Chasing the Perfect Hue: A High-Performance Dive into Graph Coloring

Tommaso Crippa, Alessandro Ruzza, Elizabeth V. Koleva, Dimitar G. Penkov

28 February 2025

Abstract

Graph coloring is a fundamental problem in combinatorial optimization with applications in scheduling, register allocation, and parallel computing [4]. The goal of this project is to determine the **chromatic number** of a given graph using a **Branch-and-Bound framework**, enhanced with heuristic-based bounding strategies.

To efficiently explore the solution space, we integrated advanced upper bound **coloring heuristics** and lower bound **maximum clique heuristics**, coupled with efficient **branching strategies**. To tackle the computational complexity of the problem, we implemented a **parallel** version of the solver using MPI, distributing the B&B search tree across multiple nodes in a manager-worker pattern.

The project was developed as part of the **EuroHPC Summit Student Challenge 2025**, leveraging the computational power of the VEGA supercomputer to achieve scalability and performance improvements.

Contents

1 Problem Description

1.1 Background

Graph coloring is the process of assigning distinct colors to each vertex of a graph $G = (V, E)$ such that no two adjacent vertices share the same color. A graph G is k-colorable if its vertices can be colored using at most k different colors while satisfying this condition. The smallest k for which G is k-colorable is called the **chromatic number** of G, denoted by $\chi(G)$.
A **clique** in a graph G is a subset of vertices where every pair of vertices is connected by an edge. The **clique number** $\omega(G)$ is the size of the largest clique in G. Since each vertex in a clique must be assigned a unique color in any valid graph coloring, the relationship $\omega(G) \leq \chi(G)$ always holds.

1.2 Project Objective

The objective of this project is to develop an algorithm that finds the chromatic number of a graph G using a **Branch-and-Bound framework**. The algorithm will utilize heuristic methods to determine bounds on the chromatic number:

- **Lower Bound**: Implement a heuristic to find the maximum clique in G. The cardinality of this maximum clique, denoted by $lb(\omega(G))$, serves as a lower bound on the clique number $\omega(G)$ and thus the chromatic number $\chi(G)$, i.e., $lb(\omega(G)) \leq \omega(G) \leq \chi(G)$.
- **Upper Bound**: Implement a heuristic to find a valid coloring of G. The number of colors used in this coloring, denoted by $ub(\chi(G))$, provides an upper bound on the chromatic number $\chi(G)$, i.e., $\chi(G) \leq ub(\chi(G))$

The branch and bound algorithm will start from the root node, computing the initial $lb(\omega(G))$ and $ub(\chi(G))$, then if $lb(\omega(G))$ and $ub(\chi(G))$ do not coincide, identify two non-adjacent vertices in the graph and create two branches:

- **First Branch**: assume the two vertices are assigned the same color.

- **Second Branch**: assume the two vertices are assigned different colors

Then continue branching and updating the bounds until the chromatic number of $\chi(G)$ is found.

2 Methodology

This section presents the main data structures and algorithms used in our approach.

2.1 Main Data Structures

2.1.1 UnionFind

The Union-Find data structure is used to efficiently track groups of vertices that must share the **same color**. Each vertex initially represents its own color class. When two vertices are merged in the Union-Find structure, they are forced to have the same color, ensuring consistency in the coloring process. This is useful in the Branch and Bound (B&B) approach, where constraints dynamically emerge during the search. By using path compression and union by rank, Union-Find allows efficient merging of color equivalence classes and quick lookups to verify if two vertices must share the same color.

2.1.2 AddedEdges

The AddedEdges data structure is a list of vertex pairs (n_1, n_2) that must be assigned **different colors**.

2.1.3 B&BNode

Each node in the Branch and Bound search tree represents a **partial coloring state**, maintaining information about coloring constraints (using a UnionFind object and an AddedEdges list).
Branching the node yields 2 child nodes, one imposing the same color on 2 vertices, the other imposing different colors on the same 2 vertices.

2.2 Branch and Bound Framework

2.2.1 Sequential Framework

To create an initial working algorithm, we started by creating a sequential B&B algorithm that would **incrementally** explores partial colorings using the Union-Find structure to track color equivalence classes and the AddedEdges list to enforce color separation constraints. By utilizing simple branching strategies, this method ensures a **systematic traversal** of potential solutions but suffers from the inherent **computational limitations** of a single-threaded approach, deteriorating in performance as the graph complexity increases.

2.2.2 Early Parallelization Independent Search

The first parallel implementation of B&B introduces task parallelism by allowing **multiple MPI ranks** to create nodes and search for solutions **independently**. The problem behind this architecture is that each rank generated and processed subproblems without coordinating with others. While this approach allowed for some degree of parallel exploration, its **lack of a centralized mechanism** to distribute work dynamically caused uneven workload.

Furthermore, since each rank worked independently, they lacked information on which parts of the solution space were already explored, leading to **inefficient resource utilization**. As a consequence, scalability was severely limited, and adding more ranks did not lead to proportional performance improvements.

2.2.3 Load Balancing the architecture: Manager/Worker Model

To overcome these inefficienties, we adopted a **manager/worker model**, which dynamically distributes tasks based on workload demand, ensuring that idle workers receive new subproblems to explore, leading to better parallel efficiency and reduced idle time. The central manager (rank 0) is responsible for **task allocation**, assigning B&B nodes to workers that process them in parallel. Once a worker completes a task, it returns the generated child nodes to the manager, ensuring continuous workload distribution. This model enables dynamic **load balancing**, as idle workers receive new tasks in real time, preventing bottlenecks and significantly improving parallel efficiency.

2.3 Upper Bound: Graph Coloring Heuristics

2.3.1 Starting Heuristic: DSatur

To compute the upper bound, we started to implement a basic but easily understandable heuristic to get used to the problem. Therefore we developed **DSatur**, a greedy heuristic that assigns colors to vertices iteratively, prioritizing the most constrained vertex (i.e., the vertex with the highest saturation degree, the number of distinct colors in its neighborhood) [1]. This heuristic provides a rapid estimation of an upper bound for the chromatic number and serves as an effective initialization step for more advanced coloring methods. Although DSatur performs well on many instances, its reliance on greedy decisions can lead to **suboptimal solutions**, particularly in complex graph structures where a more global approach is beneficial.

2.3.2 Improving the heuristic: BackTracking DSatur

To improve upon DSatur, we implemented **BackTracking DSatur**, which incorporates a backtracking mechanism to improve the initial greedy coloring when a conflict or inefficiency is detected [2]. By allowing the algorithm to **reconsider previous choices**, this approach balances the efficiency of DSatur with a deeper exploration of the solution space, leading to results that are significantly closer to the optimal chromatic number.The ability to backtrack prevents the heuristic from getting trapped in local optima and ensures a more **robust** coloring strategy.
This heuristic proved to be our best performing one to compute the upper bound, therefore we implemented further enhancements through **intra-node parallelization**, where each worker executes multiple instances of Backtracking DSatur concurrently using multiple threads. To increase diversity in the search process, a degree of randomization is introduced, ensuring that each execution follows a slightly different path. This parallel approach leverages VEGA's **multi-core architectures** to efficiently explore a wider range of potential solutions, leading to improved coloring performance across various graph instances.

2.3.3 Failed Experiment: Tabu Search

In our pursuit of an effective upper bound heuristic, we explored multiple strategies beyond DSatur-based approaches. Among these, **Tabu Search** stood out as a promising candidate. The key idea behind Tabu Search is to iteratively refine a solution while preventing the search from cycling back into previously explored state utilizing a Tabu list, managing to escape local optima.

We tested both a stand-alone and a parallel version of Tabu Search. The parallel version, similar to previous approaches, executed multiple instances of the algorithm at the same time. However, neither of them surpassed in effectiveness our previous solvers.
The primary issue was the **high computational cost** of this algorithm. steps such as neighbor evaluation and tabu list checks significantly increased the time required to complete the coloring. Since the Branch and Bound framework we utilized expected fast solutions for each node, Tabu Search was ultimately less effective than BackTracking DSatur.

2.4 Lower Bound: Max Clique Heuristics

2.4.1 Basic heuristics: Greedy Solutions

To compute the lower bound, we began by approximating the maximum clique using a **greedy** heuristic. These methods prioritize selecting vertices with high connectivity, iteratively constructing a clique by adding the most promising candidates until no more vertices can be included, resulting in a maximal, though not necessarily maximum, clique. While this method is **computationally efficient**, its effectiveness is highly dependent on the structure of the graph, often leading to **suboptimal** solutions.

2.4.2 Moving to more Dynamic solvers: DLS

To overcome the limitations of purely greedy approaches, we reviewed some of the state-of-the-art techniques to find the maximum clique [5] and experimented with some of them to introduce dynamic decision-making when computing, reducing the likelihood of getting stuck in local optima.
One of the most promising ones we introduced was **Delayed Local Search (DLS)** [3]. Instead of making immediate, deterministic choices, DLS employs **penalty mechanisms** to iteratively refine a candidate clique over multiple iterations, allowing for an escape from local optima. The core idea behind DLS is to initially construct a clique, then **iteratively perturb and refine it** by selectively penalizing frequently chosen vertices. This discourages the algorithm from repeatedly selecting suboptimal configurations and promotes exploration of alternative clique formations.
We then refined this strategy by incorporating information from the current node, such as vertices with equal or different colors by incorporating the algorithm with the **UnionFind** and **AddedEdges** data structures. By maintaining efficient connectivity updates, this heuristic avoids redundant checks and speeds up the clique expansion process. Such methods improve upon the basic greedy approach, yet they still struggle with highly structured graphs where more advanced heuristics are required. This variation of DLS, although more **computationally intensive**, leverages these additional properties to improved performance as we progress deeper into the Branch and Bound tree.

2.4.3 Advanced Techniques: Evolving Strategies

To further improve flexibility and robustness, we experimented with more evolving strategies that introduce adaptivity and dynamic decision-making when computing, adjusting their behavior based on real-time search progress.
One such approach is an **adaptive** version of DLS, which acts as a standard DLS strategy during the early stage, when coloring information is still insufficient for guiding clique selection; as new coloring constraints are introduced, the algorithm **dynamically shifts** to the color-aware strategy. This hybrid approach ensures that the search remains efficient at different stages of the computation, optimizing both **speed and solution quality**.

Another extension involves dynamic **penalty tuning**, where the penalty mechanism itself evolves throughout the search. In this variation, if the algorithm fails to find an improved clique after a predefined number of iterations, the penalty delay **increases**, allowing for a broader exploration of the search space.
Finally, as we did with the coloring heuristic, our final version opted to run multiple instances of DLS simultaneously, each with slightly different hyperparameters sampled from a **Poisson distribution**. By distributing the search across multiple processing units with different starting positions and strategies, the algorithm can effectively explore different parts of the solution space using **intra-node parallelization**, significantly increasing the probability of discovering larger cliques.

2.5 Branching Strategies

Branching is a **critical** component of the Branch-and-Bound framework, determining how the search tree is explored and influencing both efficiency and pruning effectiveness. An ideal branching strategy prioritizes decisions that lead to the fastest convergence, minimizing unnecessary expansions while maintaining the ability to explore promising areas of the solution space.

2.5.1 Random Selection

Our first implementation randomly selected non-adjacent vertices without incorporating any additional **structural information**. This strategy was computationally inexpensive, but the general framework suffered from these ineffective selections which didn't help detecting the chromatic number.

2.5.2 Structural-based Selection

An important step forward was made when we switched the branching strategy to one which prioritizes vertices based on their connectivity degree, ensuring that **highly connected vertices** are processed first. This heuristic is particularly effective in dense graphs, where branching on high-degree vertices tends to maximize constraints early in the search, reducing the number of possible extensions.
We further improved this selection by taking inspiration from the **DSatur** algorithm. Instead of statically selecting the most connected vertices, we choose the ones with the **highest saturation degree**. This algorithm attempts to branch on the most constrained choices first, leading to more effective pruning and reducing the search depth required to reach an optimal solution.

2.5.3 Failed Experiment: Hybrid Branching Strategy

We also tried adopting a hybrid strategy that adapts its selection criteria based on the depth of the current node inside the tree. The intuition behind this method is that the effectiveness of a branching heuristic depends on the phase of the search, and switching heuristics at key transition points can improve overall efficiency. In the early stages of the search it would use **clique-based selection**, where branching decisions are made based on the structure of the maximum clique. Then in the mid-stage, the strategy shifts to a **conflict-driven selection**, focusing on vertices with high connectivity and saturation. Finally, in the late stages, the **lowest available saturation vertices** are prioritized to resolve the remaining conflicts. While in theory this strategy combined the strengths of the previous heuristics, in practice it did not perform as well as the others, we therefore utilized the **saturation-based** one to compute our final results.

3 Implementation Details

This section presents the practical aspects of our implementation. We focus on the more intricate components, as the fundamental concepts have already been introduced in previous sections. Here, we provide insights into optimizations, algorithmic refinements, and technical considerations that enhance the efficiency and scalability of our approach.

3.1 UnionFind

Efficiently tracks vertex equivalence classes, ensuring that merged vertices share the same color. The class maintains a list, `self.parent`, where each vertex initially points to itself.

The `find` operation retrieves the representative of the equivalence class of a vertex, using path compression to flatten the structure and speed up future lookups.

The `union` operation merges two equivalence classes by updating the representatives, effectively merging the two color classes.

3.2 Branch and Bound Algorithm

The parallel branch-and-bound algorithm is implemented using MPI to distribute the search process across multiple ranks. It follows these steps:

a. The **manager process** (rank 0) initializes the search by computing an initial **lower bound** using the **maximum clique heuristic** and an **upper bound** using DSatur.

b. The manager maintains a **queue of branch-and-bound nodes** and assigns work to available worker ranks.

c. **Workers execute branching in parallel**:

   (a) Select a pair of non-adjacent vertices according to the Branching heuristic and create two branches:

      i. **Same color branch**: Merge vertices in UnionFind.

      ii. **Different color branch**: Add an edge in AddedEdges to enforce color separation.

   (b) Compute updated bounds for each new node and return them to the manager.

d. The manager **prunes unpromising branches** using upper bounds and redistributes new nodes to workers dynamically to balance the workload.

e. The process continues until a time limit is reached, or the optimal solution is found.

4 Results

4.1 Trend graphs

Figure 1 illustrates the relationship between the size of the problem as the number of vertices inside the graph and the time required to find a solution. As expected, larger instances tend to require more time to solve. However, the data shows some **variability**, like `homer.col`, whose graph has more than 500 vertices, but due to its mostly sparse connection the time it took to find the optimal coloring was less than 20 seconds. This suggests that factors beyond vertex count, such as **graph density** or **structural properties**, also influence computational difficulty.

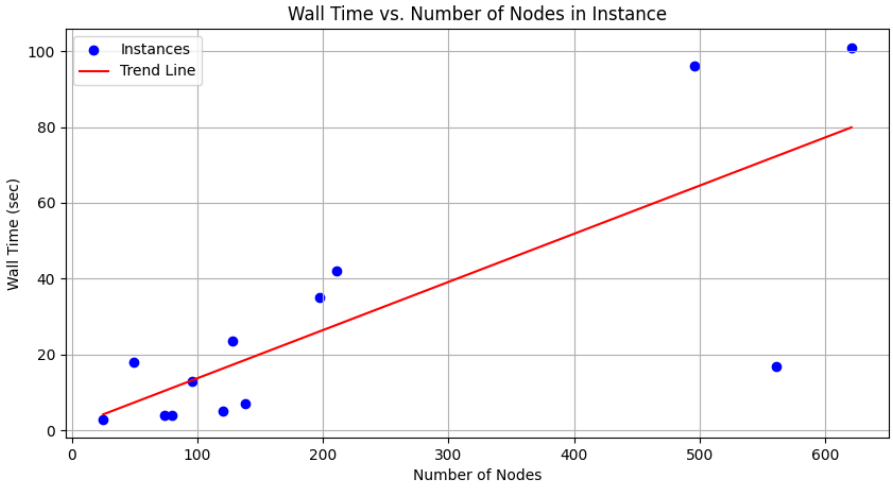

Figure 1: Wall time w.r.t. problem size

Figure 2 examines the impact of computational resources on solution time. The downward trend indicates that increasing computational resources generally reduces wall time, confirming the **effectiveness of parallelization**. However, there are some deviations from the trend line, implying that certain instances do not scale as efficiently with additional resources. This could be due to synchronization **overhead** or inherent algorithmic limitations.
Both graphs only include instances that were solved within the time limit, meaning that the observed trends are limited to feasible computations. Instances that exceeded the time limit may exhibit different behaviors, particularly for large problem sizes.

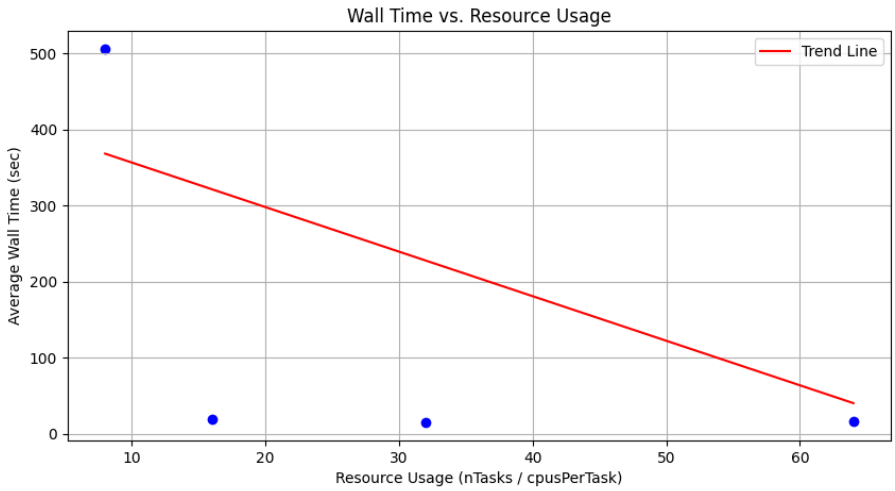

Figure 2: Wall time w.r.t. resources used

4.2 Result table

The following table summarizes our results on every benchmark instance.
We solved **16/30 instances to proven optimality** (Upper Bound = Lower Bound).
We also obtained **tight bounds** for all other instances within the time limit.
We conducted extensive experiments on a diverse set of benchmark graph instances, summarized in Table 1. The tested instances include structured graphs such as Mycielski and Queen graphs, as well as various real-world and synthetic graphs. Out of the **30** total instances we tested on, we successfully solved **16** optimally. The chromatic number $\chi(G)$ can be identified as the first value of the **(UB, LB)** column.

Instance	Vertices, Edges	(UB, LB)	Optimum	Wall Time	Tasks	CPUs/Task
anna.col	138, 986	(11, 11)	Yes	6 sec	32	32
david.col	87, 812	(11, 11)	Yes	4 sec	32	32
fpsol2.i.1.col	496, 23308	(65, 65)	Yes	89 sec	32	32
games120.col	120, 1276	(9, 9)	Yes	6 sec	32	32
homer.col	561, 3256	(13, 13)	Yes	17 sec	32	32
huck.col	74, 602	(11, 11)	Yes	3 sec	32	32
inithx.i.3.col	621, 27938	(31, 31)	Yes	78 sec	32	32
jean.col	80, 508	(10, 10)	Yes	4 sec	32	32
le450_5b.col	450, 11468	(6, 5)	Unknown	9925 sec	32	32
miles1500.col	128, 10396	(73, 73)	Yes	41 sec	32	32
miles250.col	128, 774	(8, 8)	Yes	6 sec	32	32
mulsol.i.1.col	197, 7850	(49, 49)	Yes	31 sec	32	32
myciel3.col	11, 40	(4, 2)	Unknown	9952 sec	32	32
myciel4.col	23, 142	(5, 2)	Unknown	9902 sec	32	32
myciel5.col	47, 472	(6, 2)	Unknown	9908 sec	32	32
myciel7.col	191, 4720	(8, 2)	Unknown	9913 sec	32	32
queen10_10.col	100, 2940	(12, 10)	Unknown	9908 sec	32	32
queen11_11.col	121, 3960	(13, 11)	Unknown	9905 sec	32	32
queen12_12.col	144, 5192	(14, 12)	Unknown	9912 sec	32	32
queen13_13.col	169, 6656	(15, 13)	Unknown	9907 sec	32	32
queen14_14.col	196, 8372	(17, 14)	Unknown	9912 sec	32	32
queen15_15.col	225, 10360	(18, 15)	Unknown	9910 sec	32	32
queen5_5.col	25, 320	(5, 5)	Yes	2 sec	32	32
queen6_6.col	36, 580	(7, 6)	Unknown	9901 sec	32	32
queen7_7.col	49, 952	(7, 7)	Yes	18 sec	32	32
queen8_12.col	96, 2736	(12, 12)	Yes	13 sec	32	32
queen8_8.col	64, 1456	(9, 8)	Unknown	9903 sec	32	32
queen9_9.col	81, 2112	(10, 9)	Unknown	9906 sec	32	32
school1.col	385, 38190	(14, 14)	Yes	1986 sec	32	32
zeroin.i.1.col	211, 8200	(49, 49)	Yes	33 sec	32	32

Table 1: Summary of Graph Coloring Results

4.3 Performance Analysis

Our algorithm successfully computed the optimal solution for several instances, particularly those with moderate size and edge density. For instances such as `anna.col`, `david.col`, and `jean.col`, the algorithm efficiently found the correct chromatic number within a few seconds. This demonstrates that our approach is **effective on small to medium-sized graphs** where the branching and bounding process can sufficiently constrain the search space.

The **Mycielski graphs** (`myciel3.col` to `myciel7.col`) consistently produced upper and lower bounds that **did not match**, due to their graph structure: these types of instances are **triangle-free**, therefore the maximum clique $\omega(G)$ is always equal to 2, but the chromatic number $\chi(G)$ can have values **strictly greater** than our best lower bound, preventing the solver from confirming the chromatic number as **optimal**. Despite this, the computed final upper bound for these instances can be proven to be correct.

The **Queen graphs**, especially the larger instances (e.g., `queen10_10.col` and greater sizes) posed additional difficulties, with all cases exceeding the computational time limit. These graphs exhibit highly structured constraints, where our branching strategies struggle to effectively reduce the search space. Therefore, our algorithm did not manage to find an optimal solution for them.

Finally, leveraging the computational power of the **VEGA** supercomputer and our **parallelization strategies**, we also efficiently solved larger instances (e.g., `miles1500.col`, `school1.col`, `zeroin.i.1.col`) within a remarkably short time.
These results indicate that while our approach is **effective** for a wide range of instances, further enhancements can be done to handle structured **constraint-heavy** graphs.

5 Acknowledgements

We would like to thank the **EuroHPC Joint Undertaking** for providing this opportunity.
We are also grateful to the EuroHPC Challenge supervisors, **Régis Beck, Janez Povh, Roman Kužel** for their invaluable support and guidance.

A special thanks to our mentor, **Mirko Rahn**, for his patience, feedback, and continuous guidance throughout the challenge. His expertise and encouragement have greatly contributed to improving our approach and addressing difficulties.

We also extend our gratitude to our universities, for their contributions and support:

- **Politecnico di Milano**
- **Sofia University "St. Kliment Ohridski"**
- **Università della Svizzera Italiana**
- **University of Luxembourg**
- **Universitat Politècnica de Catalunya**

References

[1] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 1979.

[2] R.M.R. Lewis. *Guide to Graph Colouring: Algorithms, Applications and Implementations*. Springer, 2nd edition, 2021.

[3] Wayne Pullan and Holger H. Hoos. Dynamic local search for the maximum clique problem. *Journal of Artificial Intelligence Research*, 2006. Available at jair.org.

[4] Ünal Ufuktepe and Goksen Bacak Turan. Applications of graph coloring. pages 522–528, 05 2005.

[5] Qinghua Wu and Jin-Kao Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, 2015. hal-02709508.