An evaluation of special case 3-coloring algorithms

Schmeink, Emiel

Award date:
2023

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Algorithms, Geometry & Applications

An evaluation of special case 3-coloring algorithms

Master's thesis

Emiel Schmeink

20 July 2023

Supervision:
dr. ir. M.J.M. Roeloffzen

Assessment committee:
Supervisor
dr. ir. I. Barosan
dr. L. Ryvkin

This is a public Master's thesis.

This Master's thesis has been carried out in accordance with the rules of the TU/e Code of Scientific Conduct.

Abstract

This thesis will compare the performance of different algorithms for the 3-coloring problem (generic 3-coloring algorithms, and 3-coloring algorithms for specific graph types). Afterwards, we will analyze the difference between what is expected from the theoretical analyses of the algorithms and what we see in practice. As far as we know, there is no existing literature on a comparison of non-generic algorithms and generic algorithms for the 3-coloring problem because it requires a lot of effort to both find graph instances for specific graph classes, and correctly implement different types of algorithms for specific graph classes. We tackle this problem by first creating a system that can generate graph instances with certain requirements of arbitrary size, and afterwards implementing the algorithms for specific graph classes. To answer the question of what we see in practice, we implemented three non-generic algorithms for the planar triangle-free, locally-connected, and (P_7, C_3)-free graph classes, together with two generic algorithms, and benchmarked them on generated graphs of sizes up to 50000 vertices, with the largest graphs containing around 275 million edges. In the end, we conclude that it is not worth it to implement non-generic algorithms. If a fast and space-efficient implementation for converting a graph into a SAT formula is available, solving the 3-coloring problem using a SAT solver is the most efficient algorithm to do so. Special case algorithms seem to be less efficient, and more error-prone.

Contents

Chapter 1

Introduction

1.1 Background and motivation

The coloring problem is one of the earliest documented NP-complete problems in computer science, as it appeared as one of Karp's 21 NP-complete problems in 1971. In its most basic form, the coloring problem is an optimization problem that asks for a given graph how many colors you need to color each vertex such that none of its neighbors have the same color.

A real-world example of the coloring problem is in the form of a cell tower frequency assignment. You can imagine that cell towers have a specific range and frequency on which they operate. If these cell towers have an area in which the coverage overlaps, and they operate on the same frequency, destructive interference occurs in this overlapping area. Therefore it is paramount that overlapping areas have at least one unique frequency. Trying to minimize the number of frequencies required, is also called the Conflict Free Coloring (CF-coloring) problem.

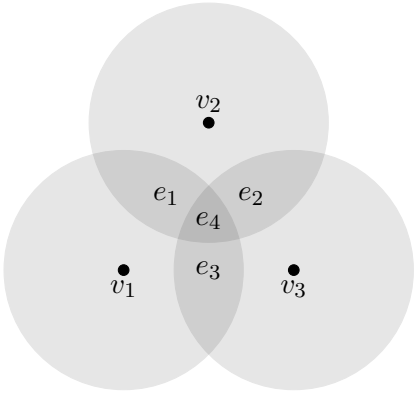


Figure 1.1: Abstraction of 3 cell towers with coverage

In figure 1.1 we can see an example of 3 cell towers with overlapping areas. We can think of the cell towers as vertices and the overlapping areas

as hyperedges in a hypergraph, with the colors representing the frequencies available. Here the hyperedges have to contain at least one uniquely colored vertex to have phone service in that overlapping area.

Besides real-world applications, coloring problems are also abundant in theoretical computer science. These also exist as decision problems of which many variants appear, like edge k-coloring, list k-coloring, and many others. However, there is one version, vertex k-coloring, which is colloquially known as the k-coloring problem. Here the problem is to assert if for a given graph the vertices can be colored using at most k colors, and so that there are no edges with endpoints that have the same color. This is also the version we will be discussing in this thesis.

In general, the k-coloring problem is NP-complete, so no polynomial algorithms exist. However, for special cases, polynomial (or better) algorithms exist (N.B. "special case algorithms" and "non-generic algorithms" will be used interchangeably throughout this thesis). A lot of research has been done on these special case algorithms, as can be seen in 1.3. In this thesis, we will be looking into the practical use of these special case algorithms, not the theoretical use, where it is clear they have for example a better runtime than the general cases. This means looking into the feasibility of implementing these special case algorithms, as well as looking at the available real-world graphs for these special cases. In theory, these special case algorithms have a much better runtime. Nonetheless, in practice, a lot of variables come into play in whether these special case algorithms actually perform better. For example, things like hidden variables in the algorithm's big O notation, or the complexity of actually implementing the special case algorithms may slow down these special case algorithms tremendously in practice.

In this thesis, we will be exploring if in practice these special case algorithms are actually faster like the theory predicts, or that theory does not match practice (even though in theory, practice matches the theory).

1.2 Contributions and organization

This paper explores the performance and feasibility of 3-coloring algorithms for specific graph classes, the (P_7, C_3)-free, locally connected, and planar triangle-free graphs. We compare generic algorithms that theoretically run in exponential time, but in practice are highly optimized, with algorithms that only work for the aforementioned graph classes. In chapter 2 we explain the methods of how the test graphs are generated, with some small examples. In chapter 3 we explain the main ideas behind the generic algorithms and give which parts of the implemented algorithms differ from the ideas. Chapter 4 does the same only then for the non-generic algorithms. In chapter 5 we outline which experiments we will run and on which graphs.

We also give more context about what the graphs look like and some limitations about the generation of the graphs, together with which software and hardware were used. The results in chapter 6 are the results of the experiments executed, with some explanations of what we can see. Lastly, chapter 7 covers the conclusions and limitations of the results, and what lessons we can take away from these conclusions.

1.3 Related work

Much research has been done on the 3-coloring problem, as it was one of Karp's 21 NP-complete problems described in 1971 in the form of the chromatic number problem. Therefore a lot of papers can be found, not only for our specific special case algorithms but for all types of problems related to 3-coloring. The papers found on 3-coloring can be divided into 9 categories: General Algorithms, Non-general Algorithms, Known 3-coloring algorithms, Constructing graphs, Parallel Algorithms, Counting Algorithms, Survey Papers, and Proofs without algorithms. From these categories, the immediately relevant ones are the General Algorithms and Non-general Algorithms, since this is what we want to compare. The Known 3-coloring polynomial algorithms, Parallel Algorithms, Counting Algorithms, and Proofs without algorithms will not be covered in this thesis since we want to focus on the comparison of general vertex 3-coloring algorithms versus the vertex 3-coloring algorithms for specific graph classes. The "Constructing graphs" and "Survey papers" are useful in obtaining datasets and ways to create graphs for specific graph classes. The Proofs without algorithms are not directly relevant to the algorithms that will be discussed in this thesis.

1.3.1 General Algorithms

The paper by Beigel and Eppstein ([3]) provides algorithms for 3-coloring, 3-edge-coloring, and 3-list-coloring based on Symbol-System Satisfiability, which runs in $O(1.3446^n)$ time.

Then they expand on the bound of their algorithms in [4], where they achieve a running time of $O(1.3289^n)$.

An improvement of 20% of some random graphs on the DSATUR approximation algorithm is presented by Sager and Lin ([25]) by giving pruning procedures for the search tree. This procedure works for all graphs, but only improves the running time for some.

An almost exact approximation algorithm is presented by Abu-Khzam and Langston ([1]), which runs in $O(1.277^n)$ and is based on maximal independent set enumeration.

1.3.2 Non-general Algorithms

An polynomial algorithm is given for k-coloring degenerate Berge graphs for fixed k, by Haddadène and Maffray ([14]).

In the paper by Zang an $O(mn)$ time algorithm is given for odd-K_4-free graphs, where m and n are the number of edges and vertices of an arbitrary graph G, respectively.

Dvořák et al. also created a linear algorithm on a variation of the triangle-free planar graphs, namely the triangle-free surface-drawn graphs in [13]. They also created a linear algorithm for triangle-free planar graphs in [12].

A linear algorithm for locally connected graphs for both 3-coloring and 3-clique-coloring is presented by Kochol ([18]).

For graphs with a minimum degree of at least 15 an algorithm that runs in $O(1.296^n)$, was given by Narayanaswamy and Subramanian ([24]) by enumerating 3-colorings of the dominating set and solving the resulting 2-list coloring instance.

The graph classes (O_4, C_4)-free, $(K_{1,3}, O_4)$-free, $(K_{1,3}, O_4, K_2 + 2K_1)$-free algorithms have been create by Malyshev ([22]). Where K_n, C_4, and O_n represent the induced complete subgraphs, induced cycles, and empty subgraphs.

The $(2P_4, C_5)$-free [15], (P_7, C_3)-free [7], and small diameter graphs [11] have algorithms that run in polynomial, polynomial, and sub-exponential time respectively.

The (P_7, C_3)-free, locally connected, and planar triangle-free cases will be discussed in detail later in this thesis.

1.3.3 Others

Known 3-coloring algorithms know that a graph is 3-colorable, and find the fastest way to 3-color the graph.

The paper by Blum ([5]) provides an approximation algorithm which uses $O(n^{2/5} \log^{8/5} n)$ colors and runs in $O(n^{0.4})$. This runtime is later improved in [6] to $O(n^{3/14})$. Later the bound on colors is improved to $O(n^{4/11})$ by Kawarabayashi and Thorup ([17]).

A spectral technique for a polynomial algorithm that colors a 3-colorable graph with high probability is given in the paper by Alon and Kahale ([2]).

In the paper by Li and Xu ([20]) methods are given to generate graphs that have a unique k-coloring, i.e. there is only one coloring for a certain amount of colors.

Parallel algorithms for 3-coloring, maximal independent set, and maximal matching are given in the paper by Latypov and Uitto ([19]) for trees that run in $O(\log^2 \log n)$ using the Massively Parallel Computation model.

There are also algorithms for finding the number of 3-colorings for a given graph. Such as the algorithm for counting the number of 3-colorings

that runs in $O(1.6262^n)$ by V and Gaspers ([26]). This is later improved to $O(1.588^n)$ by Zhu et al. ([28]).

An overview of computational results for coloring problems can be found in the survey paper by Malaguti and Toth, as well as a list of graphs used for benchmarking.

Many papers prove for a certain graph class, that the 3-coloring problem can be solved in polynomial or linear time. One such paper proves by Karthick et al. ([16]) that the (P_5, dart)-free, (P_5, banner)-free, (P_5, bull)-free, and $(\text{fork}, \text{bull})$-free graph classes can be solved in polynomial time.

However, there are also many graph classes for which the 3-coloring problem remains NP-hard, as proven for 4-regular planar, 5-regular planar, p-regular, q-ordered regular Hamiltonian graphs for every $p \geq 6$ and for every $q \geq 3$ by Cavallaro and Fluschnik ([9]). All of these graph classes remain NP-hard to solve.

From the general algorithms found, we selected only the current fastest running time algorithm ([4]), since this will be used together with a SAT solver and the DSATUR algorithm. More general algorithms would be preferred, but this is not feasible within the scope of the project. For the non-general algorithms, the decision was made only to use "normal" algorithms, so for example, no approximation algorithms, nor dynamic algorithms to keep tabs on the scope of the project. The specific non-general algorithms that will be discussed in this project are all based on specific classes of graphs. The classes covered by these algorithms are $2(P_4, C_5)$-Free, (P_7, C_3)-Free, Small diameter, Locally Connected, and (C_3)-Free Planar graphs. Here a class of (P_n, C_k)-Free graphs means the graphs without an induced path of length n, and without induced cycles of size k.

However, since the 3-coloring problem specifically is a decision problem and most real-world problems involving a coloring problem require optimization, not a decision problem, there are not many implementations of 3-coloring specifically. The currently available implementations we could find were the DSATUR greedy algorithm presented in [8], and one could argue that using a reduction to 3-SAT with a SAT solver is also an implementation, in which case the SAT-solver z3 [23] is also a viable implementation.

Chapter 2

Graph generation

2.1 Overview

Since the special case algorithms require a certain graph type, it is paramount that we can create these graphs for a certain size and density. Initially, we wanted to use real-world graphs, but most graphs found were not 3-colorable, let alone meet any of the requirements for the graph types. So this was not an option. Then the Erdös-Renyi model for generating graphs was considered. This model simply takes n vertices and a probability p and then checks for each possible edge, so $\binom{n}{2}$ edges if it should be added or not with a probability p. This however is not enough to get a graph with any requirement, simply because the chance that it will suffice for a requirement is very low. So instead we check during the edge creation if a new edge is valid for the given requirement, then we have a graph that is guaranteed to adhere to the requirement given. One drawback is that the density of p is not guaranteed anymore, but this could be fixed by trying to add more valid edges after creation. In the next sections, we will be discussing the checks needed and how they are implemented.

2.2 Induced Path

The algorithms that required a (P_n)-free graph type, could not have induced paths of size n (thus the path contains n vertices). This means that during creation we have to check if a new edge e with vertices (u, v) adds an induced path of size n. We recursively get all induced paths of size $n-1$ from u and v in the graph without e. Then for each combination of paths where $|path_u| + |path_v| = n$ is checked for crossing edges, meaning checking if any of the vertices in $path_u$ have an edge to any of the vertices of $path_v$. Here $path_u$ and $path_v$ are induced paths with starting points u and v respectively. If no crossing edges are found, we have found an induced path and the edge is not valid. Pseudocode for the algorithm to find all induced paths for a starting vertex up to l can be found here 1, with the initial call G being the

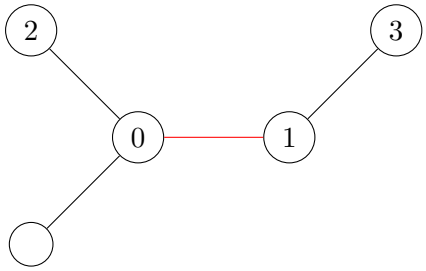


Figure 2.1: An example where a new edge $(0, 1)$ would create an induced path of size 4 with vertices $\{2, 0, 1, 3\}$

graph to find the induced paths in, s is a list containing the starting vertex, and l the maximum length of the induced path.

Algorithm 1 Find all induced paths up to length l from a given vertex

```
1:  procedure FindInducedPaths(G, s, l)
2:      init empty list L
3:      if l = 0 then
4:          return {s}
5:      end if
6:      for all neighbors v of last vertex z of s do
7:          if v is predecessor of z then
8:              skip
9:          end if
10:         if any neighbor of v is in s then
11:             skip
12:         end if
13:         L ∪ FindInducedPaths(G, s + v, l − 1)
14:     end for
15:     return L ∪ {s}
16: end procedure
```

Since we have an invariant that there is no induced path, we know that if an induced path is added it must go through the new edge. This means that we can get all induced paths of size $n - 1$ from u and v in the graph without e since at least the other vertex of the edge must be in the induced path if it exists, and we do not care about paths through the new edge, since they would be covered by the induced path finding of the other vertex of the edge. We then can combine the paths found from u and v, to find induced paths of size n. These paths are induced paths, but vertices in a path from u can have edges to a vertex in a path from v, and therefore we need to check for these crossing edges.

2.3 Induced Cycle

The algorithms that required a (C_k)-free graph type, could not have induced cycles of size k (thus the cycle contains k vertices). The approach to check whether an edge violates this requirement is similar to the induced path check. We also know that if a new cycle would be added, it must go through the new edge e with endpoints u and v. We can use the same logic as with the induced path check, only we need to check that an edge exists between the last vertex of a $path_u$ and a $path_v$, where $path_u$ and $path_v$ are also induced paths with starting points u and v.

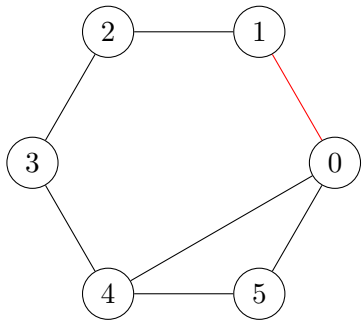


Figure 2.2: An example where a new edge (0, 1) would create an induced cycle of size 5 with vertices $\{1, 2, 3, 4, 0\}$

2.4 Planarity

Some algorithms also require that the graph is planar, which means that it can be drawn without crossing edges. This can be checked by doing the Left-Right Planarity Test published in [10]. This check checks for the entire graph if it is planar, and implementations of this check can run in linear time.

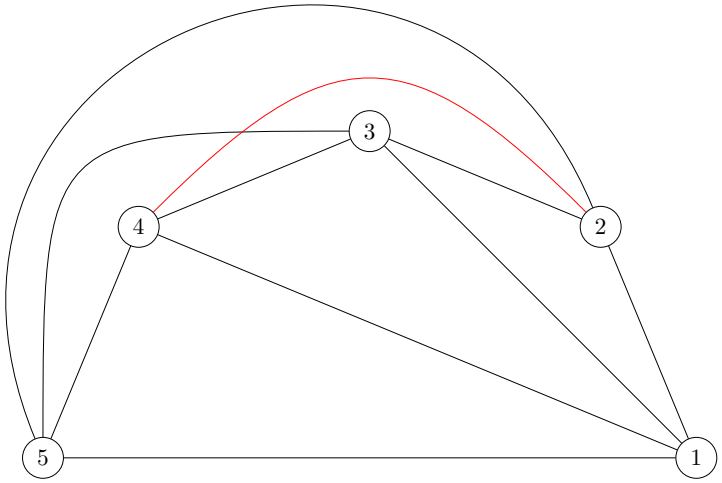


Figure 2.3: An example where a new edge (2, 4) would create a non-planar graph since a K_5 is non-planar

2.5 Diameter

Some algorithms require that the graph does not exceed a certain diameter. The diameter of a graph is the largest length of any shortest path between each pair of vertices. This can thus be calculated by getting all-pairs shortest paths. Even though at first glance this might seem to be slow, currently the graph vertex count is relatively low, but the amount of edges is quite high. And because the time complexity of the all-pairs shortest path problem is based on vertices, and most other checks on the number of edges, this check will not be the bottleneck.

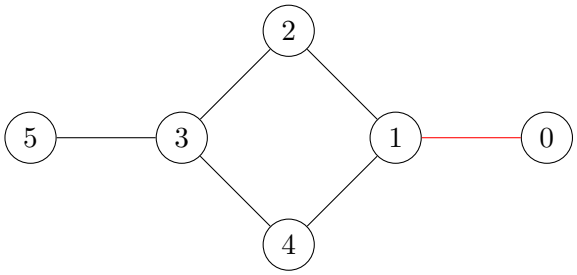


Figure 2.4: An example where a new edge (0, 1) would increase the diameter from 3 to 4 since the shortest path from 5 to 0 has 4 edges

2.6 Locally connected

Locally connected graphs have the property that for all vertices the open neighborhood is connected. Recall that the open neighborhood of a vertex v is all neighbors of v without v. These graphs are easily generated by using the normal Erdös-Renyi model, and afterward looping over all vertices and connecting the open neighborhood if they are not locally connected already. For example, in figure 2.5 we can see that edge $(1, 6)$ would make the open neighborhood of vertex 0 connected.

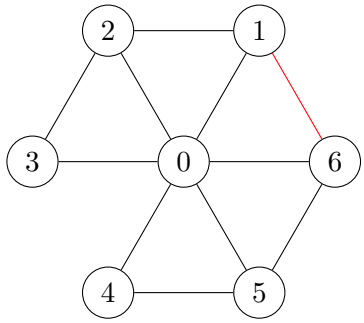


Figure 2.5: An example where a new edge (1, 6) would make the graph locally connected

Chapter 3

Generic Algorithms

Here we will give a high-level overview of the implemented generic algorithms. We will also compare them to the original ideas from the papers the algorithms originate from, or optimizations we used. We will also give a high-level overview of some non-exact algorithms.

3.1 Reduction to SAT

3.1.1 High-level overview

Recall that by SAT we mean the satisfiability problem, a fundamental problem that is defined as solving the problem of checking if a boolean formula can be evaluated to be true, and thus is satisfiable. An example of a boolean (SAT) formula can be seen in equation 3.1, where the formula contains clauses combined by the boolean "AND" operators and variables in these clauses that are combined by the "OR" operators. If there is a combination of setting the variables to true and false such that all clauses evaluate to true, we have a satisfiable formula. The graph coloring instance can be converted to a SAT instance. This can be done by creating 3 variables for each vertex (representing the 3 different colors) and creating clauses that specify that exactly one of these variables per vertex must be true. Then create clauses specifying that vertices with edges between them cannot have their respective variables both true, as this would represent the two vertices having the same color, which is not allowed. This results in the following propositional formula for vertices $\{v_0, ..., v_n\} \in G$, where the indices of the variables indicate which vertex they correspond to:

$$\begin{aligned} \bigwedge_{v_i \in V(G)} \Big((r_i \vee g_i \vee b_i) \wedge (\neg r_i \vee \neg g_i) \wedge (\neg r_i \vee \neg b_i) \wedge (\neg g_i \vee \neg b_i) \Big) \wedge \\ \bigwedge_{(v_i, v_j) \in E(G)} \Big((\neg r_i \vee \neg r_j) \wedge (\neg g_i \vee \neg g_j) \wedge (\neg b_i \vee \neg b_j) \Big) \end{aligned} \tag{3.1}$$

The SAT solver that was used to check whether a formula is satisfiable or not is z3 by Microsoft Research ([23]).

3.1.2 Deviation from algorithm & optimalization

For some algorithms, a 3-list-coloring solution is needed. This means that instead of having exactly 3 colors for each vertex, each vertex can have up to 3 colors as options. To reflect this in the proposition formula, a clause is added for each color that is not an option. Note that we do not remove the specific color from the formula, even though this would make the formula smaller. The reason is that the solver we use is highly optimized, and thus efficiently removes the impossible colors during solving. Also, the step that takes the most time during solving process is the conversion from graph to formula, and adding a lot of checks would increase the time taken to create the formula even more. So for example, if for vertex $v_0 \in G$ the color red is not allowed, we get the following propositional formula for that vertex:

$$\begin{aligned}&((r_0 \vee g_0 \vee b_0) \wedge (\neg r_0 \vee \neg g_0) \wedge (\neg r_0 \vee \neg b_0) \wedge (\neg g_0 \vee \neg b_0)) \wedge \\ &\bigwedge_{(v_0,v_j)\in E(G)} \Big((\neg r_0 \vee \neg r_j) \wedge (\neg g_0 \vee \neg g_j) \wedge (\neg b_0 \vee \neg b_j)\Big) \wedge \\ &\neg r_0\end{aligned} \tag{3.2}$$

3.2 General 3-coloring

3.2.1 High-level overview

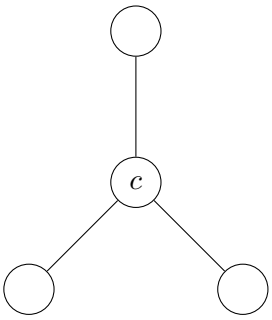
The main ideas of this algorithm are outlined in algorithm 2. For clarity, here we will explain what each step contributes to solving the coloring problem. Trivially, the first step removes all low-degree (two or fewer) vertices from the graph to quickly color at the end. Since these vertices have at most degree two, these vertices will always have an available color to be used in the coloring.

The main part of the algorithm is finding a maximal bushy forest. A bushy forest is a high-degree forest, where all internal vertices have degree greater than 4, and by definition, the leaves have no more than 3 neighbors outside the forest, because this would imply that you can extend the forest by creating a new tree with the leaf as the root. This maximal bushy forest acts as a data structure that covers a lot of the graph, together with the $K_{1,3}$ graphs. We can use these structures to greedily pre-color a large part of the graph, and remove colors from the list colorings of many vertices that have not been pre-colored, before sending it to the SAT-solver. A $K_{1,3}$ can be seen in figure 3.1a. The $K_{1,3}$ graphs can be extended to height-two trees (fig

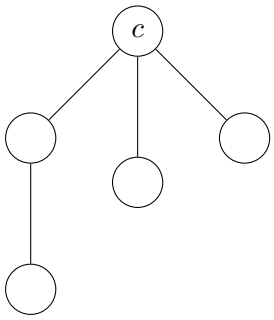
3.1b) by a network flow algorithm, where we can greedily color or remove possible colors from, these extensions.

After these steps, the vertices that do not yet have a definite color are passed to the SAT solver and are checked for satisfiability. In the original paper instead of a SAT solver, a CSP solver is used, which also runs in exponential time. We explain why in section 3.2.2. If it is not satisfiable, try another coloring for the internal vertices until either all options have been tried, or a satisfiable coloring has been found.

Figure 3.1: A $K_{1,3}$ and a height two tree



(a) A $K_{1,3}$ subgraph, with c being the center vertex



(b) A $K_{1,3}$ subgraph, extended to a height two tree

Algorithm 2 Find a 3-coloring for the given graph G by adapting the Algorithm by Beigel and Eppstein ([3])

1: If the input graph G contains any vertex v with degree less or equal to two, recursively color $G \setminus v$ and assign v a color different from its neighbors.
2: Find a maximal bushy forest F in G.
3: Find a maximal set T of $K_{1,3}$ subgraphs in $G \setminus F$.
4: While it is possible to increase the size of T by removing one $K_{1,3}$ subgraph and using the vertices in $G \setminus (F \cup T)$ to form two more $K_{1,3}$ subgraphs, do so.
5: Use a network flow algorithm to assign the vertices of $G \setminus (F \cup N(F) \cup T)$ to trees in T, forming a forest H of height-two trees.
6: Recursively search through all consistent combinations of colors for the bushy forest roots and internal vertices, and for selected vertices in H. For each coloring of these vertices, form a SAT formula describing the possible colorings of the uncolored vertices, and use the SAT solver to attempt to solve this instance. If one of the SAT formulas is solvable, return the resulting coloring. If no SAT formula is solvable, return indicating that no coloring exists.

3.2.2 Deviation from algorithm & optimalization

The steps to recursively color large connected components and cycles of degree three have been omitted. This is because even though this improves the worst case, in practice this means that we have to run the algorithm at least twice for a relatively small amount of vertices that would cover these connected components.

The major difference from the paper is that the vertices that need to be colored in the end, are being colored by a reduction to SAT instead of using a CSP solver. A Constraint Satisfaction Problem (CSP) is in a sense a generalization of SAT, where for example 3-SAT would be (2,3)-CSP because the boolean variables can only take 2 values, true or false, and the clauses have 3 variables inside. In CSP the variables and clauses can be of arbitrary size. The original paper uses CSP because it allows the paper to prove some improvements for the worst case in the algorithm in theory. We use a SAT solver instead of implementing the CSP algorithm from the paper because the reduction to SAT will be much faster in practice, even though in theory the worst case might be worse. This is mostly because the SAT solvers have been highly optimized and run on C instead of Python, which has been used for this project. Also implementing the convoluted CSP solver would take a lot of effort for little gain, and we expect the bottleneck to be the creation of, and the search through all combinations of colorings of the vertices of the bushy forest, not the actual solving.

3.3 Non-exact algorithms

We also experimented with non-exact algorithms to see how the exact algorithms would perform in comparison to the (supposedly) faster heuristic algorithms. Here we will be giving an overview of the heuristic algorithms that have been benchmarked.

DSATUR DSATUR is a heuristic algorithm for the graph coloring problem. It is greedily coloring the vertices using an ordering of "degree of saturation" (hence the name). This degree of saturation is based on the number of different colors the neighbors of a vertex have used. In case of a tie, use the largest degree.

Largest first This is even simpler, using a reverse ordering of the degrees of the graph and coloring them in that order, picking a new color if necessary.

For both heuristic algorithms, we used the implementations given by the NetworkX package.

Chapter 4

Non-generic Algorithms

Here we will explain the implemented algorithms for non-generic graph types, and compare them to the original ideas from the papers the algorithms originate from.

4.1 Triangle-free planar graphs

4.1.1 High-level overview

A linear time algorithm for finding a coloring for triangle-free planar graphs is given by Dvorak et al. in [12]. The algorithm can be summarized crudely as follows: the graph is converted into a planar embedding with facial cycles. These facial cycles (or low-degree vertices) are contracted if they satisfy certain requirements, after which they can be colored greedily. A facial cycle is an ordering of the vertices on a face, such that they form a cycle.

These facial cycles are called "multigrams" and are facial cycles of sizes 4, 5, and 6 plus the "monogram", which is a vertex with at most degree two. The paper proves that all triangle-free planar graphs have a 3-coloring and that there is always a safe multigram which can be selected during the reduction process.

The paper explains in detail how safety can be checked, and how each multigram can be reduced. Safety in this algorithm implies that reducing the multigram will not violate the triangle-freeness of the graph.

4.1.2 Deviation from algorithm & optimalization

The paper does not explicitly explain how to find the multigrams, and only instructs to use a planarity check algorithm that provides an embedding for the graph. Using this embedding we can efficiently find facial cycles.

The method to find multigrams that has been implemented is greedy. For each step to find a multigram, first is checked whether the graph contains a monogram. This is simply checking whether the graph contains a vertex with degree two or lower, and then returning that vertex as the multigram.

Only when no such vertices exist in the graph will we loop over faces to find a facial cycle of the correct size, after which safety is checked. If the multigram is not safe, the process is repeated until a safe multigram is found.

Because we try to randomly generate the triangle-free planar graphs, we hope to get as representative as possible graphs. However regardless during testing and benchmarking, some cases of the algorithm were not encountered. For example, we never encountered a safe pentagram.

The main result of the original paper is to provide a linear time algorithm, by among other things, doing a local lookup for new multigrams. A local lookup for multigrams, in this case, means keeping track of which vertices and edges have been removed or added during reductions and only using those vertices and edges to start the search for a safe multigram. The paper shows that a safe multigram must exist in the vertices and edges that have been "touched" during reduction. This greatly reduces the number of possible searches you have to do before finding a safe multigram. However, even though in theory this might be faster, it proved to be a lot of checks in practice which slows the search for a new multigram considerably. Therefore we use a greedy method in the implementation, even though this might be polynomial in practice, with a runtime of $O(n^2)$ where n is the number of vertices in a graph. This greedy method is to greedily get a monogram if it exists, and handle that before trying to look for a different multigram. The reason is that getting a monogram is as easy as getting the list of degree two or lower vertices and picking one. Also, we found that most of the time after reducing there still exist a lot of monograms, so most of the time this is a very quick operation. Only when there exist no monograms, do we use the slow search for a larger multigram, which as mentioned we can find by looping over facial cycles until we find a safe multigram.

4.2 (P7, C3)-free graphs

4.2.1 High-level overview

Bonomo-Braberman et al. describe a polynomial algorithm for solving the 3-coloring problem for (P_7, C_3)-free graphs that runs in $O(|V(G)|^5(|V(G)| + |E(G)|))$ [7]. Recall that (P_n, C_m)-free graphs are graphs that do not contain any induced paths of length n and induced cycles of size m. Since the graph is P_7 and C_3-free, the only odd cycles that can exist are a C_5, or a C_7, and the paper proves that one of these must exist. If the graph contains a C_7, the graph is 3-colorable as proven in the paper, so we assume there exists a C_5. The algorithm works by fixing a C_5 with an arbitrary coloring and then finding and extending that coloring by removing color options from the list-colorings of sets of neighbors of the C_5. These vertex sets are based on their connectivity with the C_5, for example, having exactly one neighbor in the C_5. In the left-over graph, list-coloring options can be removed by using

these sets of neighbors, for example by removing colors from the list-coloring that have already been used by the neighbors. The paper proves that when this process is done until all vertices have a color, or only 2 colors remain in their list. At this point, you can reduce the problem to 2-SAT, which is proven to be polynomial.

4.2.2 Deviation from algorithm & optimalization

Due to practical reasons, instead of implementing a 2-SAT solver or using a 2-SAT solver package, z3 is used. We expect this to be faster than most 2-SAT-specific solvers, even though z3 is not exclusively used for 2-SAT, because z3 is highly optimized.

Because the graphs for this graph class are not randomly generated, the graphs generated will not cover all cases in the algorithm. We will elaborate on this further in section 5.2. For example, we do not encounter any non-trivial connected components that need enumeration to color the connected components. Therefore these cases have not been implemented.

4.3 Locally Connected graphs

4.3.1 High-level overview

In the paper by Kochol ([18]), an algorithm is presented that solves the 3-coloring problem for locally connected graphs in $O(|V(G)|+|E(G)|)$. Locally connected graphs have the property that for all vertices the open neighborhood is connected. The open neighborhood of a vertex v is all the neighbors of v excluding v. The paper makes use of the fact that a 3-clique-ordering can be found for a locally-connected graph in $O(|E|)$. A 3-clique-ordering $(v_1, v_2, ..., v_n)$, $n = |V(G)|$ is an ordering of the vertices of a given graph G, where for all $3 < i < n$ the induced subgraph $G[(v_1, v_2, ..., v_i)]$ contains a 3-clique covering v_i. The algorithm constructs this 3-clique-ordering (the paper shows that this always exists for a locally connected graph) by fixing a 3-clique and extending the ordering with a vertex that can be quickly found using a bipartite graph with the vertices already ordered and the remaining vertices as partitions. In each iteration, a vertex v is ordered from the remaining vertices by checking that there exist two vertices that form a triangle with the unordered vertex v until all vertices are ordered.

Then using the ordering, the graph can be greedily colored by fixing the coloring of the 3 first vertices of the ordering and then coloring the next vertex in the ordering with the only remaining color. Then we either continue until all vertices have been colored, and return that coloring, or we encounter an uncolorable vertex and indicate that the graph is not 3-colorable.

4.3.2 Deviation from algorithm & optimalization

The paper does not mention this, but we have implemented a check during the 3-clique-ordering creation to check if the graph is 3-colorable up to the point of the partial ordering. If this is not the case, we can stop because the graph is already not 3-colorable, and if it is the case we continue creating the ordering. This does mean that per iteration more checks have to be done, but most locally connected graphs are not 3-colorable as all our larger generated instances are not 3-colorable, so this will result in a fast termination of the algorithm.

Chapter 5

Experimental Setup

5.1 Experiments

For each graph instance mentioned in the next section (5.2) the algorithm for that graph class is benchmarked, measuring the amount of time each algorithm takes to complete solving, together with all generic algorithms. For the DSATUR and largest first algorithms, not only time but also the number of colors the algorithm uses are measured. For the planar triangle-free and locally connected graphs, we will also check how consistent the algorithms run across different instances of the same size.

5.2 Graph Instances

The following tables 5.1 have laid out the graphs that will be used to benchmark the algorithms. We will be benchmarking the algorithms for planar triangle-free, locally connected, and (P_7, C_3)-free graphs, and thus we have generated graphs for those graph classes. We can see the maximum amount of vertices generated for planar triangle-free graphs is only 10000 instead of 50000 for the locally connected, and (P_7, C_3)-free graph classes. This is because the way graphs are generated means that generating graphs with more than 10000 vertices will take an infeasible amount of time, and thus has been cut off at 10000 vertices.

During testing, we discovered that all generated locally connected graphs are not 3-colorable. A graph was generated specifically to show that the algorithm for locally connected can produce 3-colorings, by decreasing the density and amount of vertices. This graph can be seen in figure 5.1. However, we were unable to generate a random graph with more than 100 vertices that was 3-colorable. This implies that the algorithm explained in section 4.3 can quickly assert that the graph is not 3-colorable, which greatly improves the runtime of the algorithm. For example, for a 25000 vertex locally connected graph it takes around 39 minutes to compute the entire 3-clique-ordering. But the algorithm can already conclude that the graph is not

3-colorable in 0.8 seconds, which is a 2925000% improvement.

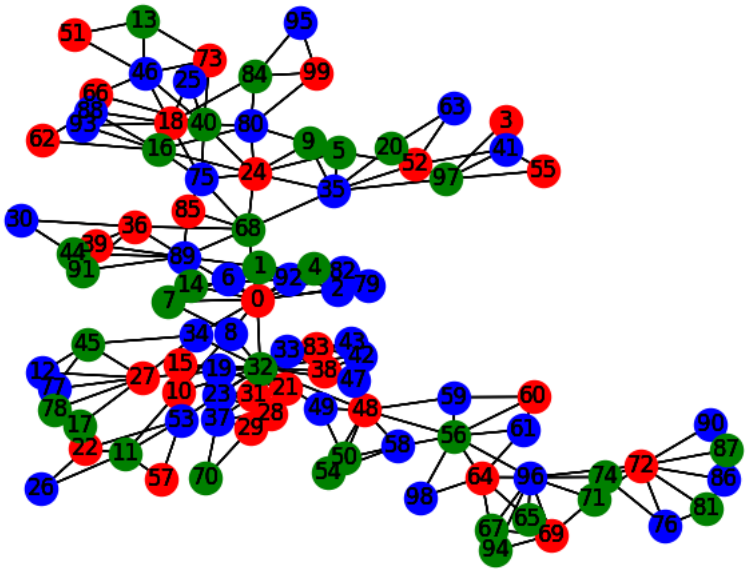


Figure 5.1: A locally connected graph that is 3-colorable, 100 vertices with density 0.04

The (P_7, C_3)-free graphs have not been generated randomly. This is because we observed that the graphs that were being generated were disconnected, and the connected components had a limit to the number of vertices. This meant that in reality, the graphs the algorithm would run on are very small, and not representable for the graph class. Therefore we had to create non-random graph instances that are of proper size and are more representable. Eventually, we created graphs that look like figure 5.2. There we are using six partitions U, V, W, X, Y, Z, where we can get induced paths of size 6, and get induced cycles of size 5. The partitions (U, V), (V, W), (X, Y), (Y, Z) are fully-connected bipartite graphs, with one edge between an arbitrary vertex from U and X, and one edge between W and Y. This creates a (P_7, C_3)-free graph of arbitrary size because all paths of size 7 must have a smaller cycle in the path, and therefore the path is not an induced path. And clearly, there are no triangles in the graph due to each combination of partitions being bipartite. Because this graph has a relatively high density, we expect this type of graph to not be a best-case scenario, and thus still allows us to check how the special-case algorithm will run in comparison to the generic algorithms.

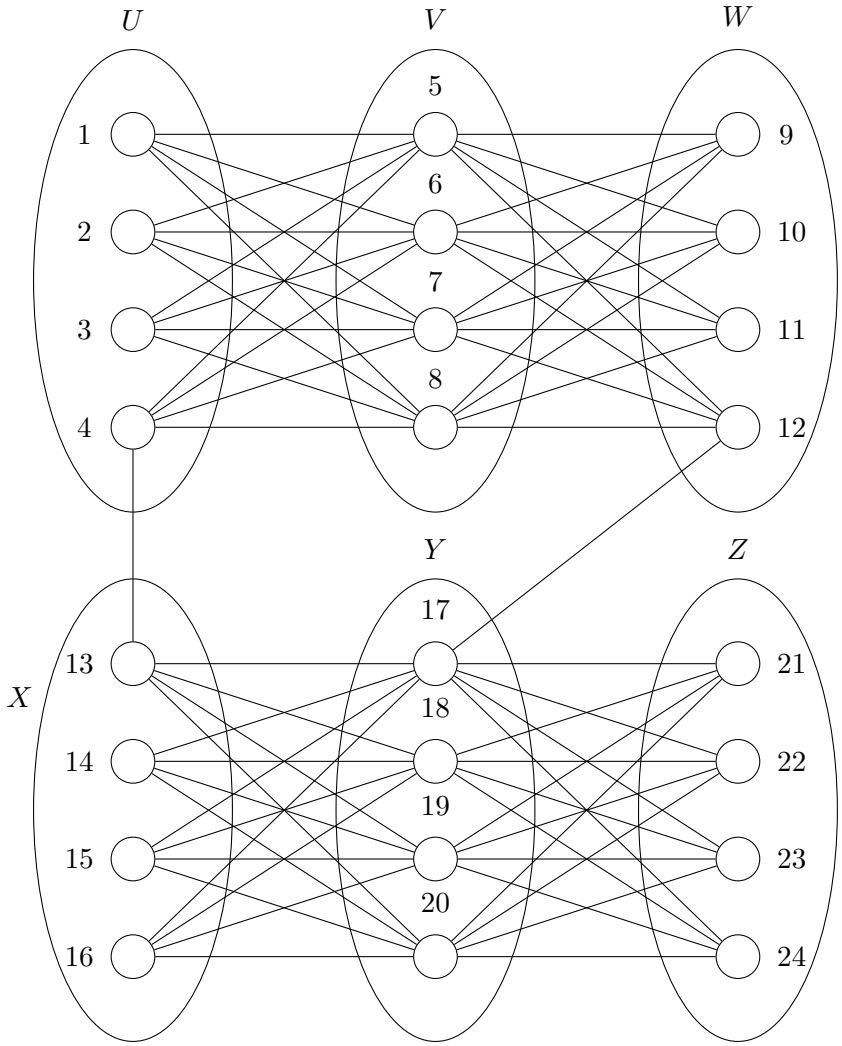


Figure 5.2: An example (P_7, C_3)-free graph with 24 vertices

Vertices	Density
100	0.030
500	0.006
1000	0.003
2000	0.00159
3000	0.00105
4000	0.00079
5000	0.00063
6000	0.00054
7000	0.00045
8000	0.00039
9000	0.00035
10000	0.00032

(a) Planar triangle-free graph instances

Vertices	Density
100	0.0549
500	0.0486
1000	0.0415
2000	0.0347
3000	0.0319
4000	0.0308
5000	0.0303
7500	0.0300
10000	0.0300
15000	0.0300
20000	0.0300
25000	0.0300
30000	0.0300
35000	0.0300
40000	0.0300
45000	0.0300
50000	0.0300

(b) Locally connected graph instances

Vertices	Density
100	0.227
500	0.222
1000	0.223
2000	0.222
3000	0.222
4000	0.222
5000	0.222
7500	0.222
10000	0.222
15000	0.222
20000	0.222
25000	0.222
30000	0.222
35000	0.222
40000	0.222
45000	0.222
50000	0.222

(c) (P_7, C_3)-free graph instances

Table 5.1: The number of vertices and the corresponding density for the time benchmarks

5.3 Hardware

All experiments are run on a Microsoft Azure virtual machine (instance type E16-8as v5), with 8 cores and 128 GB ram.

5.4 Software

All code has been written using Python 3.11, including graph generation. Some packages such as z3 use a C API, which uses a Python wrapper to interface with the API. All code can be found on GitHub here: `https://github.com/EmielSchmeink/graph-3-coloring`.

The most essential package for this project is NetworkX: `https://networkx.org/`. This package has been used for all graph representations and operations during the project because it has a large variety of different operations built-in, and is easy to use.

Chapter 6

Results

In this chapter, as explained in chapter 5 we will lay out the results of our experiments. We will first show the consistency of our algorithms across different instances of a certain size. And afterward, we will discuss the benchmarks of the generic algorithms versus the non-generic algorithms.

6.1 Consistency

Here we will discuss the consistency of the locally connected and planar triangle-free algorithms across different instances of similar size, with the results of the execution time of different instances shown in table 6.1. Consistency results for the (P_7, C_3)-free is not relevant, since the way we generate graphs means that for a certain number of vertices, the graphs will be identical across instances.

The mean density for the planar graphs is 0.000631, and the mean execution time is 2.34506, with standard deviations of 0.0000033 and 0.034004 respectively. For the locally connected graphs, the results are 0.029996 and 0.83984 as means, with 0.000011 and 0.096853 as standard deviations for the density and execution time respectively. As we can see in table 6.1 the planar triangle-free graphs are very close together with regard to the density of the graph. The execution times are also very similar, indicating that the algorithm runs consistently across different graph instances, and by extension on the planar triangle-free graph class.

Similar results are shown for the locally connected graph instances, with two outliers in execution time, instances 5 and 10. This has to do with the fact that the locally connected algorithm has to do 3 and 2 iterations respectively before the graph cannot be colored any further, whereas the other instances can be concluded after only 1 iteration. Recall that the algorithm fixes a triangle, and extends this triangle with a vertex in each iteration, so the algorithm concludes that the graph is not 3-colorable after processing 6 and 5 vertices respectively.

#	Vertices	density	Execution time
1	5000	0.00063	2.28460
2	5000	0.00063	2.33527
3	5000	0.00063	2.30630
4	5000	0.00063	2.35659
5	5000	0.00063	2.39502
6	5000	0.00063	2.30321
7	5000	0.00064	2.37897
8	5000	0.00063	2.32644
9	5000	0.00063	2.38087
10	5000	0.00063	2.32286

(a) Planar triangle-free graph instances, with the first column indicating the instance number

#	Vertices	density	Execution time
1	25000	0.03000	0.81895
2	25000	0.02998	0.83939
3	25000	0.03000	0.82499
4	25000	0.02998	0.77809
5	25000	0.02999	1.04916
6	25000	0.02999	0.77400
7	25000	0.03000	0.78091
8	25000	0.03000	0.77405
9	25000	0.03001	0.78725
10	25000	0.03001	0.95072

(b) Locally connected graph instances, with the first column indicating the instance number

Table 6.1: Consistency results

6.2 Benchmarks

In this section, we will be looking at, and discussing the results of the experiments outlined in the previous chapter. So we will look at the resulting execution times running the algorithms, and explain why the results are the way they are.

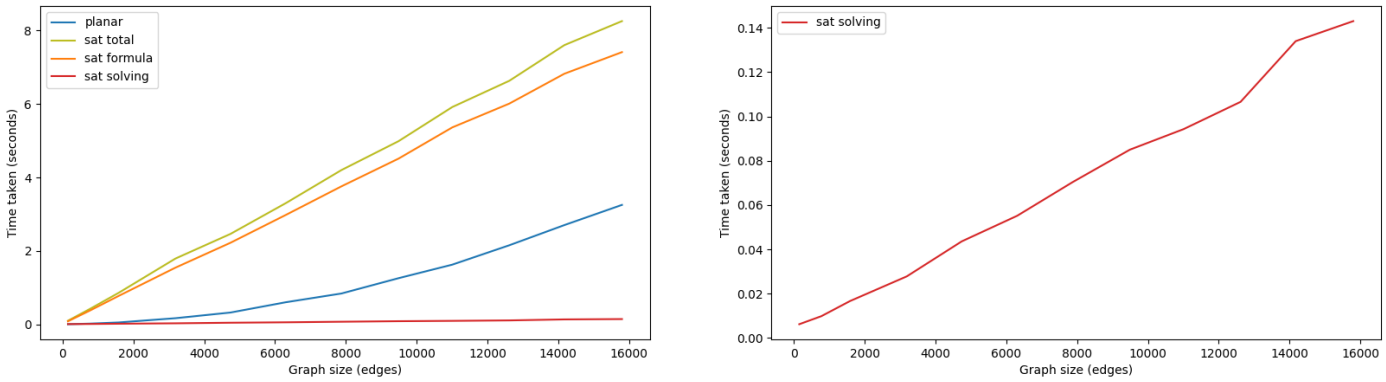


Figure 6.1: Planar triangle-free benchmarks

Planar triangle-free results In figure 6.1 we can see something that will be recurring is that the creation of the formula takes much more time than actually solving the formula, where the 'SAT total' line is the time for formula creation plus formula solving. This is because of the limitations of Python. The Python API of z3 is interfacing with the C implementation of z3, and we expect that the way that Python creates clauses is very inefficient. Similarly Python overhead is high for both space and performance with lists to store the clauses, as we will discuss soon.

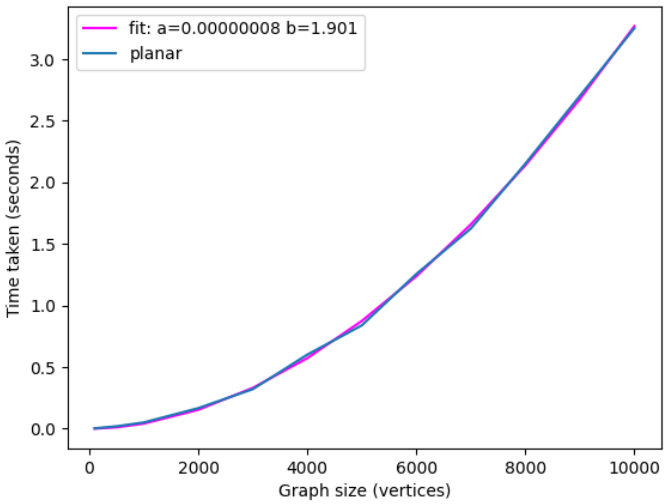


Figure 6.2: Plot of the planar benchmark results and a fit on the data in the form $y = ax^b$

The planar algorithm seems to run sub-quadratically as can be seen in

fig 6.2, with a regression fit showing a $O(n^{1.9})$ function, with n the number of edges. While the solving of the formula seems to run linearly for the number of edges. The polynomial running time of the planar algorithm is to be expected as explained in 4.1.2. Since the planar algorithm runs in polynomial time, and the total for SAT seems to run in linear time, we expect at some point for a larger instance that the time taken for the planar algorithm will surpass that of the SAT total.

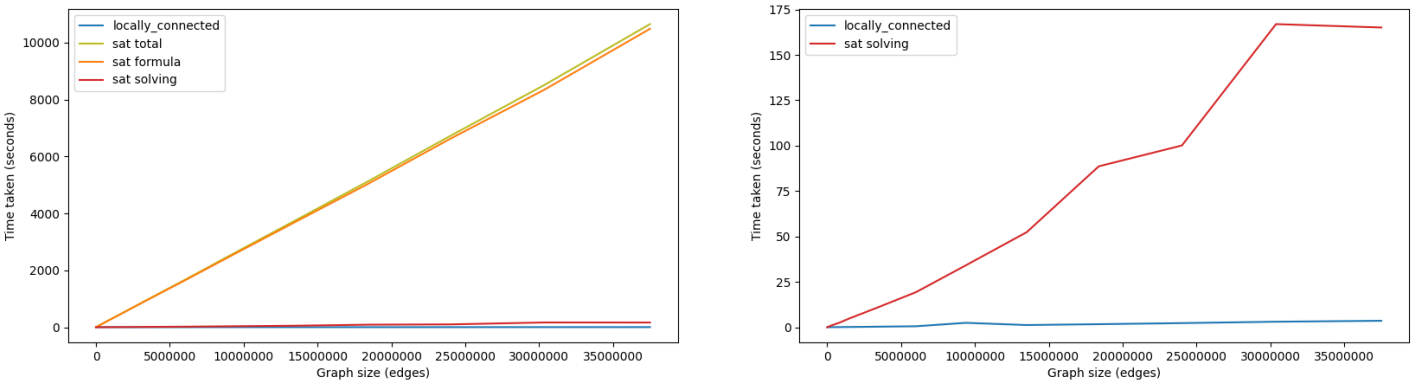


Figure 6.3: Locally connected benchmarks

Locally connected results Again in figure 6.3 we see that the majority of time taken for SAT is creating the formula, and we can see that the locally connected algorithm can very quickly conclude that the graph is not 3-colorable as explained in 5.2. Interestingly we again see an approximately linear runtime of SAT solving, while the linear formula creation is to be expected. The linear runtime of SAT solving is most likely attributable to the fact that z3 can, similarly to the locally connected algorithm, relatively quickly conclude that the graph is impossible to color. The reason for this is that the locally connected graphs are inherently dense, and therefore the likelihood of a graph being 3-colorable is low. Therefore we also saw in section 5.2 that even at 100 vertices the graph is unlikely to be 3-colorable.

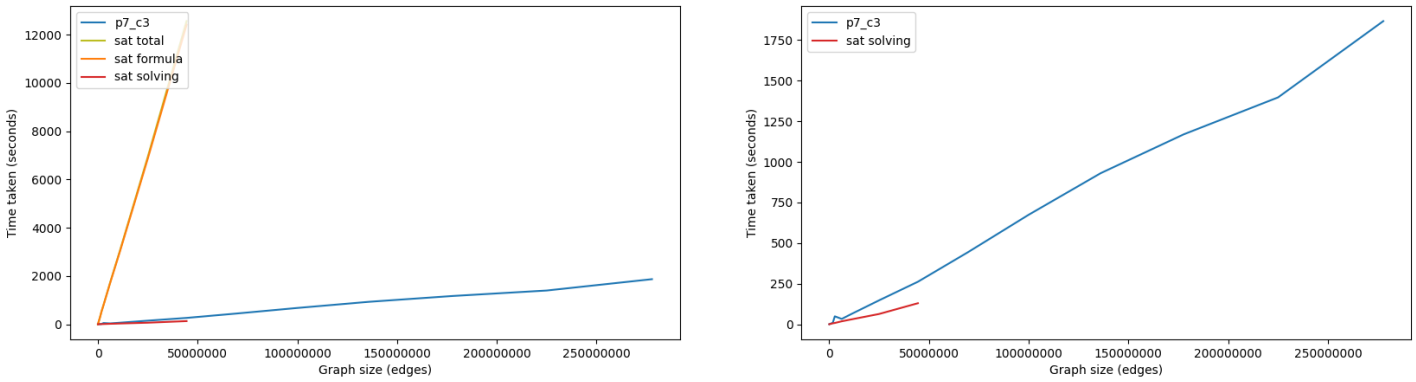


Figure 6.4: (P_7, C_3)-free benchmarks

(P_7, C_3)-free results Here we again see in figure 6.4 that formula creation is a slow process, and again linear (as to be expected). However, we can see that after around 20000 vertices and 45 000 000 edges, the SAT solver cannot complete solving anymore. This has to do with the fact that the server the algorithms were run on has 128 GB of ram, but the amount of memory needed to store the formula is more than 128 GB.

Regardless, the formula solving seems to run linearly, because it probably can quickly find a 3-coloring for the graph. The same is true for the (P_7, C_3)-free algorithm, even though the runtime is polynomial, it does not need to execute parts of the algorithm that worsen the runtime because of the limitations of the graph instances generated.

6.3 General 3-coloring

Vertices	Edges	Exec. time
100	151	0.05053
500	777	0.34443
1000	1586	0.96131
2000	3188	1594.61272
3000	4742	6.85939
5000	7893	35.46125

(a) Planar general 3-coloring algorithm results

Vertices	Edges	Exec. time
100	272	19.17467
500	6069	4.30593
1000	20711	31.09981
2000	69437	339.83842
3000	143561	2840.83127

(b) Locally connected general 3-coloring algorithm results

Vertices	Edges	Exec. time
100	1124	0.50199
500	27724	21.69876
1000	111224	155.16119
2000	444224	1194.13631

(c) (P_7, C_3)-free general 3-coloring algorithm results

Table 6.2: General 3-coloring results

We will be discussing the results of the generic 3-coloring algorithm outlined in section 3.2.1. The results can be seen in table 6.2, with a timeout of 3600 seconds due to algorithms possibly running for extremely long times.

Recall that a bushy forest is a degree four or higher forest covering a large portion of the graph. More extensively, the planar graphs' bushy forest will be relatively deep compared to the locally connected and (P_7, C_3)-free bushy forests because the latter two are much more connected, and thus fewer layers are needed before the maximal amount of vertices is covered. Thus the differences between the planar instances and the other instances are as follows: for the planar graphs, because they are much deeper, there are many more permutations of colors possible for the bushy forest, which will take a large portion of the time of the algorithm because the creation is relatively quick. And for the other two graph classes, the creation of the bushy forest itself is the more time-consuming part of the algorithm because it has to do a lot of checks per layer which scale with the number of neighbors of vertices. But because the bushy forests are quite shallow, the greedy coloring is done relatively quickly. These characteristics explain the differences in the algorithm's execution times for the different graph classes.

As explained in section 3.2.2, we expected the search through all combinations of colorings of the vertices of the bushy forest to be very slow, which we can see artifacts of here. For the planar triangle-free instance of 2000 vertices, it took the algorithm 1660 times as long as 1000 vertices, and

then the algorithm is again much faster for 3000 vertices. This has to do with the fact that the algorithm gets "unlucky" with 2000 vertices and that a lot of the colorings it finds for the bushy forest do not result in a valid 3-coloring for the graph. More concretely, if a vertex is colored near the root of a bushy forest that results in an uncolorable graph, it means that the recursive coloring of the bushy forest has to check many combinations of colorings for the bushy forest vertices, traversing up the tree, before it finds a coloring that results in a valid 3-coloring for the graph.

6.4 Non-exact results

Vertices	Planar exec. time	LC exec. time	(P_7, C_3)-free exec. time	Planar #c	LC #c	(P_7, C_3) #c
100	0.00828	0.00634	0.01034	3	4	3
500	0.13177	0.00773	0.79598	3	10	3
1000	0.53873	1.86401	5.84751	4	14	3
2000	2.34202	1.81966	47.31270	3	20	3
3000	5.18713	12.49810	161.90049	3	25	3
4000	9.54564	40.83564	518.60755	4	30	3
5000	15.14896	99.09168	882.21062	3	35	3
6000	21.96727	x	x	4	x	x
7000	29.98850	x	x	3	x	x
7500	x	584.76115	2853.20061	x	48	3
8000	39.37902	x	x	4	x	x
9000	50.76602	x	x	3	x	x
10000	63.63940	1669.25009	7506.70672	4	61	3
15000	x	5765.56430	24562.48165	x	85	3
20000	x	17125.61363	62150.74616	x	108	3

Table 6.3: DSATUR results, #c indicating the number of colors used

For the DSATUR algorithm [8], we can see in table 6.3 that we only benchmarked up to graphs of 20000, because after this size the time of execution became unfeasible, and it gives us a good enough representation of the algorithm's performance.

As we can see in table 6.3 and 6.4 DSATUR achieved a coloring using significantly fewer colors, but also taking significantly more time. For example, the 20000 vertex instance takes around 1800 times as long for DSATUR as the largest first algorithm, while using around half of the colors to color the graph.

In general, the largest first algorithm (greedily picking the highest degree vertex) is very quick and can color all graph instances in a reasonable time,

Vertices	Planar exec. time	LC exec. time	(P_7, C_3)-free exec. time	Planar #c	LC #c	(P_7, C_3) #c
100	0.00057	0.00073	0.00061	4	5	3
500	0.00138	0.00202	0.00434	3	11	3
1000	0.00166	0.00423	0.01004	4	17	3
2000	0.00314	0.01191	0.03716	4	23	3
3000	0.00450	0.02402	0.08282	4	28	3
4000	0.00605	0.04134	0.14831	4	34	3
5000	0.00746	0.06827	0.24527	4	40	3
6000	0.00883	x	x	4	x	x
7000	0.01053	x	x	4	x	x
7500	x	0.15006	0.57881	x	54	3
8000	0.01190	x	x	4	x	x
9000	0.01367	x	x	4	x	x
10000	0.01528	0.25662	1.11589	4	67	3
15000	x	0.59033	2.83627	x	92	3
20000	x	1.08807	5.16350	x	117	3
25000	x	1.77796	8.33693	x	139	3
30000	x	2.65412	12.18510	x	161	3
35000	x	3.81791	15.51984	x	183	3
40000	x	5.15247	19.73339	x	205	3
45000	x	6.81059	22.90066	x	226	3
50000	x	8.56991	34.35187	x	246	3

Table 6.4: Largest first results, #c indicating the number of colors used

which is to be expected. But the number of colors used for the coloring is quite large, and if a small coloring is needed it might not satisfy the requirements.

Chapter 7

Conclusion

Algorithm conclusion In chapter 1 we proposed the question "Does theory match practice?", in the sense that in theory, the special case algorithms should be faster than the generic algorithms. Intuitively one would expect this to be the case for large instances since the exponential growth of generic algorithms should surpass the sub-exponential non-generic algorithms at some point. However, this is not what we saw during experimenting.

During experimentation, we saw that solving the SAT formula is faster than running the non-generic algorithms, even for very large instances, except for locally connected graphs. This discrepancy arises because we could not generate large 3-colorable locally connected instances for testing, and the full execution of the locally connected algorithm was unnecessary, leading to faster execution than when the full algorithm would execute. The creation of the SAT formula is however very slow, but we will explain later how this could be optimized. These results are in contrast to the theory, where SAT solving should follow exponential growth.

In general, the best algorithm in all cases is the conversion to SAT. The z3 solver is a culmination of years of research and development and is hyper-efficient in solving even very large graphs en equivalently very large SAT formulas. The actual conversion itself was by far the bottleneck of the algorithm, and this could easily be optimized by doing the conversion in a more efficient language such as Rust or C++ since the actual implementation of this is quite simple. The runtime of the SAT solver is linear for all graph classes, up to around 45 000 000 edges. We expect that for efficient graph classes (such as the ones discussed in this thesis) the solver will always run in linear time in practice.

The conversion to SAT also has the advantage that it is general for all graphs, and is easy to implement. The special case algorithms are a lot of work to implement and are very prone to bugs and errors during the implementation because the special case algorithms are highly involved, while SAT is a straightforward conversion. This also might cause unforeseen errors later down the line, for a specific graph with a certain edge case.

For example, a bug was introduced to the Locally Connected algorithm implementation where an optimization was added by switching the lists to sets in Python for better lookup performance, but the clique ordering list was also switched to a set, sometimes resulting in incorrect orderings. This presents the complexity of the special case algorithms, and how easily they can contain bugs.

Non-exact algorithms gave mixed results. DSATUR performed very poorly in its execution time but performed better in the number of colors used for coloring. Largest first had a big improvement in execution time compared to DSATUR but used more colors, so if few colors are a requirement, it might be better to use DSATUR regardless of execution time.

The General 3-coloring algorithm might work in an absolute worst-case scenario, but it has been shown to perform very poorly in practice. This discrepancy arises from the fact that the theoretical worst-case scenario is rarely encountered in practice. As a result, the algorithm performs additional steps that provide little improvement in performance.

Limitations Python as a programming language has many advantages. It's easy to use, intuitive, has many built-in features, and has a huge variety of packages available. However, its main drawback is also a result of all of its advantages, it's slow. The interpreter is much slower than for example compiling and running a C++ program. Python has allowed us to be much faster in the implementations of graph generation and algorithms, but they may not be as fast or as optimal as perhaps a C++ implementation would be. On the flip side, if this project would have been done in C++, the amount of work produced would be much less.

One of the other issues is that the actual execution times of the algorithms may not be very helpful results for concluding which algorithm is faster because of graph limitations. Therefore we have to look at running times and how it seems to behave (linearly, polynomially, exponentially, etc.) with increasing graph sizes. And then we might encounter the problem that the trend seems to be linear but we might, at a larger instance, see a polynomial trend. However, the graph instances that we have used are of such sufficient size that we expect them to be large enough to show any trends in the running times of the algorithms.

Future work The generation of graphs is a time-consuming effort, and especially for larger graph instances, it quickly becomes infeasible to generate them within a reasonable time. It might be interesting to look at more efficiently generating large graph instances with requirements. Also, another interesting subject could be diving into if there are reduction rules, or kernelizations that z3 or other SAT solvers use that decrease the runtime of the SAT solvers to polynomial in theory for certain graph classes, not just practice. Maybe because the same structures that are in the graph classes that allow the 3-coloring problem to be solved more easily, also get converted into structures in the SAT formulas that make the formula more easily solvable. Finally, an obvious candidate for further research is other graph classes that have been proven to run in polynomial time.

Final conclusion "Theory does not seem to match practice." By this we mean that the special case algorithms are not faster than conversion to SAT, and by extension thus not faster than the generic algorithms. If you have a fast implementation to convert a graph into a SAT formula, in both space and time, solving the 3-coloring problem using a SAT solver is the most efficient algorithm to do so. Special case algorithms seem to be less efficient, and more error-prone.

Bibliography

[1] Faisal N Abu-Khzam and Michael A Langston. Almost Exact Graph 3-Coloring in O(1.277ˆn) Time. In *Cologne Twente Workshop on Graphs and Combinatorial Optimization*, 2012.

[2] Noga Alon and Nabil Kahale. A Spectral Technique for Coloring Random 3-Colorable Graphs. *https://doi.org/10.1137/S0097539794270248*, 26(6):1733–1748, 7 2006. ISSN 00975397. doi: 10.1137/S0097539794270248. URL `https://epubs.siam.org/doi/10.1137/S0097539794270248`.

[3] R Beigel and D Eppstein. 3-coloring in time 0(1.3446/sup n/): a no-MIS algorithm. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 444–452, 1995. doi: 10.1109/SFCS.1995.492575.

[4] Richard Beigel and David Eppstein. 3-coloring in time O(1.3289ˆn). *Journal of Algorithms*, 54(2):168–204, 2 2005. ISSN 01966774. doi: 10.1016/j.jalgor.2004.06.008.

[5] Avrim Blum. An O (n 0.4)-approximation algorithm for 3-coloring. *ACM*, pages 535–542, 1989. doi: 10.1145/73007.73058. URL `https://dl.acm.org/doi/10.1145/73007.73058`.

[6] Avrim Blum and David Karger. An O(n314)-coloring algorithm for 3-colorable graphs. *Information Processing Letters*, 61(1):49–53, 1 1997. ISSN 0020-0190. doi: 10.1016/S0020-0190(96)00190-1.

[7] Flavia Bonomo-Braberman, Maria Chudnovsky, Jan Goedgebeur, Peter Maceli, Oliver Schaudt, Maya Stein, and Mingxian Zhong. Better 3-coloring algorithms: Excluding a triangle and a seven vertex path. *Theoretical Computer Science*, 850:98–115, 1 2021. ISSN 0304-3975. doi: 10.1016/J.TCS.2020.10.032.

[8] Daniel Brélaz. New Methods to Color the Vertices of a Graph. *Commun. ACM*, 22(4):251–256, 4 1979. ISSN 0001-0782. doi: 10.1145/359094.359101. URL `https://doi.org/10.1145/359094.359101`.

[9] Dario Cavallaro and Till Fluschnik. 3-Coloring on Regular, Planar, and Ordered Hamiltonian Graphs. *ArXiV*, 4 2021. doi: 10.48550/arxiv.2104.08470. URL https://arxiv.org/abs/2104.08470v1.

[10] H. De Fraysseix and P. Rosenstiehl. A Depth-First-Search Characterization of Planarity. *North-Holland Mathematics Studies*, 62(C):75–80, 1 1982. ISSN 0304-0208. doi: 10.1016/S0304-0208(08)73550-3.

[11] Michał Dębski, Marta Piecyk, and Paweł Rzążewski. Faster 3-coloring of small-diameter graphs. *ArXiV*, 4 2021. doi: 10.48550/arxiv.2104.13860. URL https://arxiv.org/abs/2104.13860v1.

[12] Zdenek Dvorak, Ken-ichi Kawarabayashi, and Robin Thomas. Three-coloring triangle-free planar graphs in linear time. *ACM Transactions on Algorithms 7 (2011), Article 41*, 2 2013. doi: 10.48550/arxiv.1302.5121. URL https://arxiv.org/abs/1302.5121v1.

[13] Zdeněk Dvořák, Daniel Král', and Robin Thomas. Coloring triangle-free graphs on surfaces. In *Proceedings of the 2009 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 120–129. ACM, 2009. doi: 10.1137/1.9781611973068.14. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611973068.14.

[14] Hacène Aït Haddadène and Frédéric Maffray. Coloring perfect degenerate graphs. *Discrete Mathematics*, 163(1-3):211–215, 1 1997. ISSN 0012-365X. doi: 10.1016/S0012-365X(96)00009-X.

[15] Vít Jelínek, Tereza Klimošová, Tomáš Masařík, Jana Novotná, and Aneta Pokorná. On 3-Coloring of (2 P4, C5)-Free Graphs. *Algorithmica*, 84(6):1526–1547, 6 2022. ISSN 14320541. doi: 10.1007/S00453-022-00937-9/FIGURES/14. URL https://link.springer.com/article/10.1007/s00453-022-00937-9.

[16] T. Karthick, Frédéric Maffray, and Lucas Pastor. Polynomial Cases for the Vertex Coloring Problem. *Algorithmica*, 81(3):1053–1074, 3 2019. ISSN 14320541. doi: 10.1007/S00453-018-0457-Y/FIGURES/2. URL https://link.springer.com/article/10.1007/s00453-018-0457-y.

[17] Ken-Ichi Kawarabayashi and Mikkel Thorup. Coloring 3-Colorable Graphs with Less than N1/5 Colors. *J. ACM*, 64(1), 3 2017. ISSN 0004-5411. doi: 10.1145/3001582. URL https://doi.org/10.1145/3001582.

[18] Martin Kochol. 3-coloring and 3-clique-ordering of locally connected graphs. *Journal of Algorithms*, 54(1):122–125, 1 2005. ISSN 0196-6774. doi: 10.1016/J.JALGOR.2004.05.003.

[19] Rustam Latypov and Jara Uitto. Coloring Trees in Massively Parallel Computation. *ArXiV*, 5 2021. doi: 10.48550/arxiv.2105.13980. URL https://arxiv.org/abs/2105.13980v2.

[20] Zepeng Li and Jin Xu. Constructions of Uniquely 3-Colorable Graphs. In *2016 IEEE First International Conference on Data Science in Cyberspace (DSC)*, pages 143–148, 1 2016. doi: 10.1109/DSC.2016.40.

[21] Enrico Malaguti and Paolo Toth. A survey on vertex coloring problems. *International Transactions in Operational Research*, 17 (1):1–34, 2010. doi: https://doi.org/10.1111/j.1475-3995.2009.00696.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1475-3995.2009.00696.x.

[22] D. S. Malyshev. Polynomial-time approximation algorithms for the coloring problem in some cases. *Journal of Combinatorial Optimization*, 33 (3):809–813, 4 2017. ISSN 15732886. doi: 10.1007/S10878-016-0008-X/FIGURES/1. URL https://link.springer.com/article/10.1007/s10878-016-0008-x.

[23] Microsoft Research. Z3Prover/z3: The Z3 Theorem Prover, 2012. URL https://github.com/Z3Prover/z3.

[24] N. S. Narayanaswamy and C. R. Subramanian. Dominating set based exact algorithms for 3-coloring. *Information Processing Letters*, 111(6): 251–255, 2 2011. ISSN 0020-0190. doi: 10.1016/J.IPL.2010.11.009.

[25] Thomas J. Sager and Shi Jen Lin. A Pruning Procedure for Exact Graph Coloring. *https://doi.org/10.1287/ijoc.3.3.226*, 3(3):226–230, 8 1991. ISSN 08991499. doi: 10.1287/IJOC.3.3.226. URL https://pubsonline.informs.org/doi/abs/10.1287/ijoc.3.3.226.

[26] Fomin Fedor V and Sergeand Saurabh Saket Gaspers. Improved Exact Algorithms for Counting 3- and 4-Colorings. In Guohui Lin, editor, *Computing and Combinatorics*, pages 65–74, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73545-8.

[27] Wenan Zang. Coloring graphs with no odd-K4. *Discrete Mathematics*, 184(1-3):205–212, 4 1998. ISSN 0012-365X. doi: 10.1016/S0012-365X(97)00090-3.

[28] Enqiang Zhu, Pu Wu, and Zehui Shao. Exact algorithms for counting 3-colorings of graphs. *Discrete Applied Mathematics*, 322:74–93, 12 2022. ISSN 0166-218X. doi: 10.1016/J.DAM.2022.08.002.