Stockholms universitet

Graph coloring using modular decomposition
Graf färgning med modular decomposition

Emanuel Berggren

Handledare: Marc Hellmuth
Examinator: Lars Arvestad
Inlämningsdatum: Friday 29th December, 2023

Kandidatuppsats i datalogi
Bachelor Thesis in Computer Science

Abstract

Graph coloring is one of the most common and studied problems in computer science. It has applications in many different areas, such as for scheduling, compiler optimizations, and biological networks, but being NP-Hard means that optimal solutions are often too computationally expensive to use in practice, and in turn that many instances require the use of heuristics. This thesis examines a heuristic based on the modular decomposition of a graph, a way to split the graph into well-structured subgraphs called modules, where some parts can be colored optimally in linear time, and some parts require other heuristics. As we will show, this strategy of coloring with the modular decomposition was found to give an increase in performance for one of the tested heuristics, and no increase for the others.

Sammanfattning

Graf färgning är ett av de vanligaste och mest efterforskade problemen i datavetenskapen. Det har många användningsområden, till exempel för schemaläggning och kompilatoroptimeringar, men eftersom problemet är NP-fullständigt så är optimala lösningar oftast för långsamma för att använda i praktiken. Detta innebär att många applikationer behöver använda sig av heuristiker istället. Den här uppsatsen undersöker en heuristik baserad på en grafs "modular decomposition". Det är ett sätt att dela upp en graf i subgrafer ,"modules", där vissa modules har en speciell struktur som låter dem färgas optimalt i linjär tid, medan andra kräver heuristiker. Denna metod gav bättre färgningar i kombination med en av de testade heuristikerna, och ingen skillnad för de andra.

Contents

1 Introduction

In this thesis, we investigate a new heuristic for coloring graphs using the modular decomposition of a graph, that is used alongside other traditional graph coloring heuristics.
The modular decomposition of a graph describes the structure of the graph, by recursively splitting it into distinct modules. A module is a set of vertices in a graph, which share a common neighbourhood of vertices among vertices outside of the module. Modules also form a hierarchy, which means that modules can be further subdivided into smaller modules. This representation of a graph can be described with a rooted tree, where every vertex has one of three possible labels describing how its child modules can be combined to form the graph induced by its parent module. There are 3 specific types of modules, series, parallel, and prime, which are defined later.
Modular decomposition allows for coloring of graphs in linear time with the minimum required colors if all of the modules are either series or parallel [13]. However, if any of the vertices are prime then optimal graph coloring is in general NP-Hard [8]. The question examined here, is if one can still utilize the modular decomposition, where graph coloring heuristics are only applied on the prime modules. The modular decomposition might still color some parts of the graph optimally, and the structure it provides might provide a hint for how to apply the heuristics on the prime parts, improving performance for other heuristics.
In section 2 all of the required terminology and definitions are provided, and is split into two parts, subsection 2.1 has definitions that might be familiar to most people that have worked with graphs, and subsection 2.2 provide the definitions that are more specific to this thesis.
section 3 introduces the algorithm that forms the basis for the new coloring heuristics. This base algorithm is combined with a regular graph coloring heuristic and a strategy to create a new heuristic.
In section 4 so are the graph coloring heuristics used described.
The combination strategies are described in section 5. They determine how

the heuristics are applied, and how the structure of the modular decomposition is utilised.
In section 6 so are the test graphs and benchmarking methods described. The data used is both from standard DIMACS benchmarks [3], and custom generated data.
Finally, the way the data is evaluated and the results are presented in section 7, respectively section 8

2 Definitions

2.1 Graph basics

The definitions used here are based on the definitions in [16].

Definition 2.1 (Graph). *A graph $G = (V, E)$ is a tuple, where V is the set of vertices, and E is a set of unordered pairs of distinct vertices in V.*

Definition 2.2 (Neighbour). *For a graph $G = (V, E)$, we say that $v \in V$ is adjacent to $u \in V$ if $(v, u) \in E$.*
The neighbourhood $N_G(v)$ of a vertex $v \in V$ in a graph $G = (V, E)$, is the set of vertices that are adjacent to v, that is $N_G(v) = \{u : (u, v) \in E\}$. If $u \in N_G(v)$, we also say that u and v are neighbours.

Definition 2.3 (Degree). *The degree of a vertex $v \in V$ in a graph $G = (V, E)$, denoted by $deg(v)$, is the number of neighbours of v in G, that is $deg(v) = |N_G(v)|$.*

Theorem 2.1 (Handshake lemma). *[16] For a graph $G = (V, E)$, the sum of the degrees of every vertex twice is the number of edges, that is $\sum_{v \in V} deg(v) = 2|E|$.*

The handshake lemma gets its name from interpreting the vertices as individuals, and edges between 2 persons represent that they shook hands. It says that if we ask every person how many other persons they have shook hands with, then the total sum is twice the total number of handshakes. This can intuitively be derived by noting that as every handshake involves two people, so is that handshake going to be counted twice when we ask the people involved.
The handshake lemma is a useful when analyzing the runtime for the different graph coloring algorithms, which is done later in section 4.

Definition 2.4 (Subgraph). *A subgraph $S = (V', E')$ of a graph $G = (V, E)$, is a graph such that $V' \subseteq V$, $E' \subseteq E$.*

In many cases, one is interested in some parts of the graph, and one of the most common ways to decompose a graph is through the induced subgraph.

Definition 2.5 (Induced Subgraph). *For a graph $G = (V, E)$, the induced subgraph $G[X]$ for $X \subseteq V$, is the subgraph $(X, \{(u, v) : (u, v) \in E, u \in X, v \in X\})$.*

In other words, an induced subgraph is a graph constructed by including a subset of the original graphs vertices, and all edges between these vertices in the original graph.
Another common operation, is the graph complement.

Definition 2.6 (Graph Complement). *The graph compliment $\overline{G}$ of a graph $G = (V, E)$ is the graph $(V, \{(u, v) : u \neq v, u \in V, v \in V, (u, v) \notin E\})$.*

In other words, the graph complement of a graph is a graph that contains the same vertices as the original graph, but the vertices are adjacent only if they where not adjacent in the original graph.
Two common operations, and especially relevant for the modular decomposition further down, is the disjoint union and graph join.

Definition 2.7 (Disjoint Union). *The disjoint union $\bigcup_i G_i$ for graphs $G_1 \cdots G_n = (V_1, E_1) \cdots (V_n, E_n)$ where $\bigcap V_i = \varnothing$, is the graph $G = (\bigcup V_i, \bigcup E_i)$.*

Definition 2.8 (Graph Join). *The graph join ∇G_i for graphs $G_1 \cdots G_n = (V_1, E_1) \cdots (V_n, E_n)$ where $\bigcap_i V_i = \varnothing$, is the graph $G = (\{\bigcup V_i, \bigcup E_i \cup \{(u, v) : u \in V_k, v \in V_j, k \neq j\})$*

The disjoint union is the most simple way to combine graphs, and forms a new graph that just contains the original graphs and nothing more. The graph join is similar, in that it produces a new graph with all of edges and vertices from the original graphs, but also for every vertex in an input graph adds a new edge to all the vertices in the other input graphs.

Definition 2.9 (Path). *A path p in graph $G = (V, E)$ is a sequence of vertices $p = (v_1 \cdots v_i)$, such that $(v_j, v_{j+1}) \in E$ for $1 \leq j \leq i - 1$ and $v_j \neq v_k$ for $1 \leq j, k \leq i$ with $j \neq k$.*

One of the original applications of graph theory was to describe different ways to walk on bridges in Königsberg, so the concept of a path has been central for a long time. A path is a sequence vertices, so that the next vertex in the sequence is connected to the previous by an edge, and so that vertices are not repeated. One can then imagine "walking" between vertices by moving on edges connecting them.

Definition 2.10 (Connected Graph). *A graph $G = (V, E)$ is connected, if there exists a path between any two vertices in the graph. Otherwise, we say that a graph is disconnected.*

Definition 2.11 (Cycle). *A sequence of vertices $(v_1, \cdots, v_i, v_1)$ for a graph $G = (V, E)$ is a cycle if $(v_1, \cdots, v_i)$ is a path, and $(v_i, v_1) \in E$.*

A cycle is very similar to a regular path except that we allow, and require, that the first and last vertex are the same. A cycle can be seen as a way to walk in a circle.

Definition 2.12 (Tree). *[15] A tree is a connected graph with no cycles.*

Definition 2.13 (Rooted Tree). *[15] A rooted tree is a graph vertex pair, (G, v), such that $G = (V, E)$ is a tree, and $v \in V$. The vertex v is called the root of the tree.*

Trees have a number of useful properties. Among them is that between any two vertices so is there always an unique path [15]. This in combination with an origin, the root, gives a natural way to describe rooted trees in terms of a parent-child structure, described below.

Definition 2.14 (Child Vertex). *[15] For a rooted tree $((V, E), v)$, a vertex $v_1 \in V$ is considered a child of vertex $v_2 \in V$, if the unique path from v_1 to v starts with v_1, v_2. v_2 is also called the parent of v_1 if v_1 is a child of v_2.*

Definition 2.15 (Graph Coloring). *A graph coloring σ for a graph $G = (V, E)$ is a map from V to C, where C is a set of colors. We say that σ is a k coloring if $|C| \leq k$.*

Definition 2.16 (Proper Coloring). *A graph coloring σ for a graph $G = (V, E)$ is a proper coloring, if no neighbours share the same color, that is $(u, v) \in E$ implies that $\sigma(u) \neq \sigma(v)$*

Definition 2.17 (Partial Coloring). *[9] A partial graph coloring σ for a graph $G = (V, E)$ is a map from $P \subset V$ to C, where C is a set of colors, such that for every vertex in P no neighbours share the same color, that is $u \in P, v \in P, (u, v) \in E$ implies that $\sigma(u) \neq \sigma(v)$.*

Definition 2.18 (Improper Coloring). *[9] An improper graph coloring σ for a graph $G = (V, E)$ is a map from V to C, where C is a set of colors, allowing for clashes in coloring, that is neighbours can share the same color.*

Definition 2.19 (Chromatic Number). *The chromatic number of a graph $G = (V, E)$, denoted by $\chi(G)$, is the least number of colors needed to color the graph with a proper coloring.*

The central problem in this thesis, is to find efficient algorithms for properly coloring a given graph.
Creating a proper coloring for a graph is easy, one could for example assign to every vertex a unique color. Finding an optimal coloring however, i.e. a proper coloring using the fewest possible number of colors, is NP-Hard [8]. For this purpose, different heuristics, algorithms providing a proper coloring with few but not necessarily optimal number of colors, have to be used.
A particular type of graph for which an optimal coloring can easily be determined are so-called complete graphs.

Definition 2.20 (Complete Graph). *A graph $K = (V, E)$ is complete if every vertex $v \in V$ is adjacent to every other vertex, that is, $(u, v) \in E \iff v \neq u$. We also call the graph K_n the complete graph that has n vertices.*

As every vertex is adjacent to every other vertex in a complete graph, so must a proper coloring assign every vertex a unique color, which in turn is an optimal coloring.

2.2 Graph modules and cographs

Definition 2.21 (Cograph). *[2] A graph G is a* cograph *if $G = K_1$, or G is the disjoint union $G = \dot{\bigcup}_i G_i$ of cographs G_i, or G is a join $G = \bigtriangledown_i G_i$ of cographs G_i.*

Definition 2.22 (Graph Module). *[4] Let $G = (V, E)$ be an arbitrary graph. A non-empty vertex set $M \subseteq V$ is a module of G if, for all $x, y \in M$ it holds that $(N_G(x) \setminus M = N_G(y) \setminus M)$. A module M is strong if it does not overlap with any other module M', i.e, if $M \cap M' \in \{M, M', \emptyset\}$.*

Now we have sufficient terminology to describe the most central concept, the modular decomposition.

Definition 2.23 (Modular Decomposition). *[13] The modular decomposition MD for a graph $G = (V, E)$, is the set of all strong modules of the graph.*

The modular decomposition of a graph has a number of useful properties. The modular decomposition of a graph forms a hierarchy, meaning that strong modules can be partitioned into other strong modules. This hierarchy of modules has a natural representation as a tree, the modular decomposition tree.
This tree describes the structure of the modular decomposition of the original graph. Every vertex in the tree represents a strong module, with the root being the strong module containing all vertices. This tree also describes how

the induced subgraph of the module can be constructed from the induced subgraphs of its children, except for prime vertices.

Definition 2.24 (Modular Decomposition Tree). *[7] The modular decomposition tree $(\widetilde{T}, \widetilde{t})$ of a graph $G = (V, E)$,is a rooted vertex labeled tree, such that every vertex is associated with a strong module X in G. A vertex in the tree with associated strong module M is a child to the vertex with associated strong module M' if and only if, $M \subset M'$ and there is no other strong module M'' so that $M \subset M'' \subset M'$. Every vertex with associated module X also has a label distinguishing 3 cases:*

1. *Parallel: $G[X]$ is disconnected.*
2. *Series: $\overline{G[X]}$ is disconnected.*
3. *Prime: $G[X]$ and $\overline{G[X]}$ is connected.*

The root of the tree must also have V as the associated strong module, and all strong modules also have to be a part of the tree.

The modular decomposition of a graph and its corresponding modular decomposition tree is unique [5]. This also means that we can define a unique child-parent structure for the strong modules of a graph.

Definition 2.25 (Child Module). *A strong module M is a child module to another strong module M' for a graph $G = (V, E)$, if the vertex with associated module M is a child to the vertex with associated module M' in the modular decomposition tree for G.*

Note, a label being series in modular decomposition tree with associated module X means that the induced subgraph $G[X]$ can be constructed through graph join on the induced subgraphs of its children associated modules, and it being parallel means that $G[X]$ can be constructed through graph union on the induced subgraphs of its childrens associated modules [13].
A single vertex is a strong module, meaning that all the leafs of the tree are associated with a strong module containing a single vertex. This means that a modular decomposition without prime modules is a cograph, as it is recursively constructed by graph join and graph union on K_1.

2.3 Example

Let us consider how the modular decomposition can be constructed from a graph. We start with the initial graph in Figure 1.

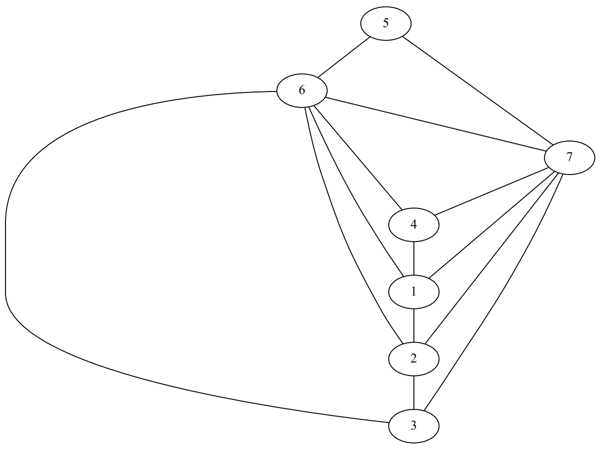


Figure 1: Initial graph

We can construct the modular decomposition, by recursively dividing up the graph into strong modules. We start then to look for the largest strong modules, that partition all of the vertices. We can see that $\{1, 2, 3, 4, 5\}$ and $\{6, 7\}$ are 2 modules, as they share a common neighbourhood. However, by examining other possible modules we see that $\{1, 2, 3, 4, 5, 6\}$ is also a module. As $\{1, 2, 3, 4, 5, 6\} \cap \{6, 7\} = \{7\}$, we can conclude that neither $\{1, 2, 3, 4, 5, 6\}$ nor $\{6, 7\}$ is a strong module. By examining the other possible modules, we can then see that the largest strong modules partitioning all of the vertices are $\{1, 2, 3, 4, 5\}, \{6\}, \{7\}$. This gives the partition shown in Figure 2. The edges between the modules have been removed for clarity, but their internal edges are preserved.

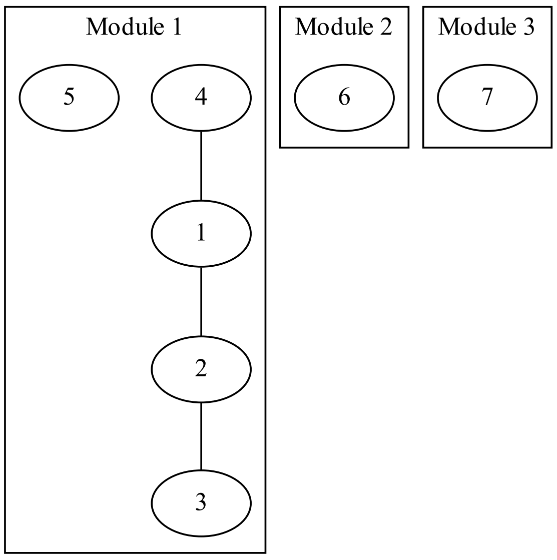


Figure 2: Top child modules identified

From this we continue to further subdivide the strong modules into smaller strong modules. From "Module 1" we can see that the single node 5 forms a module $\{5\}$, and that the whole of $\{1, 2, 3, 4\}$ forms another strong module. This gives the subdivision depicted in Figure 3

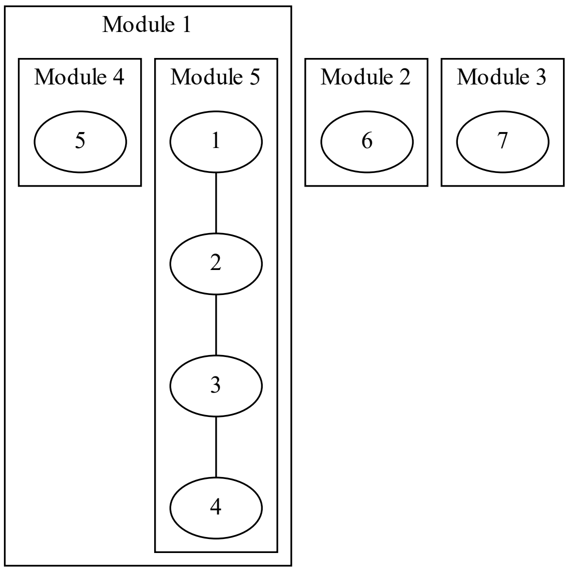


Figure 3: Modules divided recursively

Now, the question is whether or not Module 4 can be divided into further submodules. But we quickly realise that the only strong submodules of Module 4 would be the trivial modules $\{1\}, \{2\}, \{3\}$ and $\{4\}$, and that no larger strong submodules of module 4 exists. We then have all the information needed to construct the modular decomposition tree. We first start by constructing the tree structure, and then assign labels. The unlabeled tree is depicted in Figure 4

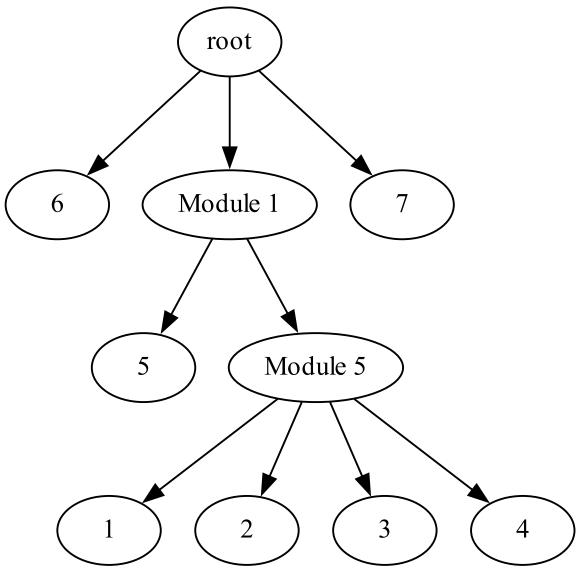


Figure 4: Unlabeled modular decomposition tree

Here the modules only containing a single vertex have been replaced by just that vertex, to improve clarity. From this the tree structure of the modular decomposition can be seen, but it is not yet labeled. We can add the labels by looking at the induced subgraph of the vertices in the module. If that induced subgraph can be constructed by disjoint union on the induced subgraphs of its submodules, we know that it is parallel, and if we can construct it by graph join on the induced subgraphs of its submodules, it is series. If none of these are the case, then the module is prime. By using this method we arrive at the complete modular decomposition tree that can be seen in Figure 5

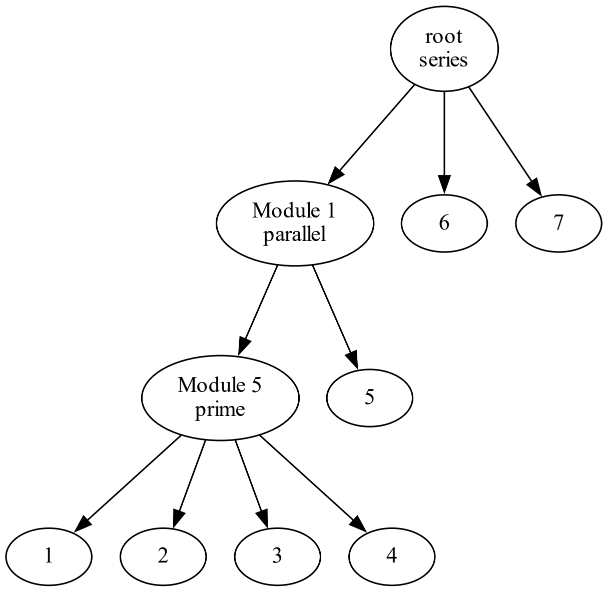


Figure 5: Complete modular decomposition tree

3 General coloring algorithm

Algorithm 1 will form the baseline for our heuristics, and is described in [13]. It provides an optimal coloring given that the graph is a cograph, that is when all modules in the modular decomposition tree is either series or parallel. It does however not provide a way to color the prime modules. This thesis examines multiple different ways these prime modules can be colored, split into 2 parts, a heuristic used, and a coloring strategy.
Algorithm 1 starts by first assigning to every vertex a unique color, and then colors the modules recursively, starting at the root. The vertices in the children of a module are colored before the module itself is colored. How a module is colored depends on which type it is, series, parallel, and prime.
A module X being series in the modular decomposition tree of a graph G means, as stated earlier, that the induced subgraph $G[X]$ can be constructed through graph join on the induced subgraph of the associated module of $X's$ children in the modular decomposition. This means that every child module is completely connected to the other children, most importantly meaning that no child modules can share a color among the other child modules. Assuming that the children have an optimal coloring so is their join also an optimal coloring, as no fewer colors can be used. Having initially given every vertex a unique color, we can also guarantee that this join will not introduce any clashes. This case is therefore "omitted" from the algorithm, keeping

the current coloring is all that needs to be done.
A module X being parallel in the modular decomposition tree of a graph G means that the induced subgraph $G[X]$ can be constructed through disjoint union on the induced subgraph of the associated modules of $X's$ children. This in particular means that the induced subgraph of the child modules have no edges between each other. Therefore, the induced subgraphs of the child modules can use the same set of colors. The children are therefore recolored to use the same smallest set of colors, which assuming that the children are optimally colored is the colors used by the module with the most number of colors.
The last case is when the module is prime. Coloring this module is in the general case NP-Hard. Here we examine how this prime module can be colored, replacing line 12 with a graph coloring heuristic. The question is whether or not the performance is improved when applying this heuristic locally on these prime module compared to applying this heuristic on the whole graph.

Algorithm 1 Modularly-minimal coloring a graph G with MD tree (T, t).

```
Require: Graph $G$ and MD tree $(\widetilde{T}, \widetilde{t})$
1  Initialize a coloring $\sigma$ s.t. all $v \in V(G)$ have different colors
2  for all $u \in V^0(T)$ in post order do
3    if $u$ is parallel then
4      $\mathcal{G} \leftarrow \{G(w) : w \in \text{child}(u)\}$
5      $G^* \leftarrow \arg\max_{w \in \text{child}(v)} |\chi(G(w))|$
6      $S \leftarrow \sigma(V(G^*))$
7      for $H \in \mathcal{G} \setminus \{G^*\}$ do
8        randomly choose an injective map $\phi : \sigma(H) \rightarrow S$
9        for all $x \in H$ do
10         $\sigma(x) \leftarrow \phi(\sigma(x))$
11   else if $u$ is *prime* then
12     Construct a modularly-minimal coloring of $G(u)$ with colors contained in $\sigma(G(u))$ and adjust $\sigma$
       accordingly
```

Algorithm 1 is therefore used in combination with a graph coloring heuristic. The heuristics tested in combination with Algorithm 1 is described below, in section 4.
To distinguish the different ways the graph is colored, a strategy is also defined below, in section 5. The strategy describes how the heuristic is used, and two have already been described here, applying it locally on the prime modules, or applying it on the whole graph, not utilising the modular decomposition at all. Every way a graph is colored here can therefore be described by specifying which heuristic and which strategy is used.

3.1 Modular decomposition

The modular decomposition implementation used is [10]. It implements the algorithm described in [11].
This algorithm computes the modular decomposition tree by first creating a "pseudo-modular decomposition tree". This pseudo-modular decomposition tree is similar to the modular decomposition tree, the difference being that modules of the associated vertices does not need to be strong. The regular modular decomposition tree can then be constructed from this by trimming it down, joining superfluous children to their parents.
This pseudo-modular decomposition tree is in turn constructed by selecting a vertex v from the graph, and partitioning the vertices in the graph with modules that are either $\{v\}$, or the largest not overlapping modules not containing v. From this partition, we can construct base pseudo modular decomposition. We then recursively add to this tree, by doing the same procedure for the parts in this partition, and combine their trees.

4 Heuristics

A graph coloring heuristic is an algorithm for coloring a graph, that does not necessarily give an optimal coloring, but uses various methods to approximate a good coloring.
In this section, the different heuristics used to color the graphs is described, while section 5 describe how they are applied.

4.1 Greedy

The classic greedy coloring algorithm. It traverses all of the vertices in the graph in an arbitrary order, and for every vertex assigns the first colored not shared amongst its neighbours. Assigning the first available color can trivially be done by for every color checking whether or not the vertex has a neighbour with that color, and using it if it is not the case. This method does however have a worst case runtime of $O(|C||N_G(v)|) = O(|V|^2)$.
A more efficient way to do this assignment is with the function **get first color**, which uses a map called *used* from colors to Booleans, see Algorithm 2.
When coloring v, we first examine the neighbours of v, if a neighbor has a color c, index c in *used* is set to true, which correspond line 2-4 in Algorithm 2.

After having iterated through every neighbour, we step through *used*, and use the first color that is not set to true. A maximal $deg(v)$ of vertices can be set to true through this process, which means that finding the first available color from *used* this way is at most $|deg(v)| + 1$ steps. This part correspond to lines 5-8.
Then, we reset the *used* map by going through all of the neighbours the same way and setting the index of *used* to false instead, so that the next time when calling **get first color**, *used* maps all colors to false. This part are the final lines 9-12.
The whole process of **get first color** is therefore $O(|deg(v)| + |deg(v)| + |deg(v)|) = O(|deg(v)|)$. Using the handshake lemma, we can conclude that calling **get first color** on all vertices has a total runtime of $O(|E|)$. *used* however needs enough space to contain every color, which in the worst case is $|V|$ number of colors, which means that the total runtime is $O(|V| + |E|)$.

Algorithm 2 get first color

```
Require: Graph $G = (V, E)$
Require: vertex $v \in V$
Require: partial coloring $\sigma$
Require: $used \leftarrow$ map from colors to Booleans, with all values initially set to false
1  $ReturnValue \leftarrow NULL$
2  for all $u \in N_G(v)$ do
3    if $u$ is colored in $\sigma$ then
4      $used(\sigma(u)) \leftarrow true$
5  for all $c \in \sigma$ do
6    if $used(c) = false$ then
7      $ReturnValue \leftarrow c$
8      break
9  for all $u \in N_G(v)$ do
10   if $u$ is colored in $\sigma$ then
11     $used(\sigma(u)) \leftarrow false$
12 return $ReturnValue$
```

Algorithm 3 Greedy

```
Require: Graph $G = (V, E)$
1  $V' \leftarrow$ List containing all $v \in V$ in any order
2  $C \leftarrow$ List of possible colors $\{1 \cdots |V|\}$
3  $\sigma \leftarrow$ Initial empty partial coloring
4  $used \leftarrow$ map from colors in $C$ to Booleans, initialised so all values are false
5  for all $v \in V$ do
6    update $\sigma$ so that $\sigma(v) =$ get first color$(G, v, \sigma, used)$
```

4.1.1 Example

Here we give an example for how **get first color** is executed. We have the partially colored graph in Figure 6, and want to color the vertex 1 with **get first color**.

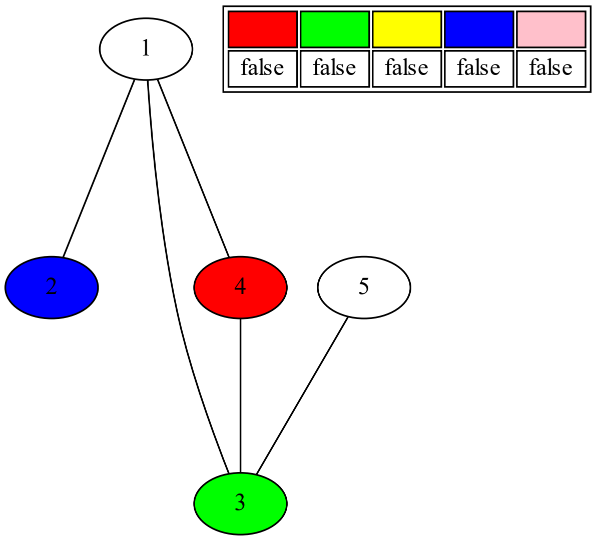


Figure 6: Partially colored graph

Here we can see our *used* map to the right. At the start of every call to **get first color** so is every color maped to false. Now we go over every neighbour of 1, that is 2,3 and 4, and update the map so that their respective colors is set to true. This gives the state depicted in Figure 7.

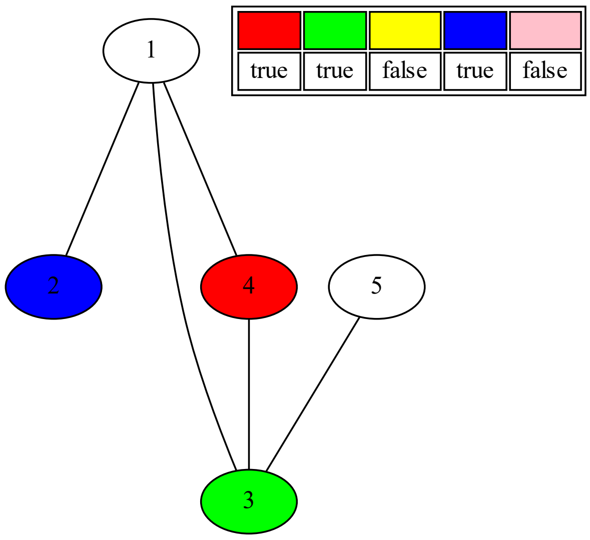


Figure 7: Used map updated

Now we step through the map from the leftmost column to the right, and examine whether or not the current column maps the color to a "false" value. If it does, we have found the first available color, and assign it to the vertex we want to color. In this case the first available color is yellow. After having determined the first available color so is the *used* map reset so all colors map to false again, and appropriate for further application of **get first color**. The vertex is now colored using **get first color**, and we arrive at the state in Figure 8.

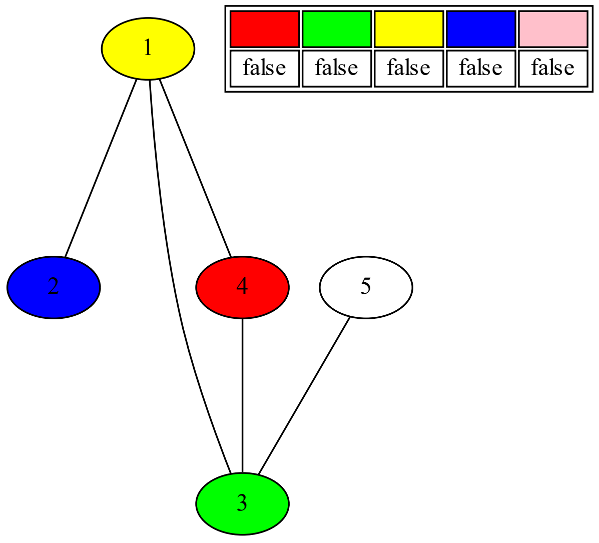


Figure 8: Vertex 1 colored, Used map reset

4.2 Dsatur

Dsatur is an algorithm that is similar to Greedy, in the sense that it traverses every vertex, and assigns to each of them the first available color. The difference is mostly in how this traversal is constructed. In greedy, this traversal is an arbitrary order, but for Dsatur this traversal is constructed in a specific way, by using the saturation degree.

Definition 4.1 (Saturation degree). *[9] The saturation degree $sat(v)$ for a vertex $v \in V$ for a graph $G = (V, E)$ and a partial coloring σ from $P \subset V$ to the set of colors C, is the number of unique colors among its colored neighbours, that is $sat(v) = |\{c : c = \sigma(u), u \in N_G(v), u \in P\}|$.*

The vertices are colored one at a time, until all of the vertices are colored. At every step of the iteration, we identify the vertex that has the highest

saturation degree, and if there are ties we choose the vertex among them with the highest degree, and an arbitrary vertex is chosen among the ties of there are ties remaining. This also means that the first vertex colored is the vertex with the highest degree, as the saturation degree of every vertex is zero at the start of the algorithm. This step corresponds to line 4 in Algorithm 4.
We then color this vertex with the first available color, and update the partial color accordingly, which means that the saturation degree for its neighbours are changed, this step being line 5.
Dsatur being a well studied algorithm also means that it has some variations in the literature, and this presentation is based on [9].

Algorithm 4 Dsatur

```
Require: Graph $G = (V, E)$
1 $C \leftarrow$ List of possible colors $\{1 \cdots |V|\}$
2 $\sigma \leftarrow$ Initial empty partial coloring
3 while There exists uncolored vertices in $V$ do
4     $v \leftarrow$ uncolored vertex in $V$, such that $sat(v)$ is minimal. In case of ties, choose the $v$ that also minimizes $deg(v)$ for the subgraph of $G$ induced by the uncolored vertices. Remaining ties are broken by choosing an arbitrary vertex among the ties.
5     $\sigma \leftarrow$ updated coloring where $\sigma(v)$ is the first available color in regards to $C$.
```

When analyzing the runtime for Dsatur, we can begin by stating that assigning the first available color can be done just as in greedy with **get first color**, giving us at least $O(|V| + |E|)$ runtime.
Determining which nodes has the highest saturation degree can be done with a priority queue. First we initialise it by setting the saturation degree for all vertices to zero. With the priority queue, we can then get the vertex with the highest saturation degree in constant time. After having colored it, we update the saturation degree for its neighbours, which implies the removal of that neighbour from the list, and then adding it back with the updated saturation degree. With appropriate data structures, the initialization can be done in $O(|V|log(|V|)$, and removing and inserting an element can be done in $O(log(|V|))$ time. As this insertion is done for every neighbour, so can we again use the handshake lemma and see that the total runtime for updating neighbours this way is $O(|E|log(|V|))$. This gives a total runtime of $O(|V| + |E| + |V|log(|V|) + |E|log(|V|)) = O((|V| + |E|)log(|V|))$. A more thorough description of this implementation can be found in [9].

4.3 Recursive largest first

Recursive largest first is more complicated than both Greedy and Dsatur, and subsequently has a higher runtime.

The idea behind Recursive largest first, is to create a partition of the vertices of the graph, where the vertices in the different parts are all non-adjacent to each other. This ensures that all the vertices in a given part can share the same color. This partition is constructed one part at a time.
Vertices are added one at a time when creating a part in the partition. Doing the partition "largest first" means that we want to add the vertices with the highest degree first. We only consider the degrees from the subgraph induced by the vertices that are adjacent to the current part. This is similar to Dsatur, in that we want to color the currently "most restricted" vertices first.
A part in the partition is therefore created by first adding the vertex with the highest degree among the vertices in the induced subgraph of vertices not currently in a partition. Having added it to this part, we then consider that vertex assigned, meaning that the induced subgraph of vertices not currently in a partition is updated. We also add the neighbours of this vertex, that have not been assigned to a part, to a list of vertices that contains the vertices adjacent to the current part. These steps corresponds lines 6-8 in Algorithm 5
After having added the initial vertex, we now add vertices that are non-adjacent to all of the vertices in the current part, which corresponds to lines 9-16. We do this similarly to when adding the initial vertex. We identify the vertex among the unpartitioned vertices that has the highest number of neighbours in the list of vertices that are adjacent to the current part, choosing the vertex with the highest degree in the induced sugraph of unpartitioned vertices in case of ties. We add this vertex to the current part, its neighbours that are unassigned to the list of adjacent vertices, and remove it from the unassigned vertices. We continue this process until no more vertices can be added, that is the there are no more unpartitioned vertices that are not adjacent to the current part.
At this point, if there are still vertices to assign, we repeat by creating another part the same way, going back to line 6. If all vertices have been assigned, so can we assign every part in the partition a unique color, and then color every vertex in that part with the assigned color.
This presentation of the algorithm is based on [9].

Algorithm 5 Recursive largest first (RLF)

```
Require: Graph $G = (V, E)$
 1 Partition ← {}
 2 $C$ ← List of possible colors $\{1 \cdots |V|\}$
 3 $M \leftarrow \{\}$
 4 $S \leftarrow V$
 5 while $|S| > 0$ do
 6   $v$ ← the vertex maximizing $deg(v)$ for $G[S]$
 7   $M \leftarrow \{v\}$
 8   $Adj \leftarrow N_{G[S]}(v)$
 9   while TRUE do
10     Candidates ← $S \setminus Adj$
11     if |Candidates| $= 0$ then
12       break
13     $u$ ← the $u \in$ Candidates maximizing $deg(u)$ in $G[Adj \cup \{u\}]$, ties broken by the $u$ maximizing
       $deg(u)$ in $G[S]$. Remaining ties are broken by choosing an arbitrary vertex among the ties.
14     $M \leftarrow M \cup \{u\}$
15     $Adj \leftarrow Adj \cup N_{G[S]}(u)$
16     $S \leftarrow S \setminus \{u\}$
17   Partition ← Partition ∪ $\{M\}$
18 Assign consecutive colors from $C$ to the partitions in Partitions, and then color every vertex in that
   partition with that color.
```

Analyzing the runtime, we can see that the adding vertices to a part in the partition and the number of partitions is at most $O(|V|)$. The runtime is therefore dependant on how these vertices are chosen, and how to ensure that the partitions do not have any overlapping vertices.
We can ensure that no adjacent vertices are added to the current part, by keeping the current possible candidates in a hash map, and whenever we add a vertex to the current part remove its neighbours from the current candidates. Doing this is, with the handshake lemma, $O(|E|)$ when all vertices has been added, leading to at least a $O(|V| + |E|)$ runtime.
We can also ensure that the vertex added is the vertex with the highest degree among the non-adjacent vertices by recalculating the respective degrees for the uncolored vertices, in both the subgraph induced by the unpartitioned vertices and the vertices adjacent to the current part, which can be done in $O(|E|)$ time every time a vertex is added, which is done $|V|$ times. The algorithm can therefore be implemented in $O((|V||E|) + |V| + |E|) = O(|V||E|)$ time [9].

4.3.1 Example

An example run of the algorithm is demonstrated here. We start with the graph Figure 9.

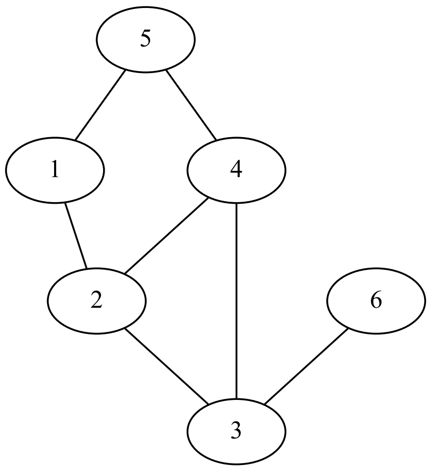


Figure 9: Initial graph

Now, we start by constructing the first part in the partition. The first vertex in the partition is the vertex with the highest degree among the unpartitioned vertices. We can see that the vertex with the highest degree is vertex 4. Now we remove it from this graph and add it to the current part, as well as adding its neighbour to the set of adjacent vertices to the current part. We represent vertices in the current partition with a red color, and vertices that are adjacent to the current part with a dashed outline. We now get the graph in Figure 10

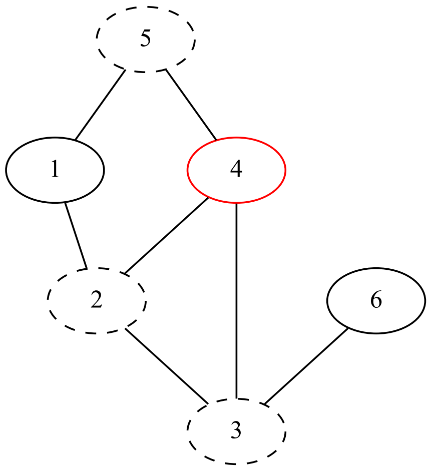


Figure 10: One vertex has been assigned

Here we can see that the vertices that are not adjacent to the current part,

$\{4\}$, is 1 and 6. The next vertex we add is the vertex among the unassigned vertices not adjacent to the current part that has the highest number of neighbors that are unassigned and adjacent to the current part. We can see that 6 has one neighbour that is unassigned and adjacent to the current part, while 1 has two. We therefore add 1 next and get the graph in Figure 11

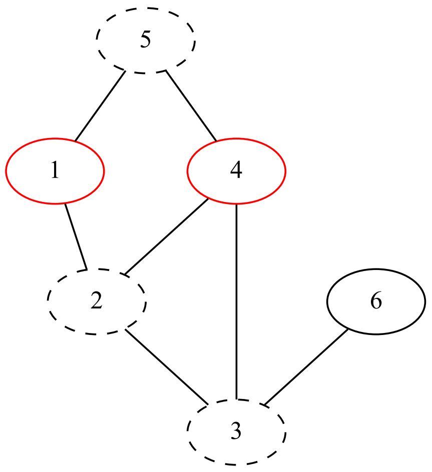


Figure 11: Two vertices has been assigned

Now we can see that the only alternative left is 6, and add it to the current part. After this step, so is all of the remaining vertices adjacent to the current part, which means that we have to construct a new part. This part is constructed from the unnassigned vertices, which means that we have to remove the vertices 1, 4 and 6 before. We therefore repeat this procedure on the graph depicted in Figure 12.

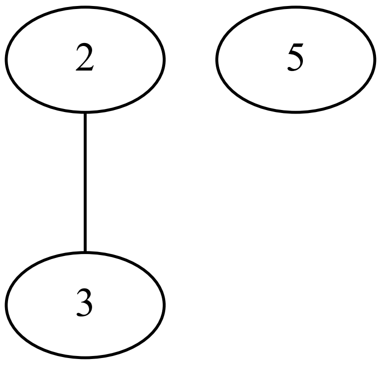


Figure 12: Construct new part on the induced subraph of unassigned vertices

By repeating the previous step until all vertices have been assigned a part, we arrive at the partition $\{\{4, 1, 6\}, \{3, 5\}, \{2\}\}$. Now we can assign every part in this partition a color, and color the vertices in that part with the same color. This gives us the final coloring depicted in Figure 13.

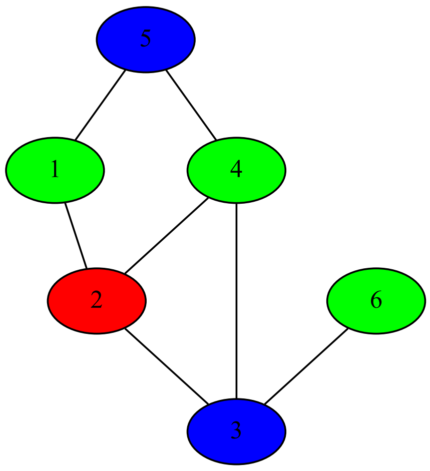


Figure 13: Final coloring

4.4 TabuCol

TabuCol is a graph coloring algorithm that unlike the previous algorithms, does not create a coloring in a constructive fashion, but instead tries to find a coloring for a specific number of target colors.

TabuCol first forms a random improper coloring of the vertices with colors from the allowed set, and then modifies this coloring by looking at the vertices that forms a clash, that is, if neighbours have the same colors. For these vertices, every new coloring are evaluated in how many clashes they would result in. Then the recoloring that has the lowest number of clashes, even if the number clashes are greater than the current number of clashes, is applied. This can however introduce cycles of recolorings. Certain recolorings might lead into each other so that the algorithm just iterates through this cycle every iteration, preventing it from ever reaching a better coloring. The Tabu list is introduced to prevent this . A Tabu list is a list of vertex-color pairs (v, u). Whenever a new vertex recoloring is made, it is added to the Tabu list. A new vertex recoloring is only considered if it is not a part of the Tabu list. This tabu list have a set size, so that the first element is removed when a new vertex-color pair is added and the list is already full. Ties are broken randomly in the case that multiple colorings share the same lowest increase in clashes. These ties are broken randomly instead of arbitrarily in order to avoid cycles in colorings.

Tabu moves are however allowed in some scenarios, specifically if applying that coloring would create a better recoloring than the previous best, which is commonly referred to as the aspiration. Whenever a new coloring is applied, the current number of clashes are calculated, if it is below the current aspiration then the aspiration is updated to that number. A new coloring that is on the tabu list is then allowed given that applying that coloring would result in a new number of clashes below the current aspiration.

This process of picking a new vertex recoloring is repeated until a coloring with zero clashes is produced, or a predetermined number of moves have been made. As the algorithm only gets a valid coloring for a specific k, so must we also have a way to utilise this algorithm to get the lowest possible coloring. Here a similar method described in [9] is used, that is, first a coloring is made with RLF, and the number of used colors is assigned to k and the resulting coloring to σ. Then we try to color the graph with TabuCol with $k - 1$ allowed colors . If this results in a proper coloring, this step is repeated and we subtract k by one, and we save the coloring to σ. If the coloring is not proper, then the current σ is the coloring that is applied. This procedure can be seen in Algorithm 8

TabuCol has some subtle variations, and this particular definition is based on the presentation in [9]. Alternatives are for example whether or not all possible recolorings are consider, if only the first coloring with a lower clash count is considered, and if one allows vertices that do not have any clashes to be recolored.

Algorithm 6 Clashes

```
Require: Improper coloring $\sigma$
Require: Graph $G = (V, E)$
return $|\{(u, v) : (u, v) \in E, \sigma(u) = \sigma(v)\}|$
```

Algorithm 7 TabuCol

```
Require: Graph $G = (V, E)$
Require: Integer $k > 0$
Require: Integer $MaxIt > 0$
Require: Integer $MaxTabu > 0$
$\sigma \leftarrow$ random improper coloring with $k$ colors
$CurIt \leftarrow 0$
$CurrentClash =$ Clashes$(\sigma, G)$
$Asp \leftarrow CurrentClash - 1$
$Tabu \leftarrow$ Empty tabu list
while $CurrentClash > 0$ and $CurIt < MaxIt$ do
  $Reps \leftarrow \emptyset$
  for all $v \in V$ do
    for all $c \in \sigma$ do
      if $(v, c) \notin Tabu$ or Clashes$(\sigma, G)$ with $(v, c)$ applied $\leq Asp$ then
        $Reps \leftarrow Reps \cup \{(v, c)\}$
  $(v', u') \leftarrow$ where $(v', u') \in Reps$ and Clashes$(\sigma, G)$ with $(v', u')$ applied is minimal, ties broken randomly
  Update $\sigma$ so that $\sigma(v') = u'$
  $CurrentClash \leftarrow$ Clashes$(\sigma, G)$
  if $CurrentClash \leq Asp$ then
    $Asp \leftarrow CurrentClash - 1$
  Update $Tabu$ to contain $(v', u')$, and remove the oldest element if $|Tabu| > MaxTabu$
  $CurIt \leftarrow CurIt + 1$
  if $CurrentClash = 0$ then
    break
```

Algorithm 8 TabuCol-lowest

```
Require: Graph $G = (V, E)$
$k \leftarrow$ number of colors used by RLF applied on $G$.
$\sigma \leftarrow$ the coloring of $G$, after applying RLF
while true do
  if TabuCol applied on $G$ with $k$ allowed colors results in a valid coloring then
    $k \leftarrow k - 1$
    $\sigma \leftarrow$ the resulting coloring for TabuCol with $k$ allowed colors
  else
    break
```

By utilizing a large $|V| \times k$ matrix that stores how many neighbours to v that share the color c in the (v, c) position of the matrix, looking up the recoloring that results in the lowest number of clashes can be done by scanning through the whole matrix, that is $O(|V|k)$ time. Keeping this method up to date in turn requires a step of $|deg(v)|$ for every vertex getting recolored. As the number of repetitions is constant, the total runtime of

an single iteration of TabuCol can therefore be implemented in $O(|V|k + deg(v)) = O(|V|k + |V|) = O(|V|k)$ [9]. The initial creation of the $|V| \times k$ matrix with appropriate entries requires iterating through every edge, giving a runtime of $(|V|k+|E|)$, and in turn a total runtime for TabuCol at $O(|E|+(|V|k) \cdot MaxIt)$. This does however not account for how many times one needs to apply the whole TabuCol method to determine the lowest color that can be achieved by it.
The worst case scenario would be that the initial k was the number of vertices in the graph, and the lowest color achieved by repeated applications was 1. In this case the runtime would be $O((|E|+(|V|k) \cdot MaxIt)|V|) = O(|E||V|+(|V|^2k \cdot MaxIt))$. This is in practice often a too pessimistic analysis, as the difference between the initial coloring and the coloring achieved by TabuCol is rarely that large. It is also worth noting that the number of repetitions needed for good results in most cases greatly outnumber the number of vertices in the graph, meaning that this heuristic in practice exhibits the largest runtimes.

5 Strategies

In the case where the whole graph does not contain any prime modules, so can an optimal coloring can be obtained with Algorithm 1 in linear time [13]. We will use different strategies on how to apply the different heuristics described in section 4 in case the graph contains prime modules, these strategies are described below. An example run of the different strategies is also given in subsection 5.4

5.1 Whole graph

In this baseline test, so is the whole graph colored using the heuristic, not utilizing the modular decomposition at all. This strategy is used as a baseline to compare whether or not the other strategies improves the performance for the graph.

5.2 Whole prime coloring

With this strategy so is the whole prime module colored using the heuristic. This modifies the behaviour of Algorithm 1 in to ways. First, line 12 is replaced by a call to the heuristic used with this strategy, and secondly, the children of the prime module is not colored before it, as doing so would be

redundant and waste time. This also means prime modules contained within other prime modules are not colored individually with the heuristic.
This method is the simplest combining strategy, but worth noting is that is equivalent to coloring the whole graph using the heuristic in the case where the root node is prime, and therefore only offers a possible improvement to existing heuristics when the root in the modular decomposition is not prime.

Algorithm 9 Whole prime coloring

```
Require: Graph $G = (V, E)$
Require: Modular decomposition tree $MD$ of $G$
Require: Graph coloring heuristic $H$
Require: Module $M$ in the modular decomposition
1  Initialize a coloring s.t. all $v \in V$ have different colors
2  if $M$ is parallel or $M$ is series then
3    for all Child module $u$ of $M$ do
4      Color the vertices in the child module with
       Whole prime coloring$(G, MD, H, u)$
5    if $M$ is parallel then
6      $M_{child} \leftarrow$ the child modules of $M$ in $MD$
7      $M' \leftarrow$ the child module in $M_{child}$ that is colored using the most number of colors
8      $C_{M'} \leftarrow$ the colors used in the coloring of $M'$
9      for $m \in M_{child} \setminus \{M'\}$ do
10       $\sigma \leftarrow$ coloring of $m$
11       $\phi \leftarrow$ injective map from the colors used in $\sigma$ to $C_{M'}$
12       for all $v \in m$ do
13         $\sigma(v) \leftarrow \phi(\sigma(v))$
14 else
15   Color the vertices in $M$ by applying the heuristic $H$ on $G[M]$, using colors present in the current
     coloring of $M$.
```

The whole graph G can be colored with Algorithm 9 by applying it on the root module of the modular decomposition.
The requirement in line 15 that the coloring have to use colors already present in the vertices of the prime module is needed because otherwise we could introduce clashes. Algorithm 1 as well as Algorithm 9 assumes that the children of a series module do not share colors after being colored. We can ensure this by only using the colors available in the module being colored, as all vertices are initially given a unique color, which means that modules with the same parent start with a disjunct set of colors.

5.3 Quotient recoloring

Definition 5.1 (Quotient graph). *The quotient graph Q for a graph $G = (V, E)$ over a partition $P = \{P_1 \cdots P_i\}$ of the vertices V, is a graph $Q = \{P, \{(P_i, P_j) : j \neq i, \exists v \in P_i, \exists u \in P_j((u, v) \in E)\}\}$*

This coloring, unlike the previous two, also attempts to colorize the prime modules using the modular decomposition tree. Here, every prime module

is colored just like in 'Whole prime coloring', but only if it contains under a predetermined number of vertices. Otherwise, all of its children are colored first, then the quotient graph for the children of the prime modules is constructed. This quotient graph is then colored with some graph coloring heuristic, potentially different from the ones used in coloring the modules, and then every child module with the same color in the quotient graph can now be recolored to use the same set of colors.
The quotient graph is a graph describing whether or not parts in a vertex partition of a graph are adjacent or not, instead of individual vertices. Two vertex parts are adjacent in the quotient graph if there exists an edge between any pair of vertices, where one vertex is in the first part, and the other vertex is in the other part. A quotient graph where the partition is all of the original vertices individually, is therefore isomorphic to the original graph.

Algorithm 10 Quotient coloring

```
Require: Graph $G = (V, E)$
Require: Modular decomposition tree $MD$ of $G$
Require: Module $M$ in the modular decomposition
Require: Threshold for heuristic $T$
Require: Graph coloring heuristic $H$ for prime modules
Require: Graph coloring heuristic $H'$ for quotient graph
Initialize a coloring s.t. all $v \in V$ have different colors
for all Child module $u$ of $M$ do
    Color the vertices in the child module with
    Quotient coloring$(G, MD, u, T, H, H')$
if $M$ is parallel then
    $M_{child} \leftarrow$ the child modules of $M$ in $MD$
    $M' \leftarrow$ the child module in $M_{child}$ that is colored using the most number of colors
    $C_{M'} \leftarrow$ the colors used in the coloring of $M'$
    for $m \in M_{child} \setminus \{M'\}$ do
        $\sigma \leftarrow$ coloring of $m$
        $\phi \leftarrow$ injective map from the colors used in $\sigma$ to $C_{M'}$
        for all $v \in m$ do
            $\sigma(v) \leftarrow \phi(\sigma(v))$
else if $M$ is prime then
    if $|M| \leq T$ then
        Color the graph $G[M]$ with $H$, using colors present in the current coloring of $M$
        return
    $Q \leftarrow$ quotient graph of $G[M]$ where the child modules of $M$ in $MD$ is the partition
    $\sigma_Q \leftarrow$ Coloring of $Q$ after coloring it with $H'$
    for all colors $c$ in the coloring of $Q$ do
        $Q_c \leftarrow \{m : m \in V(Q), \sigma_Q(m) = c\}$
        $m' \leftarrow$ the module in $Q_c$ with the most colors
        $C_{m'} \leftarrow$ the colors used in the coloring of $m'$
        for all $m \in Q_c \setminus \{m'\}$ do
            $\sigma \leftarrow$ the coloring of $m$
            $\phi \leftarrow$ map from the colors used in $\sigma$ to $C_{m'}$
            for all $x \in m$ do
                $\sigma(x) \leftarrow \phi(\sigma(x))$
```

As a quotient graph can for a graph $G = (V, E)$ be constructed in $O(|V| +$

$|E|$) time for a partition of the vertices, we can see that the pathological worst case for constructing the quotients, are when only one vertex is removed for each recursion. In that case, the construction of the quotient graphs are $O(|V|^2 + |E||V|)$. The runtime is then dependant on the properties of the heuristic applied on the quotient graph to color it. The time for the heuristic applied when the modules that are small enough can be seen as constant, as the maximal size and complexity of the graph is bounded.
This notably implies that the runtime for quotient coloring is in the worst case slower than any of the individual constructive algorithms, that is Greedy, Dsatur and RLF.

5.4 Example

Here an example of the difference in how these strategies are applied is given. We start by looking at the graph in Figure 14.

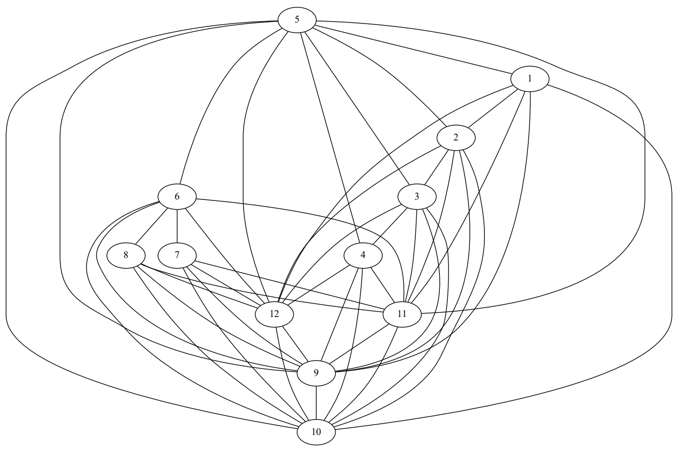


Figure 14: Graph to color

This graph is relatively large and has a lot of edges. We can gain some insight into its structure by looking at its modular decomposition tree, depicted in Figure 15.

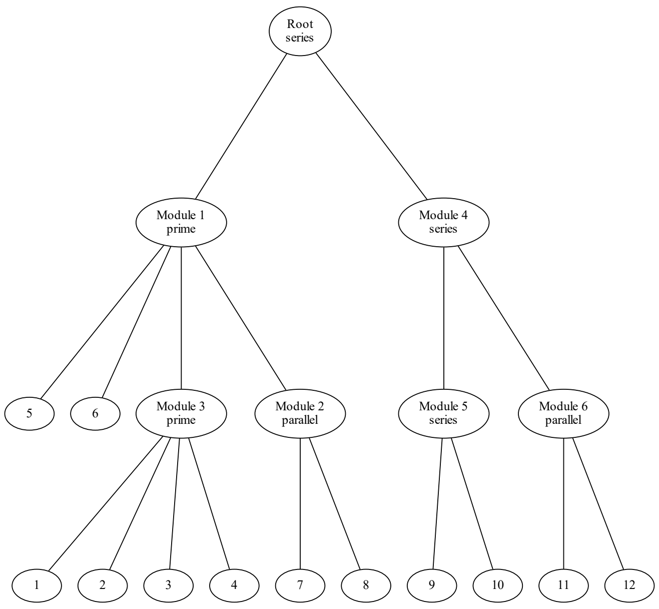


Figure 15: Modular decomposition of graph

We can see from this figure that the root node is not prime, and that it contains a prime module with a nested prime module, as well cograph subgraph in module 4 containing 4 vertices.
From this we can now apply the different strategies. The case for 'Whole-Graph' is omitted here as it just involves a regular application of a graph coloring algorithm. The heuristic we use in combination with the strategies is RLF.

5.4.1 Whole prime

When applying the 'Whole prime' strategy, we start with an initial coloring where every vertex has a unique color, depicted in Figure 16.

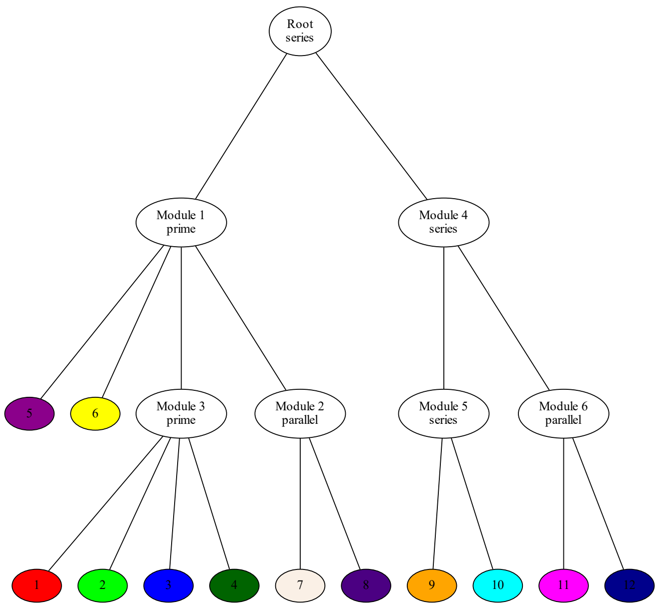


Figure 16: Modular decomposition of graph

When coloring the root module, we start by coloring its children first. Coloring Module 4 can be made optimally as it contains no prime modules, and doing so we get the coloring in Figure 17.

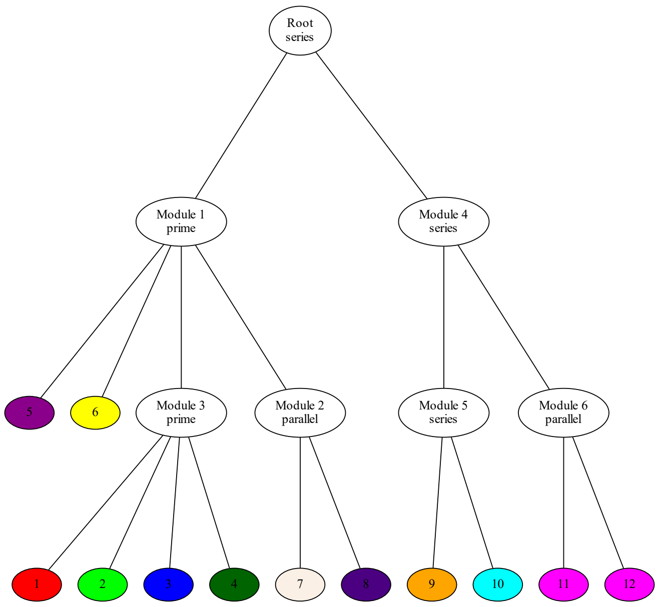
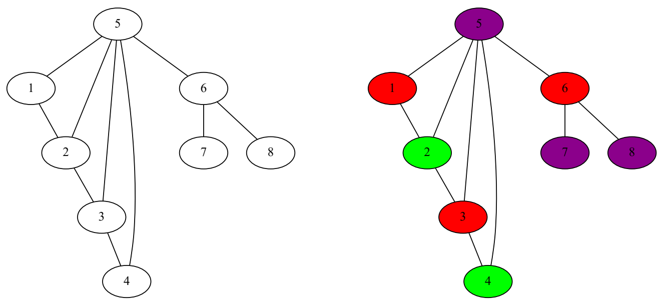


Figure 17: Module 4 optimally colored

Now we color Module 1. Here we do not recurse further down into the tree, and instead color the whole subgraph induced by Module 1 with a graph coloring algorithm. That is, we want to color the graph Figure 18a below. The result of applying RLF is shown in Figure 18b.



(a) Subgraph induced by module 1 (b) Subgraph colored with RLF

Figure 18: Coloring of Module 1 using 'Whole prime' strategy

Now all of the child modules of the root module have been colored, and the vertices are colored as displayed in Figure 19.

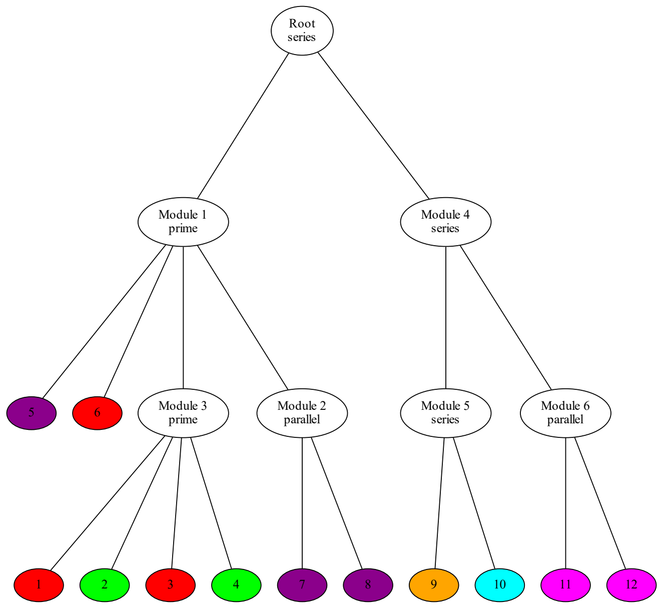


Figure 19: All child modules of the root module have been colored

The root module being series means that no recoloring of the child modules have to be done, and that we have a finished coloring, which can be seen in Figure 20

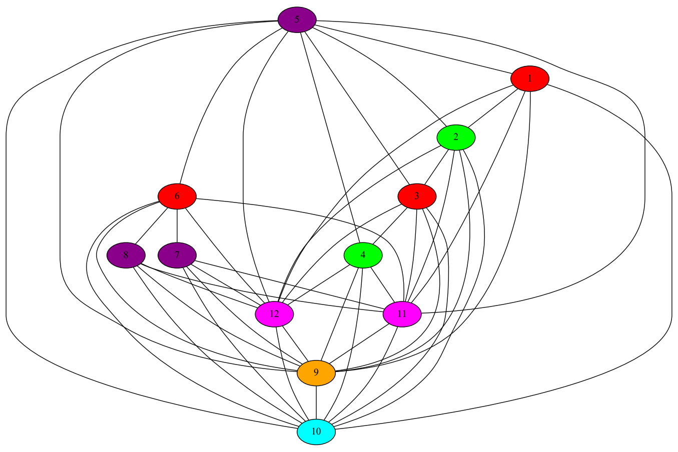


Figure 20: Graph colored using 'Whole prime' coloring

5.4.2 Quotient coloring

Now we color the same graph using the quotient coloring. It behaves the same way as 'Whole prime' coloring, except for when coloring prime modules. We therefore skip to the step where we have colored Module 4, and are about to color Module 1, that is the state in Figure 17.
Unlike in 'Whole prime' coloring, when coloring a prime module with the quotient strategy so are we utilising the modular decomposition to color the prime module. We start by coloring the child modules of Module 1.
The trivial modules just retain their color, and Module 2 can be colored optimally. Coloring Module 3 we again have to apply the prime module case. But as Module 3 contains only trivial modules so is it is quotient graph isomorphic to the original graph, and therefore equivalent to just applying the heuristic directly on it. After having colored the children of Module 1 we arrive at the graph Figure 21
Note that the threshold set here for coloring prime modules of a specific size directly with a heuristic is 0, as the example would otherwise be equivalent to 'Whole prime' coloring.

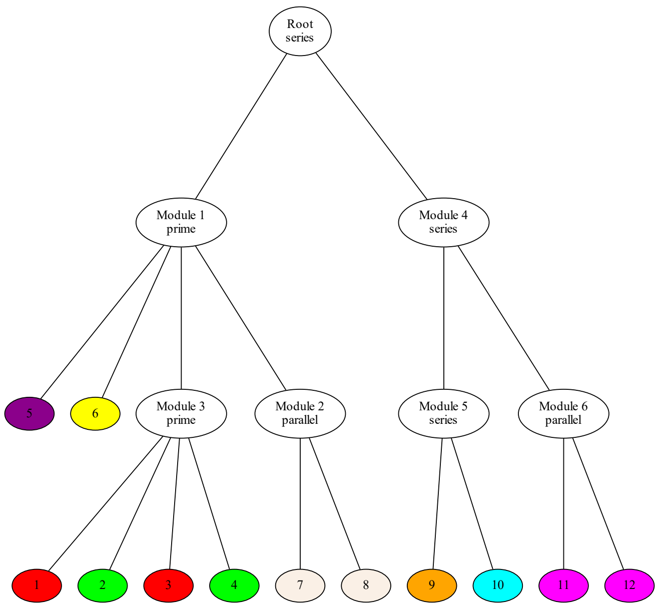
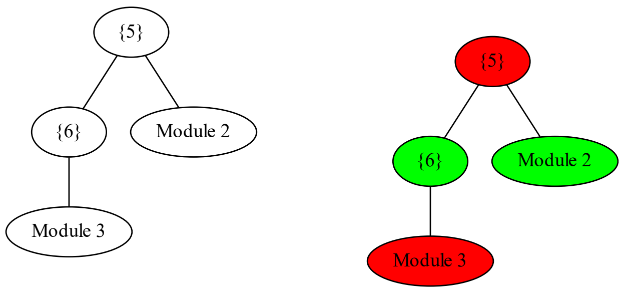


Figure 21: Children of Module 1 colored

Now the novel step of the quotient strategy is applied. Now we want to combine the results for the coloring of the children to a good coloring of Module 1. The current coloring is a proper coloring, but it is not necessarily optimal. The quotient strategy tries to improve this coloring by examining whether or not child modules can share the same set of colors. To do this, we first construct the quotient graph, where the partition is the vertices in the child modules, that is $\{\{6\}, \{5\}, \{1, 2, 3, 4\}, \{7, 8\}\}$. This quotient graph can be seen in Figure 22a. We then color this graph using some predetermined heuristic, and get the result in Figure 22b



(a) Quotient graph of the children of module 1

(b) Colored quotient graph

Figure 22: Quotient graph coloring

The modules that share color can not be neighbours in a proper coloring, which by the definition of a quotient graph means that there are no edges between 2 vertices with different modules of the same color. This in turn means that they can share the same set of colors. To ensure that we can recolor them and end with a proper coloring, so are we using the set of colors from the module with the most number of colors for every color category. This means that we recolor module $\{5\}$ and Module 3 with beige, and Module 2 and $\{6\}$ with red and green. This gives the coloring in Figure 23.

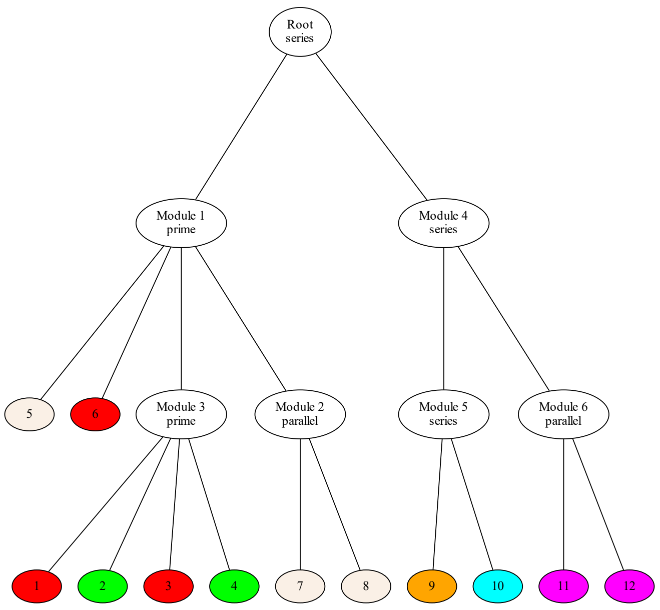


Figure 23: All child modules of Root colored using quotient strategy

Now all of the child modules of the root module have been colored. The root being series means that no further recoloring is done, and we can see the final coloring in Figure 24

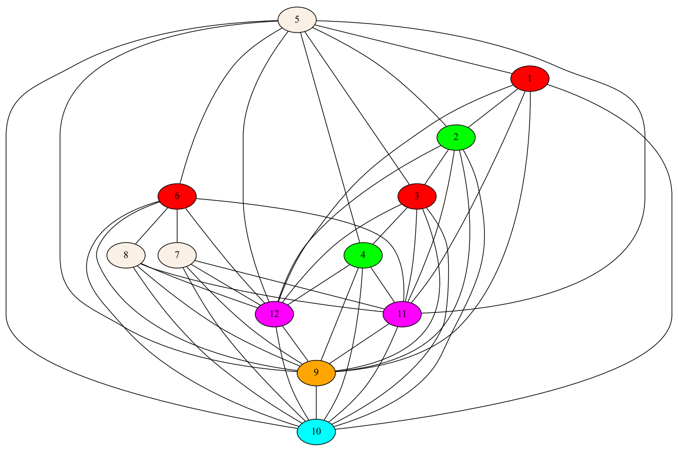


Figure 24: Finished coloring using the quotient strategy

6 Data generation

The test sets used are graphs from the DIMACS benchmark set [3], and graphs generated that are not prime, as outlined below.
All of the graphs from the DIMACS benchmark sets are difficult to color, and also have modular decomposition where the root vertex is prime. As they are commonly used for benchmarks, they also provide known best current colorings for the different graphs. However, the modular decomposition might provide a more efficient way to color graphs where only some of the vertices are in a prime module.
The algorithm for generating these graphs is described in Algorithm 12. First an ordinary binary tree is randomly generated with a specified number of leafs , and then every vertex in this tree is given a label "series" or "parallel". This then describes a cograph, where the leafs are the K_1 bases, and a label of "series" means that the children are joined by disjoint union and a label of "parallel" means that the children are joined by graph join. From this so can the corresponding graph be constructed. From this cograph, we then randomly add edges in predetermined number of modules and with a predetermined size.

Algorithm 11 Construct cograph

```
Require: Vertex labeled tree $T$
if $|V(T)| = 1$ then
    return $(V(T), \emptyset)$
$G \leftarrow$ empty graph $(\emptyset, \emptyset)$
for all $v \in N_T(root(T))$ do
    if Label of $root(T)$ is parallel then
        $G \leftarrow \bigcup G$, Construct cograph applied on the subtree of $T$ with $v$ as root
    else if Label of $root(T)$ is series then
        $G \leftarrow \nabla G$, Construct cograph applied on the subtree of $T$ with $v$ as root
```

Algorithm 12 Random disturbed cograph

```
Require: Percent $p$ for series, $0 \neq p \neq 1$.
Require: Percent $pe$ for new edge, $0 \neq pe \neq 1$.
Require: Total number of leafs $l$, $0 < l$.
Require: Prime modules size $ms$
Require: Prime modules count $mc$
$bg \leftarrow$ Random binary graph with leaf count equal to $l$
for all $v \in V(bg)$ do
    Randomly assign a label series, or parallel to $v$, so that the probability for series is $p$.
$CG \leftarrow$ Construct graph applied on $bg$
$pm \leftarrow$ All strong modules $M$ of $CG$ such that $|M| \geq ms$, ordered in increasing order by size
for $i$ in $1$ in $1 \cdots mc$ do
    $cm \leftarrow$ The module at index $i$ in $pm$
    for all vertex pairs $(u, v) \in E(CG[cm])$ do
        Modify $CG$ by adding edge $(u, v)$ with a chance of $pe$
return $CG$
```

The cographs are disturbed in such a way that the expected number of prime modules and size of these modules can be tuned beforehand. By only adding edges within a module, we can guarantee that only vertices in that module can be part of a new prime module. Assuming that the root module is not a part of the first mc modules of size greater than ms, we can therefore also ensure that the root module is not prime.
By specifying how many modules we want and their size, we can also construct modular decomposition with various percentage prime modules.

7 Evaluation

There are a number of parameters that affect the generated graph, as **Random disturbed cograph** takes as arguments the number of vertices in the generated graph, the probability of "series" when generating a random cograph, and the probability for edges within a module, and finally the number and size of the prime modules.
The heuristics used also have some parameters that can be tuned. TabuCol has to have a set number of iterations, and the size of the Tabu list, and the

quotient coloring strategy requires the threshold for applying the heuristic directly, and the heuristic used to color the quotient graph.
How all of these parameters are set is described below.

7.1 Generated data

The generated graphs are made up by creating random graphs for every combination of parameters to **Random disturbed cograph** that get assigned different values. The parameters that get assigned different values are size, module count and series probability, with 4 possible values for graph size, 2 for module count, and 2 for series probability, resulting in a total of 16 different combinations of parameters. The value assigned to these parameters are described in the subsections below.
The generated graphs can be found in this git repository [1], and are stored in the directory "TestGraphs". This directory contains a directory for all the different combinations of parameters, and the directory name encodes the value of these parameters. The first number is the number of vertices in the graph, the second number is the series probability, and the last number the prime module count. For example, the directory "DisturbedCoGraph_1000_35_5" contains graphs with 1000 vertices, that where generated with a series proability of 35%, and that contain 5 prime modules. For every combination so are 15 graphs generated, a number set so that they can be colored in parallel efficiently.
The value these different parameters can take are described below.

7.1.1 Graph size

The number of vertices in the graphs are either 1000,750,500 and 250.

7.1.2 Module count and size

Graphs can have the same number of vertices that are within a prime module, but the size of the individual prime modules can differ. It could be possible that few large prime modules have better performance when using for example the 'Whole prime' strategy compared to graphs with many smaller prime modules. The number of modules tested are 5 prime modules and 10 prime modules.
Ensuring that roughly the same fraction of vertices are within a prime module therefore means that the module size depends on the size of the graph, and the number of prime modules. The module size is set so that $module\ size\ \cdot$

module count is half the size of the graph, meaning that roughly half of the vertices can be expected to be a part of a prime module.

7.1.3 Module edge probability

The edge probability within modules is a constant 50%, so that all top prime modules have roughly the same edge density.

7.1.4 Series probability

The final split is with series probability. A higher series probability yields a graph with more edges, and lower a graph with fewer edges, as a graph join in Algorithm 11 introduces new edges, whereas a graph union in the case of a parallel label keeps the number of edges constant. The series probability tested where 35% and 70%.

7.2 Heuristics

TabuCol contain some parameters that can be tuned. The number of iterations allowed was for these test were set to 10000, making it the slowest algorithm to apply, while still being reasonably fast. The size of the tabu list was also set to 7, as recommended by [14].
For the quotient coloring, the threshold for the number of vertices in a prime module to color directly with a heuristic was set to a constant 20. The heuristic used to in turn color the quotient graph was set to RLF. It is reasonably fast with good performance. The reason the same heuristic was not used to color the quotient graph is because it would in the case of TabuCol possibly result in very excessive computation times, as the modules can be deeply nested.

7.3 Code

The code used to generate the graphs, and the code used to evaluate the different colorings can be found at [1].
The graph coloring is implemented in python 3.10, in "ModularColoring.py". It depends on networkx [6] and ModularDecomposition [10]. The script is invoked as

```
python3 ModularColoring.py <input-file>
```

Here, <input-file> is interpreted as a path to a file containing the graph to color, encoded as a networkx adjacency list [12].

The script colors the graph described in the file with a every combination of heuristic described in section 4, and for every strategy defined in section 5. The result is printed on stdout, formatted as comma separated lines suitable for a csv file. The first value is the name of the input file, the second value is the modular decomposition label for the root vertex in the modular decomposition. 1 Means parallel, 0 series and p prime. The third column is the name of the heuristic tested, the fourth column is the name of the strategy used, the fifth column is the number of colors used in the coloring, and the last column is the amount of time the coloring took in seconds.

8 Results

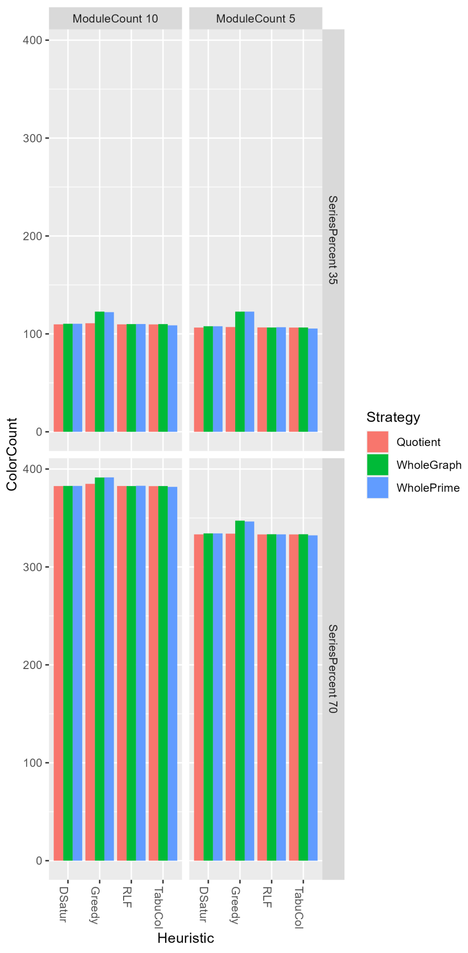
The following figures contain the averaged performance for coloring the graphs in the respective test sets. All graphs are colored with all combinations of heuristic and strategy. A figure is also provided that displays the total average between all disturbed cographs, as well as the performance on DIMACS graphs.

8.1 Interpreting the figures

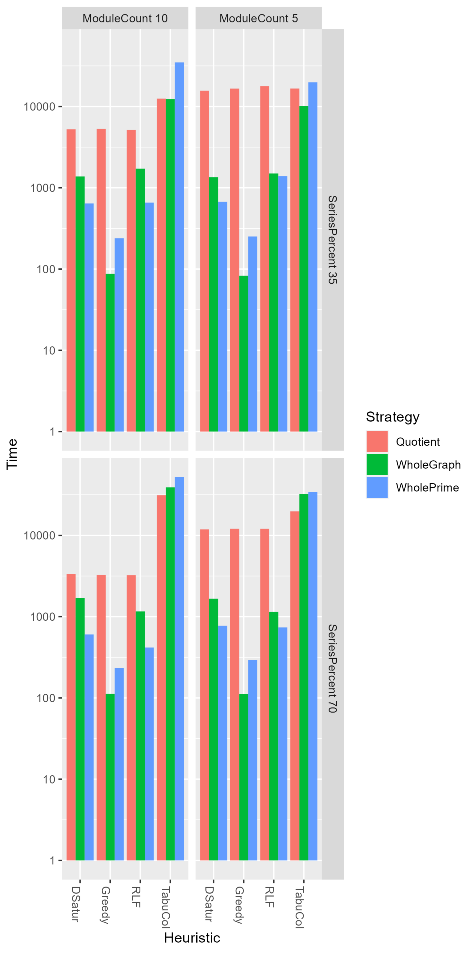
The result for the different categories are displayed grouped by their graph size, except for DIMACS and the table containing the averaged result. These groups in turn divide the result into separate groups depending on the number of modules in the graph, and the series probability. This means that each figure contains the result for 4 different graph categories.
Two figures are displayed for each category, one displaying the number of colors used for a given heuristic, and the other displaying the time in milliseconds it took for that heuristic to color the graphs. The heuristics are displayed on the x-axis, and the time/colors used are displayed on the y-axis. Time is displayed on a logarithmic y-axis, because of the large difference in time taken for different combinations of heuristic and strategy. The strategy used in the coloring is represented by the color of the staple for a given heuristic. The result displayed is the averaged color count/time for every graph in the category.
This means, that to determine the average colors used for RLF on graphs with 1000 vertices that contain 10 prime modules and with 35% series probability using the 'Whole prime' strategy, one would look at the blue staple in the third column in the upper left facet in Figure 25a.
For the DIMACS graphs so are only the "quotient" and "whole graph" strategy displayed, as all of the graphs have a prime root module so is the "whole

graph" strategy the same as the "whole prime" strategy.
When interpreting the runtime for the different algorithms, note that system load may vary between different runs, and that because of the long runtime for some test sets so are the tests not repeated to ensure statistical significance. The time should therefore be seen as a rough estimate.
The implementation and the tested graphs can be accessed as a git repository at [1].

8.2 Results

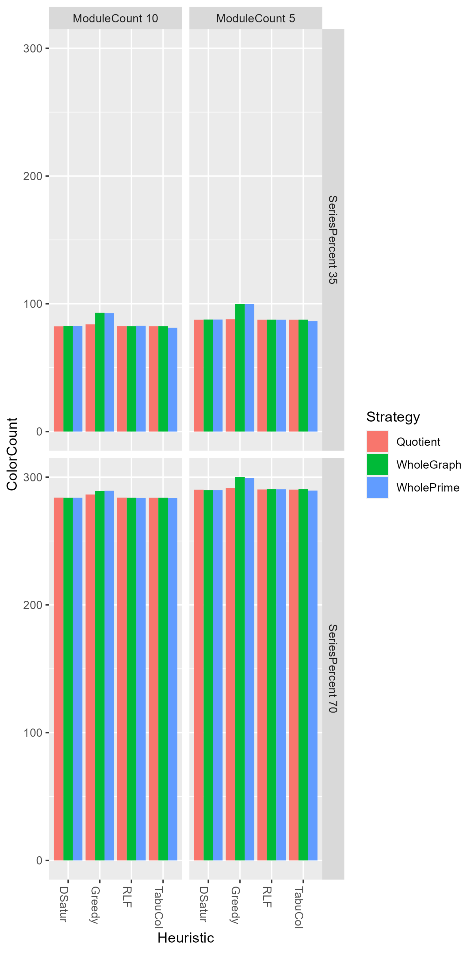


(a) Coloring results

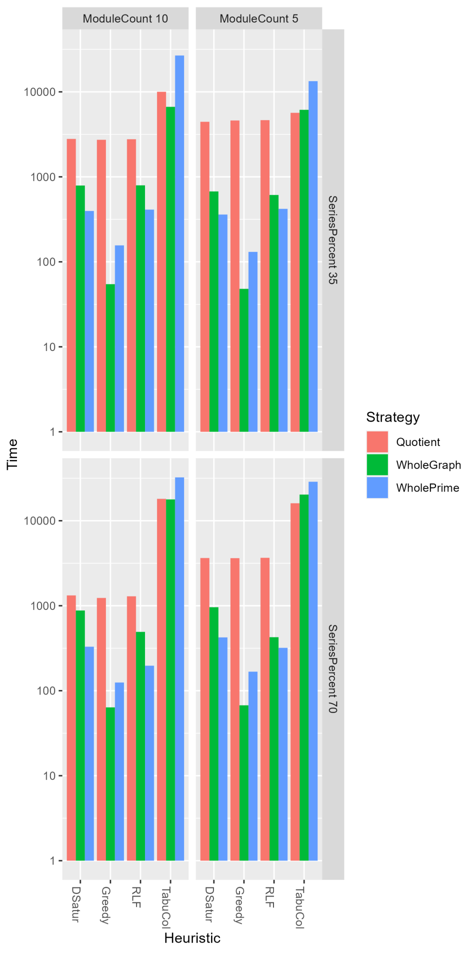


(b) Runtime

Figure 25: Result for generated graphs with 1000 vertices. Graph coloring algorithm is on the x-axis, color/time is on the y-axis, staple colors represent the different strategies. Result faceted by Module size and series probability.

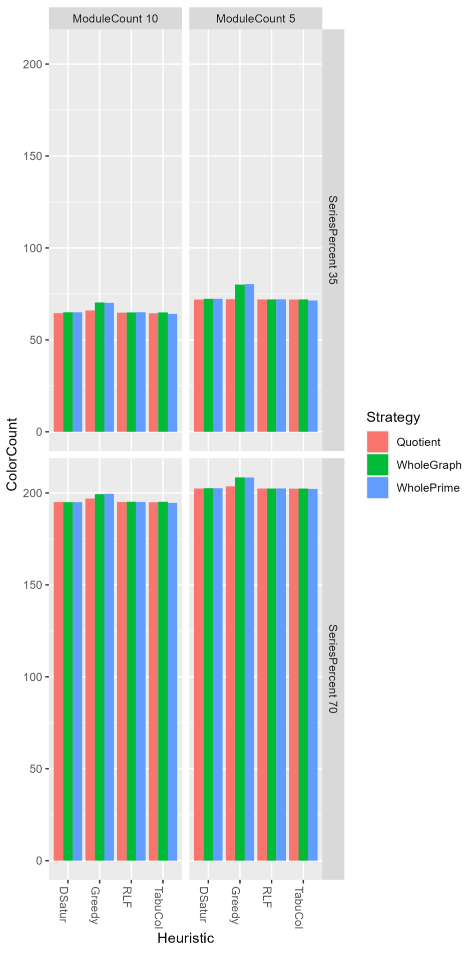


(a) Coloring results

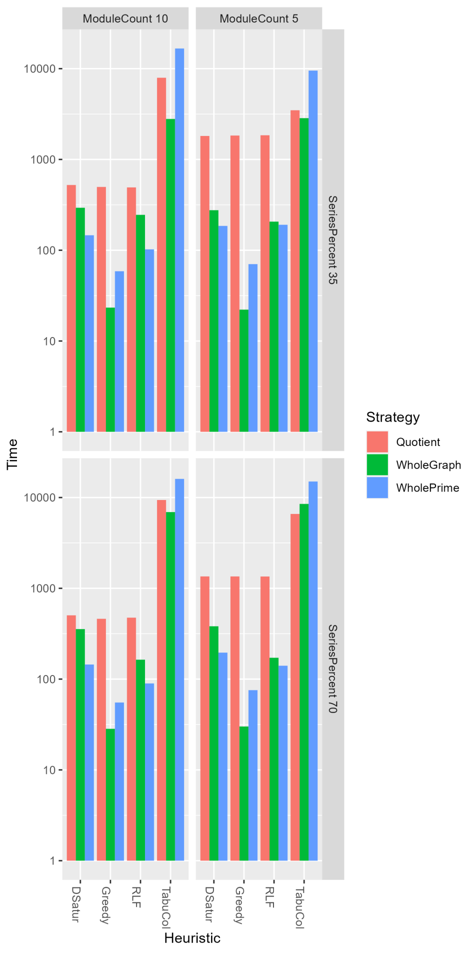


(b) Runtime

Figure 26: Result for generated graphs with 750 vertices. Graph coloring algorithm is on the x-axis, color/time is on the y-axis, staple colors represent the different strategies. Result faceted by Module size and series probability.

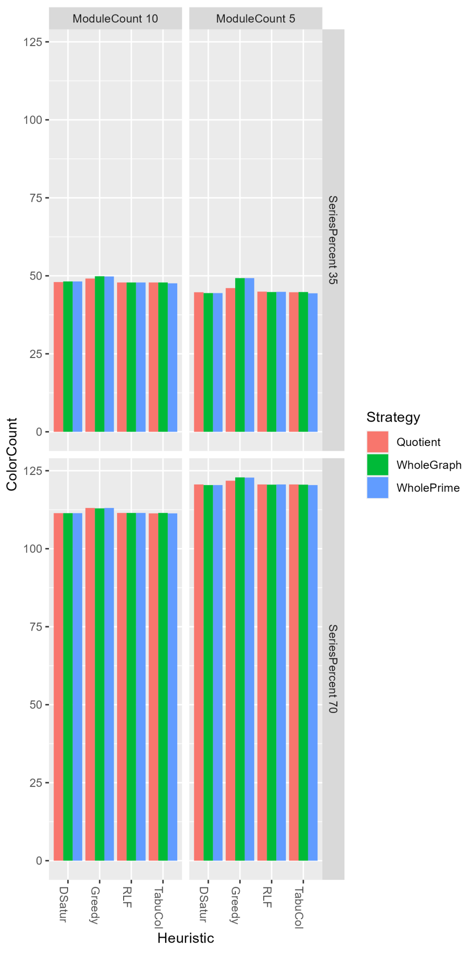


(a) Coloring results

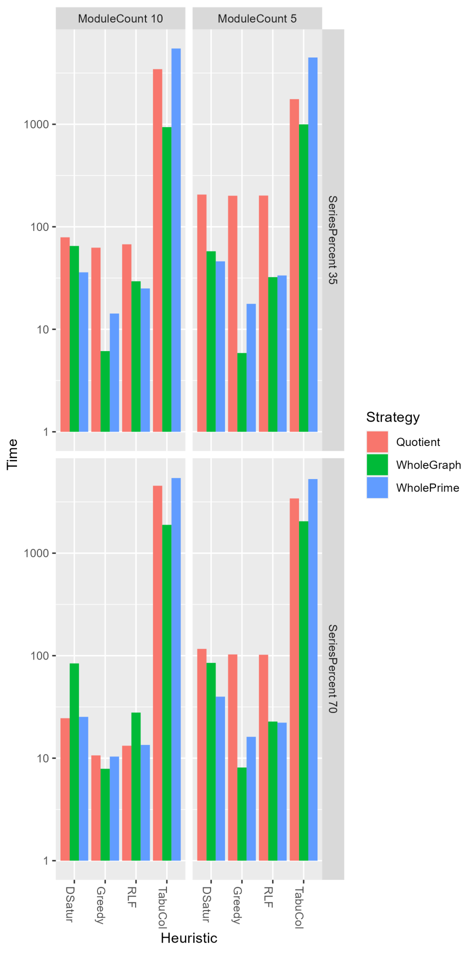


(b) Runtime

Figure 27: Result for generated graphs with 500 vertices. Graph coloring algorithm is on the x-axis, color/time is on the y-axis, staple colors represent the different strategies. Result faceted by Module size and series probability.

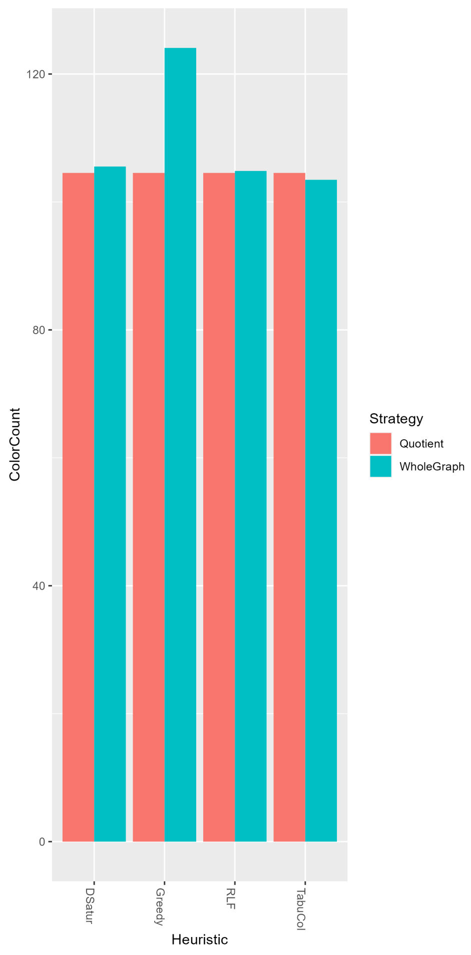


(a) Coloring results

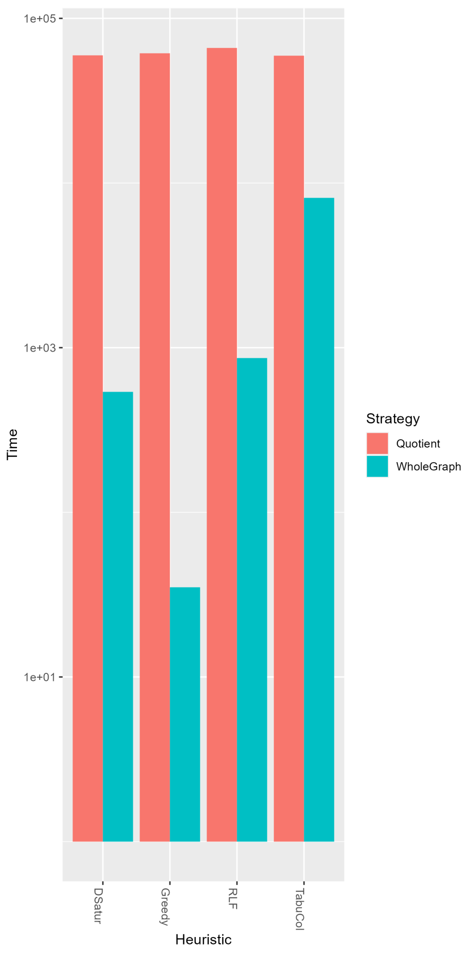


(b) Runtime

Figure 28: Result for generated graphs with 250 vertices. Graph coloring algorithm is on the x-axis, color/time is on the y-axis, staple colors represent the different strategies. Result faceted by Module size and series probability.

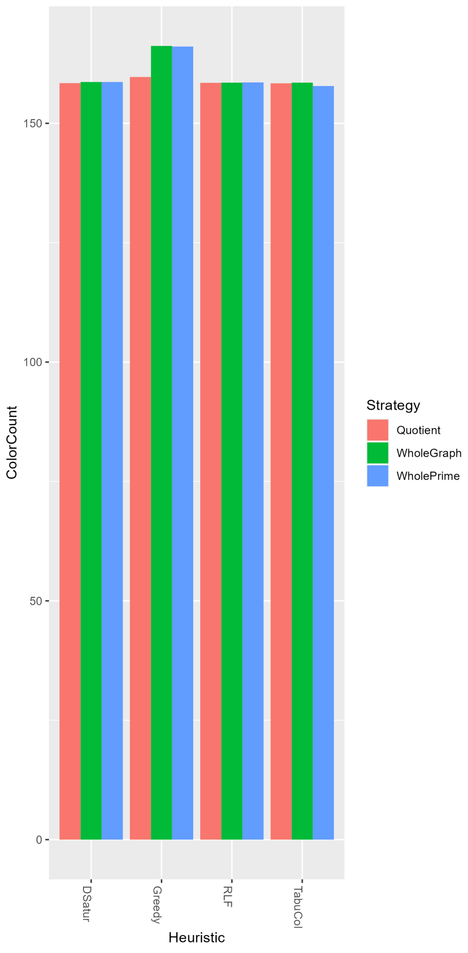


(a) Coloring results

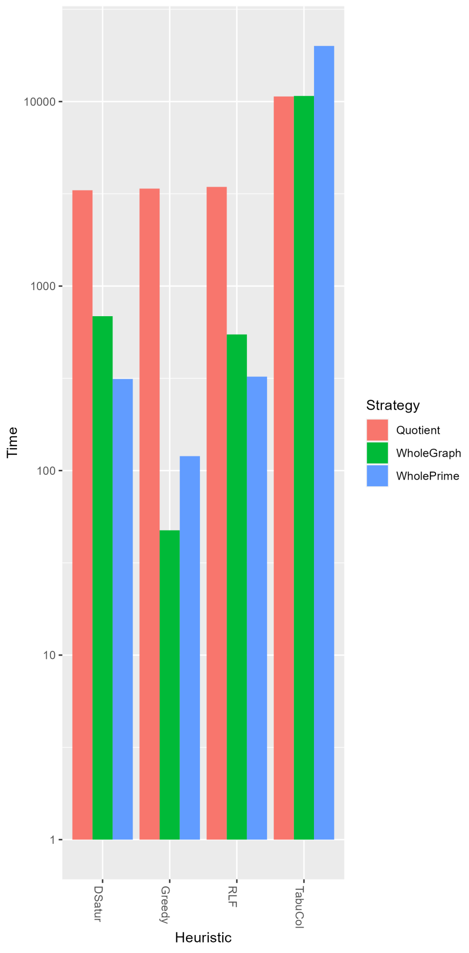


(b) Runtime

Figure 29: Result for the DIMACS graphs. Graph coloring algorithm is on the x-axis, color/time is on the y-axis, staple colors represent the different strategies.



(a) Coloring results



(b) Runtime

Figure 30: Combined results for all generated graphs. Graph coloring algorithm is on the x-axis, color/time is on the y-axis, staple colors represent the different strategies.

8.3 Discussion

Something that we can see from the tables, most easily from Figure 30, is that applying a heuristic locally on prime modules, that is using the 'WholePrime' strategy, does not significantly affect the resulting number of colors for RLF,Greedy and Dsatur. It does however seem to give a minor improvement for TabuCol. In situations where one would consider using TabuCol with its comparably larger execution time in the first place, so could it be reasonable to use the modular decomposition for increased performance. We can see that the number of colors used for DSatur and RLF was for these graphs mostly the same independent of used strategy, and TabuCol had slightly better performance compared to those two. Greedy was by far the worst performing with significantly worse results than all the others in all cases, but it also has significantly faster runtime.
Looking at the difference between series probability of 35% versus 70%, it seems like the relative improvement when using TabuCol instead of another coloring heuristic is very slightly better for the sparser graphs with 35% series probability. There also does not seem to be any direct difference with the number of prime modules in the resulting number of colors used by TabuCol. The result for the quotient strategy is always comparable to the performance of RLF. It can be seen to outperform pure RLF very slightly for some combinations, such as in Figure 29. It also exhibits a much larger runtime, most notable for the same DIMACS test set in Figure 29. The potential decrease in colors most likely does not compensate for the increased runtime however, even with a more optimized implementation.
Looking at the time taken to color the graphs, we can see that the modular decomposition might have a use in decreasing the execution time. We can see that for DSatur and RLF so is the 'WholePrime' strategy almost always faster than coloring the 'WholeGraph' strategy, by a significant amount. This difference seems to be affected by both series probability and the number of graph modules, as well as the graph size. A higher series probability gives a bigger difference in performance for 'WholeGraph' and 'WholePrime' in most cases, and more modules gives an even bigger difference. The only situations where 'WholeGraph' was faster were for RLF on the smaller graphs with 250 vertices.
Graph coloring heuristics with execution time that does not scale linearly with the number of edges/vertices could theoretically see a speedup when applied on smaller subgraphs instead of the whole graph, given that combining the results is in linear time, which is the case for the 'WholePrime'

strategy. Both DSatur and RLF does not scale linearly with the number of edges, whereas TabuCol and Greedy does, which could explain why these heuristics does not see the same benefit, and instead take longer time to execute. The more vertices and the more edges a graph has the larger this difference would be , which would explain why the difference is higher for 70% series probability, and why it can even take longer in the case of RLF on graphs with 250 vertices, the overhead of creating the subgraphs is not yet compensated by the speedup from applying on smaller graphs.
Whether or not it is worth it in practice to use the modular decomposition for a speedup in execution time depends on the ability to implement the modular decomposition tree in linear time, and if the added computations can be compensated by large/dense enough graphs.
Another possible advantage with the modular decomposition is that it could allow for more easily implemented parallelised coloring algorithms. As the child modules of a module in the modular decomposition tree creates a partition of the parent module, so can the different parts of the modular decomposition tree be colored in parallel without causing race conditions, which could compensate for the increased runtime when for example using TabuCol with the 'WholePrime' strategy.

9 Conclusion

Using the modular decomposition did not significantly improve the performance of the tested heuristics, except in the case for TabuCol. This does show that performance benefit can vary from heuristics to heuristic, but the total difference is also most likely relatively small. The modular decomposition coloring does however show some potential in improving the runtime for different coloring heuristics, and combined with parallelised coloring of prime modules might lead to larger improvements when examining colors relative to time taken.
The coloring was mostly improved when using TabuCol, and it is possible that other similar algorithms could see an improvement as well. It is also possible that there are other ways to utilise the modular decomposition when coloring prime modules other than the quotient strategy.
Another area worth investigating is finding examples of real-world graphs that have modular decomposition without a prime root module. The parameters set for the generated graphs might be different from graphs encountered in real-world use cases, and could exhibit different properties that make them more/less suitable for these coloring methods.

References

[1] Emanuel Berggren. *Graph coloring using modular decomposition*. https://github.com/MrBoboGet/BachelorThesis. 2023.

[2] D.G. Corneil, H. Lerchs, and L.Stewart Burlingham. "Complement reducible graphs". In: *Discrete Applied Mathematics* 3.3 (1981), pp. 163–174. ISSN: 0166-218X. DOI: https://doi.org/10.1016/0166-218X(81)90013-5. URL: https://www.sciencedirect.com/science/article/pii/0166218X81900135.

[3] *DIMACS coloring benchmarks*. http://archive.dimacs.rutgers.edu/pub/challenge/graph/benchmarks/color/. Accessed: 2023-04-24.

[4] T. Gallai. "Transitiv orientierbare Graphen". In: *Acta Mathematica Academiae Scientiarum Hungarica* 18.1 (1967), pp. 25–66. ISSN: 1588-2632. DOI: 10.1007/BF02020961. URL: https://doi.org/10.1007/BF02020961.

[5] Michel Habib and Christophe Paul. "A survey of the algorithmic aspects of modular decomposition". In: *Computer Science Review* 4.1 (2010), pp. 41–59. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2010.01.001. URL: https://www.sciencedirect.com/science/article/pii/S157401371000002X.

[6] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: (2008). Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman, pp. 11 –15.

[7] Marc Hellmuth and Guillaume E. Scholz. "From modular decomposition trees to level-1 networks: Pseudo-cographs, polar-cats and prime polar-cats". In: *Discrete Applied Mathematics* 321 (2022), pp. 179–219. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2022.06.042. URL: https://www.sciencedirect.com/science/article/pii/S0166218X22002359.

[8] Richard M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9. URL: https://doi.org/10.1007/978-1-4684-2001-2_9.

[9] R. M. R. Lewis. "Bounds and Constructive Heuristics". In: *Guide to Graph Colouring: Algorithms and Applications*. Cham: Springer International Publishing, 2021, pp. 39–76. ISBN: 978-3-030-81054-2. DOI: 10.1007/978-3-030-81054-2_3. URL: https://doi.org/10.1007/978-3-030-81054-2_3.

[10] Anna Lindeberg. *Modular Decomposition*. https://github.com/AnnaLindeberg/ModularDecomposition. 2023.

[11] Ross M. Mcconnell and Jeremy P. Spinrad. "Ordered Vertex Partitioning". In: *Discrete Mathematics & Theoretical Computer Science* Vol. 4 no. 1 (Jan. 2000). DOI: 10.46298/dmtcs.274. URL: https://dmtcs.episciences.org/274.

[12] NetworkX. *Adjacency List*. https://networkx.org/documentation/stable/reference/readwrite/adjlist.html. 2023.

[13] Dulce I. Valdivia, Manuela Geiß, Maribel Hernández Rosales, Peter F. Stadler, and Marc Hellmuth. "Hierarchical and Modularly-Minimal Vertex Colorings". In: (2020). arXiv: 2004.06340 [math.CO].

[14] D. de Werra. "Heuristics for Graph Coloring". In: *Computational Graph Theory*. Ed. by G. Tinhofer, E. Mayr, H. Noltemeier, and M. M. Syslo. Vienna: Springer Vienna, 1990, pp. 191–208. ISBN: 978-3-7091-9076-0. DOI: 10.1007/978-3-7091-9076-0_10. URL: https://doi.org/10.1007/978-3-7091-9076-0_10.

[15] S.G. Williamson and Bender E.A. *Lists, Decisions and Graphs*. S. Gill Williamson. URL: https://books.google.se/books?id=vaXv_yhefG8C.

[16] R.J. Wilson. *Introduction to Graph Theory*. Longman Scientific & technical. Longman, 1985. ISBN: 9780582446854. URL: https://books.google.se/books?id=1-nuAAAAMAAJ.

Datalogi
December, 2023
www.math.su.se

Beräkningsmatematik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm