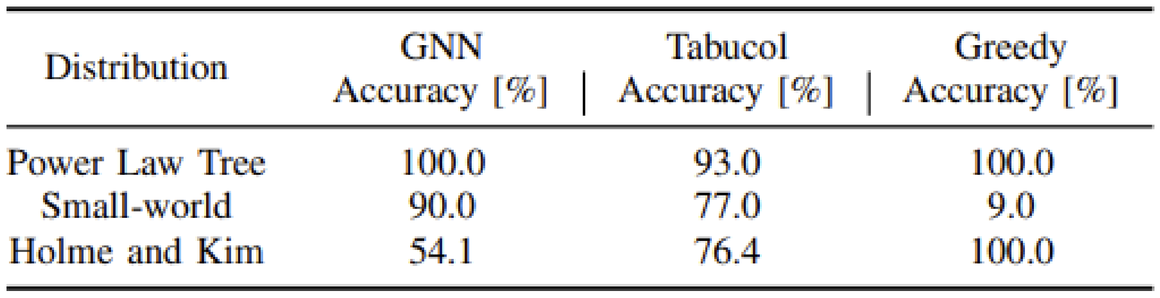


Figure 8: Prediction distributions over 4096 unseen test instances, with similar features to those seen in training, for GNN model, Neurosat, Tabucol and the Greedy algorithm. Note that the darker the main diagonal, the better the predictions.



Instance	Size	χ	Computed χ GNN	Tabucol	Greedy
queen5_5	25	5	6	**5**	8
queen6_6	36	7	**7**	8	11
myciel5	47	6	5	**6**	**6**
queen7_7	49	7	8	8	10
queen8_8	64	9	8	10	13
1-Insertions_4	67	4	**4**	5	5
huck	74	11	8	**11**	**11**
jean	80	10	7	**10**	**10**
queen9_9	81	10	9	11	16
david	87	11	9	**11**	12
mug88_1	88	4	3	**4**	**4**
myciel6	95	7	**7**	**7**	**7**
queen8_12	96	12	10	**12**	15
games120	120	9	6	**9**	**9**
queen11_11	121	11	12	NA	17
anna	138	11	**11**	**11**	12
2-Insertions_4	149	4	**4**	5	5
queen13_13	169	13	14	NA	21
myciel7	191	8	NA	**8**	**8**
homer	561	13	14	**13**	15

Figure 9: The chromatic number produced by the GNN model and two heuristics on some instances of the COLOR02/03/04 dataset. As the GNN model faces unseen graph sizes and larger chromatic numbers it tends to udnerestimate its answers.



Distribution	GNN Accuracy [%]	Tabucol Accuracy [%]	Greedy Accuracy [%]
Power Law Tree	100.0	93.0	100.0
Small-world	90.0	77.0	9.0
Holme and Kim	54.1	76.4	100.0

Figure 10: Strict accuracy of the GNN model and the two algorithms considering three random graph distributions.

4.2 Acceleration Based Particle Swarm Optimization for Graph Coloring Problem

Particle swarm optimization is a simple and powerful technique employed to solve the graph coloring problem. Its main drawback is its tactlessness of being trapped in the local optimum. Therefore, to overcome this, an efficient Acceleration based Particle Swarm Optimization (APSO) was introduced. Empirical study of the proposed APSO algorithm is performed on the second DIMACS challenge benchmarks. The APSO results are compared with the standard PSO algorithm and experimental results validates the superiority of the proposed APSO.

4.2.1 Description of the concept

Particle swarm optimization is a swarm intelligence approach which is inspired by the intelligence, experience-sharing, social behavior of bird flocking and fish schooling. In 1995 Kennedy and Eberhart developed PSO to solve continuous-valued space but later on it was modified for binary/discrete optimization problems. PSO is a searching technique in which a collection of particles find the global minimum that move through the search space. In PSO, a group of a particle's position and particle velocity is initialized randomly. Each particle position represents a possible solution and particle velocity represents the rate of changes of the next position with respect to the current position.

In PSO, there are local and global extremes. The fitness value is computed. If this fitness value has better-quality than the best fitness value then the current fitness value is set as new local best. Now the particle with the best fitness value of all particles is chosen as the global best. Finally for each particle velocity and particle position is calculated and updated using equations (1) and (2) respectively. It has been used across a wide range of applications, such as image and video analysis, design and restructuring of electricity networks, control, antenna design, electronics and electromagnetic. The advantages of particle swarm optimization are there is no centralized controller in the system. Hence, failure of any particle does not affect the search process. PSO is a simple, easy and only few parameters needed to be adjusted.

$$Vid(t+1) = wVid(t) + c_1R_1(Pid(t) - Xid(t)) + c_2R_2(Pgd(t) - Xid(t)) \quad (3)$$

$$Xid(t+1) = Xid(t) + Vid(t+1) \quad (4)$$

where,

Vid - Velocity of the i^{th} particle of dimension d

Pid - Best previous position of the i^{th} Particle of dimension d

Pgd - Best position of the neighbours of dimension d

Xid - Current position of the i^{th} particle of dimension d

R_1, R_2 - Random function in the range [0, 1]

w - Inertia weight that forces the particle to move in the same direction of the previous iteration

c_1 & c_2 - Acceleration constants. The first coefficient "c_1" controls the impact of the cognitive component on the particle trajectory and the second coefficient, "c_2" controls the impact of the social component. While c_1 component is responsible to maintain the diversity, c2 component is responsible to ensure convergence.

If $c_1 \ll c_2$, then particles may result into premature convergence as particles are attracted more towards the global best position then to their personal best positions. If, $c_2 \ll c_1$ then particles may result in slow convergence or may not converge at all because each particle is more attracted to its personal best positions then to global best. Since the PSO relies on a combination of both personal and social knowledge of the given search space, so the coefficients c_1 and c_2 are chosen such that $c_1 \cong c_2$. For too large values of c_1 and c_2, particle velocities accelerate too fast, leading in swarm divergence. On the other hand for too small values of c_1 and c_2 , swarm convergence time increases as particles move too slowly. In order to find the optimum solution efficiently proper control on global exploration and local exploration is crucial. Generally high diversity is necessary during the early part of the search to allow the use of the full range of the search space on the other hand, during the latter part of the search, when the algorithm is converging to the optimal solution, fine tuning of the solution is important to find the global optima efficiently. Therefore a proper control of these two components is very important to find the optimum solution accurately and efficiently. Particles draw their strength from their cooperative nature, and are most effective when c_1 and c_2 are adaptive to the particle value to facilitate exploitation and exploration of the search area. Hence to enhance the performance of PSO, Acceleration based PSO(APSO) was introduced.

4.2.2 Acceleration Based Particle Swarm Optimization (APSO)

In this research work [1] , an Acceleration based Particle Swarm Optimization (APSO) algorithm is developed to solve premature convergence problem of the standard PSO. In APSO, the acceleration coefficients are selected based on the fitness value which will increase the accuracy of the results. The selection of acceleration coefficient values in APSO algorithm is described as follows:

$$nc_1 = c_1 \times (1 - \lambda) \quad (5)$$

$$nc_2 = c_2 \times (1 + \lambda) \tag{6}$$

In eq. (3) and (4), the λ value is computed based on the fitness value and is calculated as:

$$\lambda = \frac{\chi(1 + \varphi(f_{max} - fmin)^{\omega} - (f_{avg}))}{\delta(f_{max} - f_{min})^{\omega} - f_{avg}^{\omega}} \tag{7}$$

$$\lambda = \left(\frac{f_{max} - fmin}{f_{avg}}\right)^{\omega} \tag{8}$$

Where,
χ - Alteration probability
ω, φ - Coefficient factors
F_{max}, F_{min}, F_{avg} - Maximum, minimum and average fitness of the particles

By exploiting eq. (3) and (4), the acceleration coefficients values, the velocity formula which is given in eq. (1) is updated by eq. (7). However the position update equation remains the same:

$$Vid(t+1) = wVid(t) + nc_1R_1(Pid(t) - Xid(t)) + nc_2R_2(Pgd(t) - Xid(t)) \tag{9}$$

4.2.3 Experiments and Results

The values of parameters in the APSO algorithm for the graph coloring problem are presented in Table 1. These values are selected based on some preliminary trials. To validate the performance of the proposed APSO algorithm, an extensive experiments on some benchmarks are conducted. Benchmark graphs are derived from the well known DIMACS challenge benchmarks illustrated in Table 2. These instances cover a variety of types and sizes of graphs. For each instance in table 2, 10 independent runs of the algorithm were carried out. Table 3 shows the experimental results. Column 2 records the graph name, column 3 records the expected chromatic number. Column 4 is reported for standard PSO which is divided in 2 sub columns in which first column represent the total steps and other column represent PSO result. Column 5 is recorded for APSO which is divided into two parts in which one part represent the total steps and second part represent the APSO result. We run APSO independently on every graph and found that in case of Mycielski graphs such as myciel3.col, myciel4.col, myciel5.col, APSO found the optimal solution in each run. Then Stanford GraphBase (SGB) such as huck.col, jean.col, david.col. games120.col, anna.col is tested independently and APSO found an optimal solution in each run. For miles graphs was found that for miles250.col APSO is able to produce an optimal solution in each run while miles1000.col produces an optimal solution in 9 runs. On queen graph i.e on queen 5_5.col an optimal solution with the predefined parameter was not obtained. Hence. the parameter value of population size was updated to 50 and maximum step size to 500 then the optimal solution was obtained using APSO.

4.2.4 PSO VS APSO comparison

Standard PSO was compared with the proposed APSO algorithm. Experimental results show that the APSO algorithm for graph coloring problem is feasible and robust. The algorithm was able to scale across the different graphs and produce optimum solutions in each case. APSO is capable of finding an optimal solution for all the graphs with a very high probability. APSO finds an optimal solution in each run for all graphs except miles1000.col and queen 5_5.col in 9 and 7 runs respectively while standard PSO was unable to find an optimal solution in 100 steps. The standard PSO algorithm failed to produce the optimum solution for all graphs except the two graphs named myciel3.col and myciel4.col. From the simulation result, it has been observe that APSO is competitive with the standard PSO algorithm.

Table 1: Parameter setting

Parameter	Population size	Maximum step size	Inertia weight	Maximum velocity
Value	100	100	0.9	10

Table 2: Benchmark Graphs

S No	Graph Name	Vertices	Edges
1	myciel3.col	11	20
2	myciel4.col	23	71
3	myciel5.col	47	236
4	huck.col	74	301
5	jean.col	80	254
6	david.col	87	406
7	games120.col	120	638
8	miles250.col	128	387
9	miles1000.col	128	3216
10	anna.col	138	493
11	queen5_5.col	25	160

Table 3: Experimental Results

S No	Graph	Expect. $\chi(G)$	Steps PSO	PSO res	Steps APSO	APSO res
1	myciel3.col	4	14	4	1	4
2	myciel4.col	5	59	5	1	5
3	myciel5.col	6	100	-	1	6
4	huck.col	11	100	-	1	11
5	jean.col	10	100	-	1	10
6	david.col	11	100	-	1	11
7	games120.col	9	100	-	1	9
8	miles250.col	8	100	-	1	8
9	miles1000.col	42	200	-	2	42
10	anna.col	11	100	-	1	11
11	queen5_5.col	5	100	-	32	5

4.2.5 Results analysis

In this paper was proposed an Acceleration based Particle Swarm Optimization method (APSO) to solve the graph coloring problem. With a view to achieve a highly accurate optimal outcome, the defects inherent in the PSO technique such as premature convergence and loss of diversity. These have to be properly tackled and surmounted by initiating effective alteration or augmentation in the procedure of PSO. Therefore, to attain a further precise outcome and to steer clear of the PSO defects, rather than fixing the value of acceleration coefficient, an Acceleration base Particle swarm optimization (APSO) technique in which acceleration coefficient values are updated on the basis of evaluation function was introduced. The algorithm was tested on a set of ten DIMACS benchmark test while limiting the number of usable colors to the expected optimal coloring number. Compared with the standard PSO, was found that APSO succeeded in solving the sample data set and even outperformed the standard PSO algorithm in terms of the minimum number of colors and minimum number of steps. APSO finds an optimal solution in one step for all graphs, while standard PSO is unable to find an optimal solution in 100 steps. The standard PSO algorithm failed to produce the optimum solution for all graphs except the two graphs named myciel3.col and myciel4.col. Hence computational results show that APSO is feasible and competitive with the standard PSO algorithm.

4.3 Hybrid Evolutionary Algorithm in a Duet

To introduce a variation on a hybrid evolutionary algorithm [5] ,we use a population of 2 individuals (p_1 and p_2) initialized at random and we follow first try to extract to children from the parents by applying a Greedy Partition Crossover(GPX) after which we apply a Tabu Search from the 2 children obtained.

We also have 2 other candidate solutions (similar to elite solutions), elite1 and elite2, in order to reintroduce some diversity to the duet. Indeed, after a given number of generations, the two individuals of the population become increasingly similar within the search-space. To maintain the population diversity, the idea is to replace one of the two candidates solutions by a solution previously encountered by the algorithm. We define one cycle as a number of Itercycle generations. Solution elite1 is the best solution found during the current cycle and solution elite2 the best solution found during the previous cycle. At the end of each cycle, the elite2 solution replaces one of the population individuals. Figure 12 presents the graphic view of algorithm 2.

This elitist mechanism provides relevant behaviors to the algorithm as it can be observed in the results. Indeed, elite solutions have the best fitness value of each cycle. It is clearly interesting in terms of intensification. Moreover, when the elite solution is reintroduced, it is generally different enough from the other individuals to be relevant in terms of diversification.

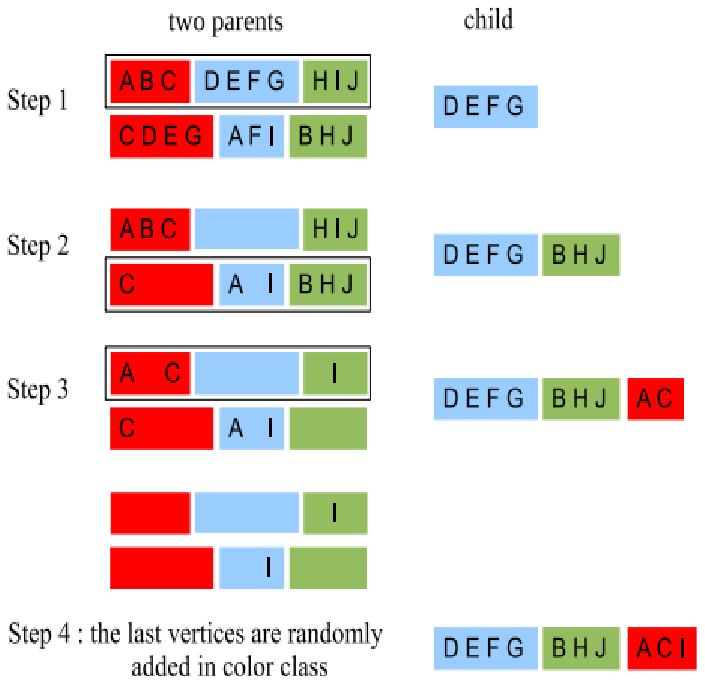
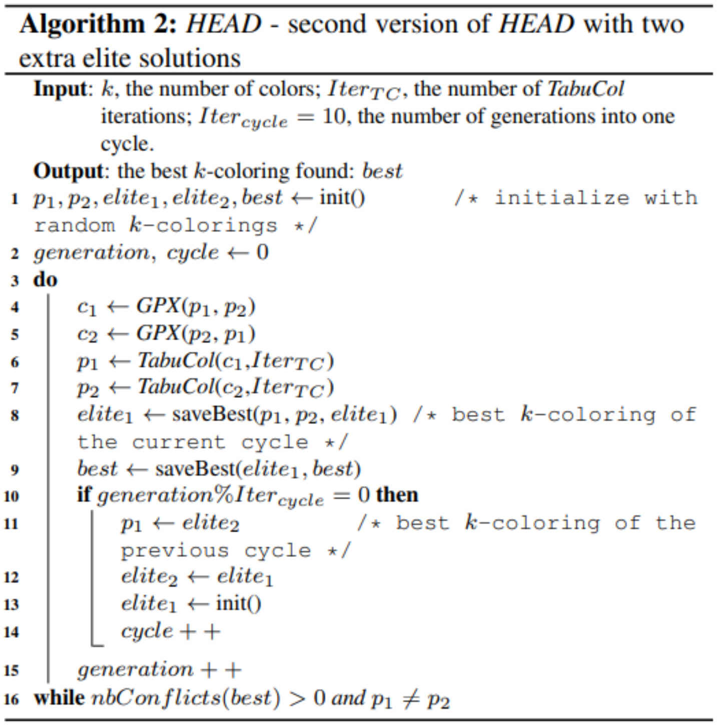


Figure 11: Greedy Partition Crossover



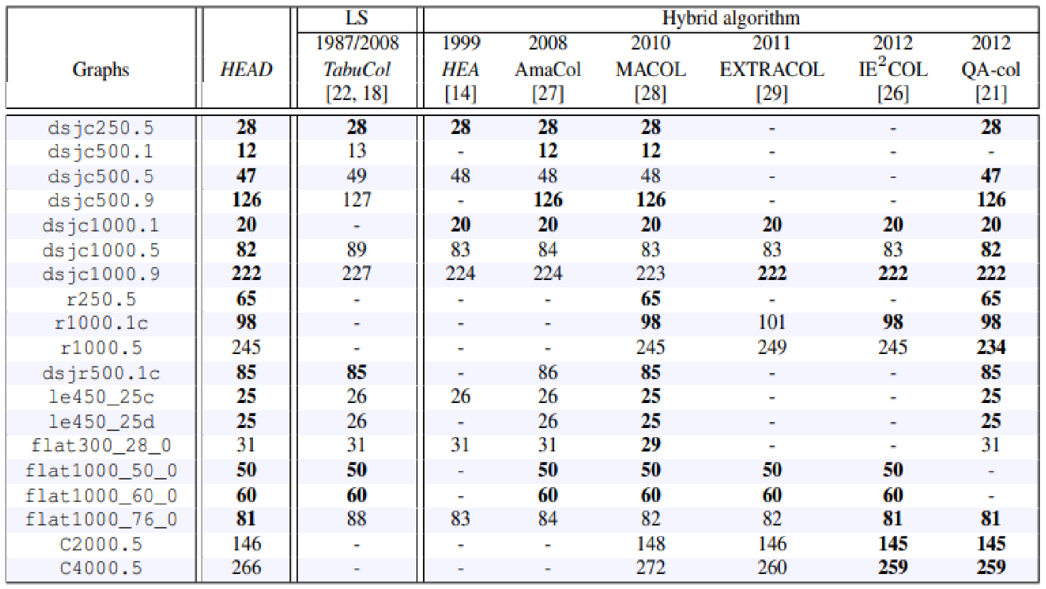
Algorithm 2: *HEAD* - second version of *HEAD* with two extra elite solutions

```
Input: $k$, the number of colors; $Iter_{TC}$, the number of TabuCol
       iterations; $Iter_{cycle} = 10$, the number of generations into one
       cycle.
Output: the best $k$-coloring found: $best$
$p_1, p_2, elite_1, elite_2, best \leftarrow$ init()    /* initialize with
random $k$-colorings */
$generation,\ cycle \leftarrow 0$
do
    $c_1 \leftarrow GPX(p_1, p_2)$
    $c_2 \leftarrow GPX(p_2, p_1)$
    $p_1 \leftarrow TabuCol(c_1, Iter_{TC})$
    $p_2 \leftarrow TabuCol(c_2, Iter_{TC})$
    $elite_1 \leftarrow$ saveBest($p_1, p_2, elite_1$) /* best $k$-coloring of
    the current cycle */
    $best \leftarrow$ saveBest($elite_1, best$)
    if $generation \% Iter_{cycle} = 0$ then
        $p_1 \leftarrow elite_2$          /* best $k$-coloring of the
        previous cycle */
        $elite_2 \leftarrow elite_1$
        $elite_1 \leftarrow$ init()
        $cycle++$
    $generation++$
while $nbConflicts(best) > 0$ and $p_1 \neq p_2$
```

Figure 12: HEAD Algorithm

4.4 Results for HEAD

Test instances are selected among the most studied graphs since the 1990s, which are known to be very difficult (the second DIMACS challenge of 1992-1993)



Graphs	*HEAD*	LS	Hybrid algorithm					
		1987/2008 *TabuCol* [22, 18]	1999 *HEA* [14]	2008 AmaCol [27]	2010 MACOL [28]	2011 EXTRACOL [29]	2012 IE^{2}COL [26]	2012 QA-col [21]
dsjc250.5	**28**	**28**	**28**	**28**	**28**	-	-	**28**
dsjc500.1	**12**	13	-	**12**	**12**	-	-	-
dsjc500.5	**47**	49	48	48	48	-	-	**47**
dsjc500.9	**126**	127	-	**126**	**126**	-	-	**126**
dsjc1000.1	**20**	-	**20**	**20**	**20**	**20**	**20**	**20**
dsjc1000.5	**82**	89	83	84	83	83	83	**82**
dsjc1000.9	**222**	227	224	224	223	**222**	**222**	**222**
r250.5	**65**	-	-	-	**65**	-	-	**65**
r1000.1c	**98**	-	-	-	**98**	101	**98**	**98**
r1000.5	245	-	-	-	245	249	245	**234**
dsjr500.1c	**85**	**85**	-	86	**85**	-	-	**85**
le450_25c	**25**	26	26	26	**25**	-	-	**25**
le450_25d	**25**	26	-	26	**25**	-	-	**25**
flat300_28_0	31	31	31	31	**29**	-	-	31
flat1000_50_0	**50**	**50**	-	**50**	**50**	**50**	**50**	-
flat1000_60_0	**60**	**60**	-	**60**	**60**	**60**	**60**	-
flat1000_76_0	**81**	88	83	84	82	82	**81**	**81**
C2000.5	146	-	-	-	148	146	**145**	**145**
C4000.5	266	-	-	-	272	260	**259**	**259**

Table 1: Best coloring found

Figure 13: Results of the HEAD

5 Our approaches

5.1 GNN for chromatic number estimation

This approach consists of a graph neural network that will output an approximation of the chromatic number of a graph given as input.

5.1.1 Training instances

It is very important to have a big balanced training set, otherwise, if most of our instances have the chromatic number 4 and some instances with chromatic number 2, 3 and 10, the neural network will tend to predict numbers close to 4 for graphs with chromatic number of 7,8, 12 for example. It would be ideal to have training instances for all the possible chromatic numbers the graphs given as input can have. To find the chromatic number of a random generated instance we used ORTools to described each graph coloring problem as a constraint satisfaction problem after which we find the chromatic number and the coloring associated using binary search.

Unfortunately,because of the complexity of generating even instances with a chromatic number of 6 taking too much,we instead generated we generated random graph instances with 20 - 60 nodes and 7.5% - 25%.We created 2 datasets ,one with generate 10^4 and another with 10^5 instances randomly which we used for training.For example,this is the distribution of the chromatic number for the larger dataset 2: 740, 3: 36158, 4: 49408,5: 13686, 6: 8 or 2: 0.007, 3: 0.3616,4: 0.4941,5: 0.1369,6: 0.0001.

5.1.2 Graph Neural Networks introduction

Recently, deep learning on graphs has emerged to one of the hottest research fields in the deep learning community. Graph Neural Networks (GNNs) aim to generalize classical deep learning concepts to irregular structured data (in contrast to images or texts) and to enable neural networks to reason about objects and their relations.

This is done by following a simple neural message passing scheme, where node features $x_v^{(l)}$ of all nodes $v \in \vartheta$ in a graph $G = (\vartheta, \varepsilon)$ are iteratively updated by aggregating localized information from their neighbors $N(\nu)$:

$$x_v^{(l+1)} = f_\theta^{(l+1)}(x_\nu^{(l)}, \{x_w^{(l)} : w \in N(\nu)\})$$

5.1.3 PyTorch Geometric (PyG) library

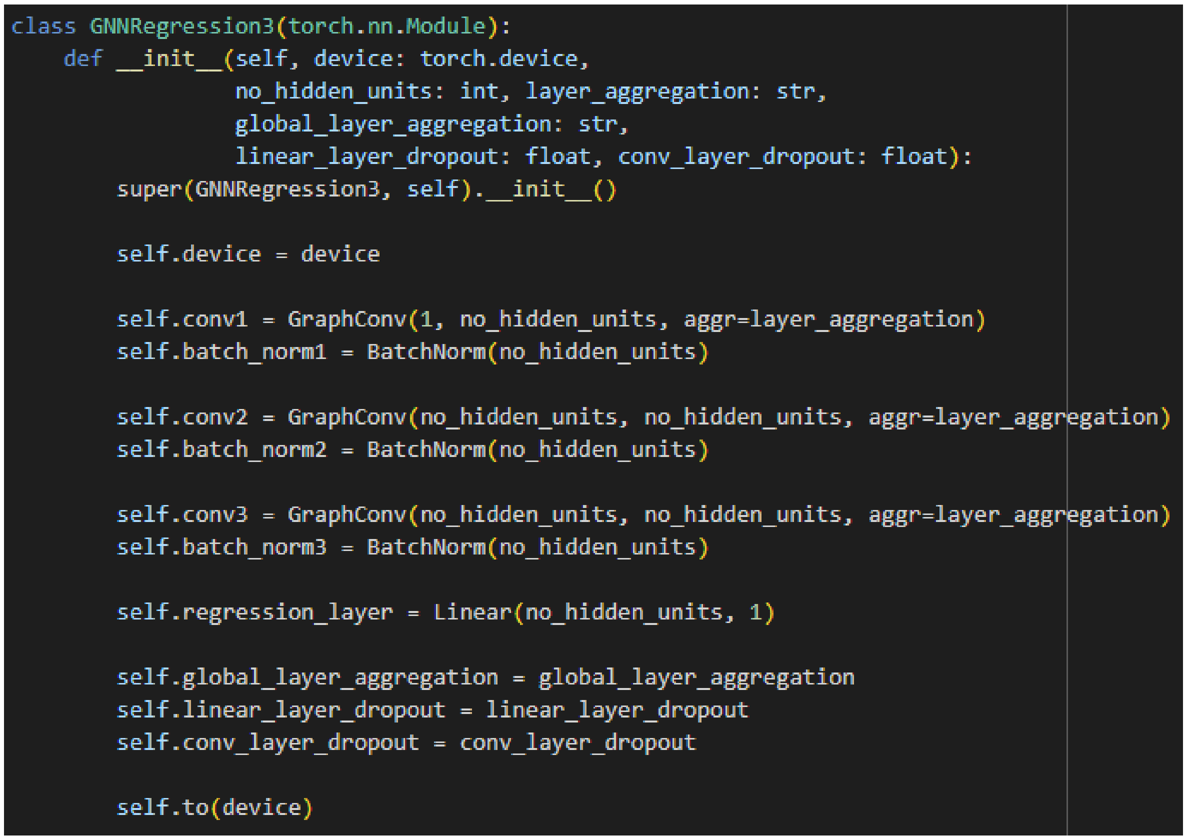
PyTorch Geometric is an extension library to the popular deep learning framework PyTorch, and consists of various methods and utilities to ease the implementation of Graph Neural Networks. Each graph in PyTorch Geometric is represented by a single Data object, which holds all the information to describe its graph representation. This data object holds 4 attributes:

1. The edge_index property holds the information about the graph connectivity, i.e., a tuple of source and destination node indices for each edge.
2. Node features as x (each of the 34 nodes is assigned a 34-dim feature vector)
3. Node labels as y (each node is assigned to exactly one class)
4. There also exists an additional attribute called train_mask, which describes for which nodes we already know their community assignments. In total, we are only aware of the ground-truth labels of 4 nodes (one for each community), and the task is to infer the community assignment for the remaining nodes.

5.1.4 Creating the GNN architecture

The GCN layer is defined as $x_v^{(l+1)} = W^{(l+1)} \sum_{w \in N(\nu) \cup \{\nu\}} \frac{1}{C_{w,\nu}} x_w^{(l)}$ where $W^{(l+1)}$ denotes a trainable weight matrix of shape [num_output_features, num_input_features] and $C_{w,\nu}$ refers to a fixed normalization coefficient for each edge. PyG implements this layer via GCNConv, which can be executed by passing in the node feature representation x and the COO graph connectivity representation edge_index.

We chose as an architecture with 3 Graph Convolutional Layers along with a global aggregation layer as well as a linear layer for regression.The reason behind this is to see how well a simple network like this performs with a small number of parameters.We also employed dropout for the linear layers and convolutional layers since we intend to have a high number of epochs to further help the aggregation of information and learning.



```
class GNNRegression3(torch.nn.Module):
    def __init__(self, device: torch.device,
                 no_hidden_units: int, layer_aggregation: str,
                 global_layer_aggregation: str,
                 linear_layer_dropout: float, conv_layer_dropout: float):
        super(GNNRegression3, self).__init__()

        self.device = device

        self.conv1 = GraphConv(1, no_hidden_units, aggr=layer_aggregation)
        self.batch_norm1 = BatchNorm(no_hidden_units)

        self.conv2 = GraphConv(no_hidden_units, no_hidden_units, aggr=layer_aggregation)
        self.batch_norm2 = BatchNorm(no_hidden_units)

        self.conv3 = GraphConv(no_hidden_units, no_hidden_units, aggr=layer_aggregation)
        self.batch_norm3 = BatchNorm(no_hidden_units)

        self.regression_layer = Linear(no_hidden_units, 1)

        self.global_layer_aggregation = global_layer_aggregation
        self.linear_layer_dropout = linear_layer_dropout
        self.conv_layer_dropout = conv_layer_dropout

        self.to(device)
```

Figure 14: GNN architecture

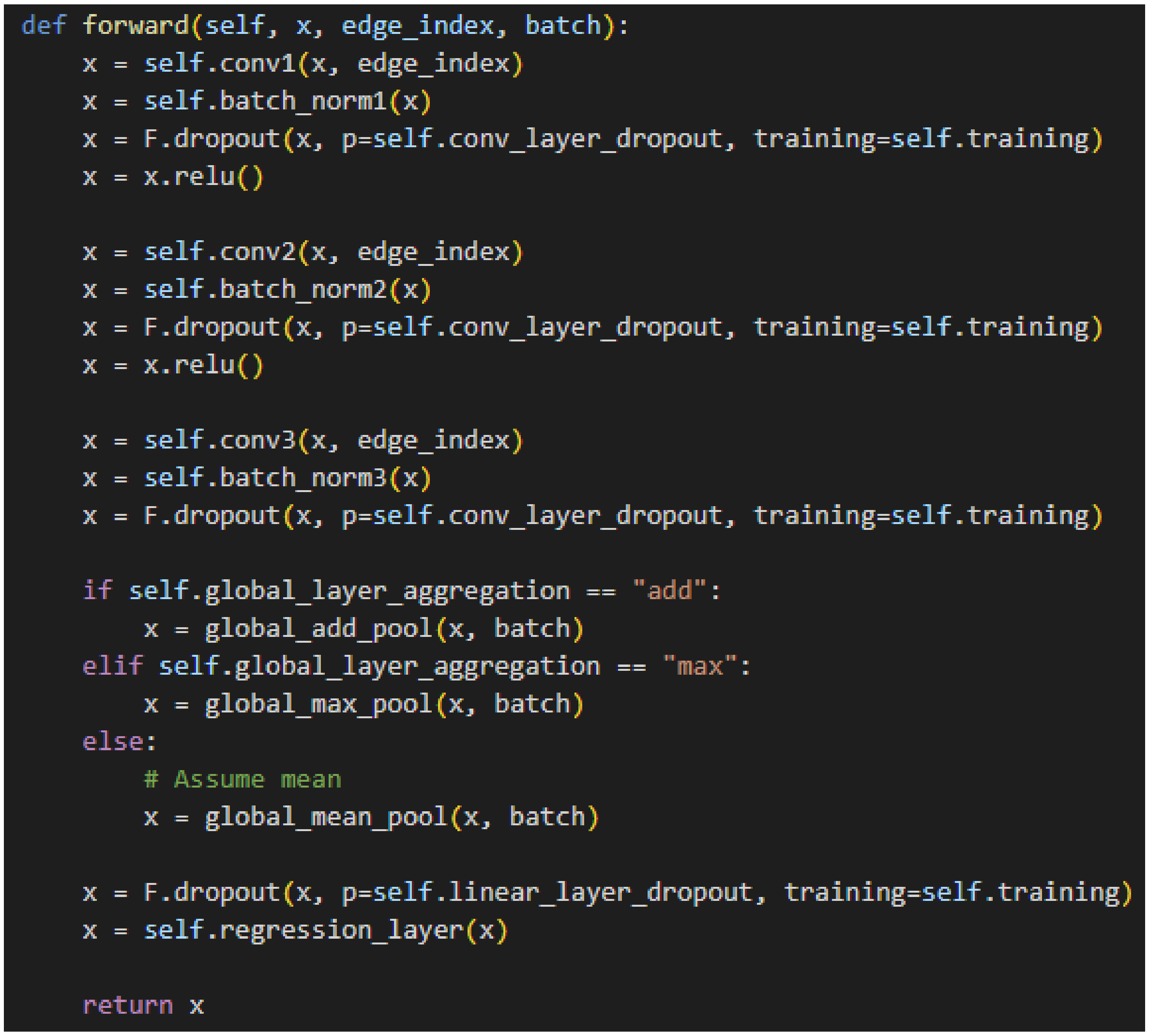

```
def forward(self, x, edge_index, batch):
    x = self.conv1(x, edge_index)
    x = self.batch_norm1(x)
    x = F.dropout(x, p=self.conv_layer_dropout, training=self.training)
    x = x.relu()

    x = self.conv2(x, edge_index)
    x = self.batch_norm2(x)
    x = F.dropout(x, p=self.conv_layer_dropout, training=self.training)
    x = x.relu()

    x = self.conv3(x, edge_index)
    x = self.batch_norm3(x)
    x = F.dropout(x, p=self.conv_layer_dropout, training=self.training)

    if self.global_layer_aggregation == "add":
        x = global_add_pool(x, batch)
    elif self.global_layer_aggregation == "max":
        x = global_max_pool(x, batch)
    else:
        # Assume mean
        x = global_mean_pool(x, batch)

    x = F.dropout(x, p=self.linear_layer_dropout, training=self.training)
    x = self.regression_layer(x)

    return x
```

Figure 15: GNN architecture

5.1.5 Mini-batching of graphs

A good idea is to batch the graphs before inputting them into a Graph Neural Network to guarantee full GPU utilization. In the image or language domain, this procedure is typically achieved by rescaling or padding each example into a set of equally-sized shapes, and examples are then grouped in an additional dimension. The length of this dimension is then equal to the number of examples grouped in a mini-batch and is typically referred to as the batch_size.

However, for GNNs the two approaches described above are either not feasible or may result in a lot of unnecessary memory consumption. Therefore, PyTorch Geometric opts for another approach to achieve parallelization across a number of examples. Here, adjacency matrices are stacked in a diagonal fashion (creating a giant graph that holds multiple isolated subgraphs), and node and target features are simply concatenated in the node dimension:

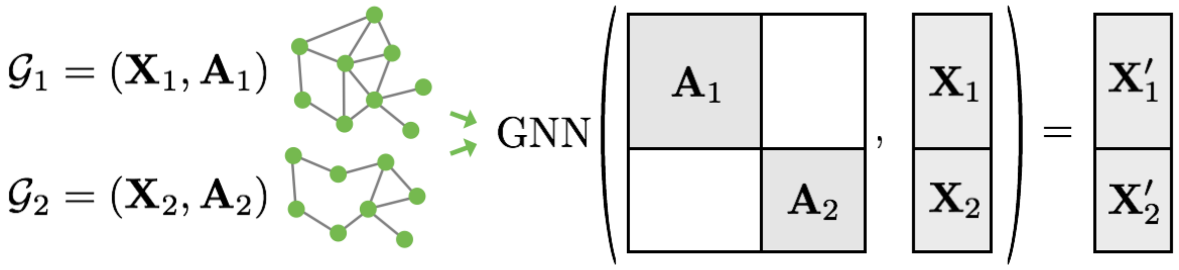

$\mathcal{G}_1 = (\mathbf{X}_1, \mathbf{A}_1)$
$\mathcal{G}_2 = (\mathbf{X}_2, \mathbf{A}_2)$
GNN
$\mathbf{A}_1$
$\mathbf{A}_2$
$\mathbf{X}_1$
$\mathbf{X}_2$
$\mathbf{X}'_1$
$\mathbf{X}'_2$

Figure 16: Graph batch

This procedure has some crucial advantages over other batching procedures:

1. GNN operators that rely on a message passing scheme do not need to be modified since messages are not exchanged between two nodes that belong to different graphs.
2. There is no computational or memory overhead since adjacency matrices are saved in a sparse fashion holding only non-zero entries, i.e., the edges.

PyTorch Geometric automatically takes care of batching multiple graphs into a single giant graph with the help of the torch_geometric.data.DataLoader class.

5.1.6 Training a Graph Neural Network (GNN)

Training a GNN for graph classification usually follows a simple recipe:

1. Embed each node by performing multiple rounds of message passing
2. Aggregate node embeddings into a unified graph embedding (readout layer)
3. Train a final classifier on the graph embedding

There exists multiple readout layers in literature, but the most common one is to simply take the average of node embeddings:

$$x_G = \frac{1}{|V|} \sum_{\nu\ inV} x_v^{(L)}$$

PyTorch Geometric provides this functionality via torch_geometric.nn.global_mean_pool, which takes in the node embeddings of all nodes in the mini-batch and the assignment vector batch to compute a graph embedding of size [batch_size, hidden_channels] for each graph in the batch.

5.1.7 Experiments and results

Instances	Size	χ	GNN Computed χ
queen5_5	25	5	6
queen6_6	36	**7**	**7**
myciel5	47	6	5
queen7_7	49	7	8
queen8_8	64	9	10
huck	74	11	5
jean	80	10	4
queen9_9	81	10	11
david	87	11	5
myciel6	95	**7**	**7**
queen8_12	96	12	13
games120	120	9	5
anna	138	11	5
queen13_13	169	13	19
myciel7	191	8	10
homer	561	13	4

Table 4: Chromatic number by GNN obtained for 100k dataset

As we can see from the table,only for the instances queen6_6 and myciel7 is able to predict the chromatic number and for the instances queen5_5,myciel5,queen7_7,queen8_8,queen9_9, queen8_12 it is 1 point away from the chromatic number.Considering the fac that the network has a simple architecture and was trained on an unbalanced dataset with random instances with a chromatic number between $[2, 5]$ this shows that GNN can generalize very well on harder and unseen instances.

We also tried fine tuning based on a validation dataset for minimizing the validation loss but this lead to overfitting quickly(in the first few epochs) resulting in models that always print values in the range $[3, 5]$.

Another tried idea was to create a more balanced dataset by taking an unbalanced dataset and adding for certain instances a clique along with a random number of edges in order to force the chromatic number to be higher.Interestingly,after balancing the dataset presented earlier in this way,the results were even worse giving negative values or too high values for the test instances,which would mean that simply adding a clique and a few random edges doesn't characterize the more difficult instances.

5.2 Improved PSO Algorithm (PSO-AWDV)

In this study [1], the authors propose an enhanced version of Particle Swarm Optimization (PSO) named Adaptive Weighted Delay Velocity PSO (PSO-AWDV). The research introduces a novel approach by incorporating weighted delay velocity into a new PSO variant termed PSO with Weighted Delay Velocity (PSO-WDV). Following this, the authors develop the Adaptive PSO-AWDV algorithm, designed to dynamically update the velocity inertia weight. This adaptive approach leverages a unique estimation method that evaluates the evolutionary state of the particle swarm. Through comprehensive testing on well-known benchmark functions, the authors demonstrate that their proposed Adaptive PSO-AWDV algorithm outperforms several established PSO variants and intelligent optimization algorithms documented in the literature. The new update equations for velocity and position are:

$$\begin{gathered} v_i = wv_i + (1-w)v_{i-1} + c_1 r_1 (pbest_i - p_i) + \\ c_2 r_2 (gbest - p_i) \end{gathered} \tag{10}$$

$$p_i = p_i + v_i \tag{11}$$

Here, the only difference from standard PSO version is the term v_{i-1}, multiplied by $(1-w)$ which is the delayed velocity. The new adaptive scheme is developed according to the estimation of the evolutionary state of the particle swarm, thereby a new PSO with adaptive weighted delay velocity (PSO-AWDV) is presented on the basis of the updating functions:

$$c_1 = (c_{1i} - c_{1f}) \times \frac{k_{max} - k}{k_{max}} + c_{1f} \tag{12}$$

$$c_2 = (c_{2i} - c_{2f}) \times \frac{k_{max} - k}{k_{max}} + c_{2f} \tag{13}$$

where c_{1i} and c_{1f} indicate, respectively, the initial and final values of the acceleration factor c_1; likewise, the initial and final values of the acceleration factor c_2 are represented by c_{2i} and c_{2f}, respectively; k and k_{max} denote, respectively, the current and maximal iterations in the optimization. In particular, the inertia weight of velocity is adaptively regulated in accordance with the evolutionary state in the optimization, and it is calculated as follows:

$$w = 1 - \frac{a}{1 + e^{b \cdot E(k)}} \tag{14}$$

where a and b are two parameters that can be designed to adjust the search performance of PSO; $E(k)$is the estimation value(EV) of the evolutionary state at the k-th iteration and can be computed as follows:

$$E(k) = \frac{|f_{max}(k)| - |f_{min}(k)|}{|f_{max}(k)|} \tag{15}$$

where f_{max} and f_{min} stand for, respectively, the maximal and minimal fitness values of the particles at the k-th iteration. In sum,the PSO-AWDV algorithm pseudocode is illustrated in **Algorithm 2**.

The particle swarm optimization (PSO) algorithm was adapted for the graph coloring problem by modifying the position and velocity update rules. A particle's position represents the colors assigned to the nodes in the graph. To ensure the correctness of a solution, the PSO was modified such that every position of a particle is a list with the

Algorithm 2 PSO-AWDV Algorithm

```
Initialize the parameters of the PSO-AWDV algorithm, including maximum iteration $k_{\max}$, parameters of the inertia weight $a$ and $b$, parameters of the acceleration factor $c_{1i}$, $c_{1f}$, $c_{2i}$, and $c_{2f}$;
repeat
    Evaluate the fitness of every particle of the swarm and estimate the evolutionary state $E(k)$ of the particle swarm at the current iteration according to Equation (8);
    Compare the fitness value of each particle with its pbest and gbest of the swarm, and select the corresponding better particle to replace the original one;
    Compute the acceleration factors and inertia weight according to Equations (5) - (7);
    Update the velocity and position vectors of the particles at the current iteration according to Equations (3) and (4);
until Maximum iteration
```

length equal to number of nodes containing random integer between 0 and the number of colors - 1 (representing the encoding of the colors). The velocities were clipped between a specific range to prevent them from becoming too large or too small. The positions were rounded and converted to integers in each iteration to ensure that they represented valid color assignments.

The number of iterations was set to 2500 for small graph coloring instances and 5000 for larger graph coloring instances. This number of iterations was chosen based on empirical testing, which showed that it was sufficient to find good solutions for most graphs.

In addition to the modifications mentioned above, the PSO algorithm was also implemented with a number of other parameters, including the the initial and final values for acceleration factors

$$c_{1i} = c_{2i} = 1.8$$

$$c_{1f} = c_{2f} = 1.2$$

and the specific factors used for the evolutionary state calculation

$$a = 0.8, b = 0.5$$

The configuration of the parameters was chosen to minimize the number of adjacent nodes having the same color.

The fitness function consists of the count of the violated coloring restrictions. It should be minimized and a perfect graph coloring should have a fitness function equal to 0.

References

[1] Jitendra Agrawal and Shikha Agrawal. “Acceleration based particle swarm optimization for graph coloring problem”. In: *Procedia Computer Science* 60 (2015), pp. 714–721.

[2] Russell Eberhart and James Kennedy. “A new optimizer using particle swarm theory”. In: *MHS’95. Proceedings of the sixth international symposium on micro machine and human science*. Ieee. 1995, pp. 39–43.

[3] Palash Goyal and Emilio Ferrara. “Graph embedding techniques, applications, and performance: A survey”. In: *Knowledge-Based Systems* 151 (2018), pp. 78–94.

[4] Henrique Lemos et al. “Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems”. In: *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2019, pp. 879–885.

[5] Laurent Moalic and Alexandre Gondran. “Variations on memetic algorithms for graph coloring problems”. In: *Journal of Heuristics* 24 (2018), pp. 1–24.

[6] Yangming Zhou, Jin-Kao Hao, and Béatrice Duval. “Reinforcement learning based local search for grouping problems: A case study on graph coloring”. In: *Expert Systems with Applications* 64 (2016), pp. 412–422.