Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 28, no. 1, pp. 313–349 (2024)
DOI: 10.7155/jgaa.v28i1.2956

Experimental Analysis of Algorithms for the Dynamic Graph Coloring Problem

Menno Theunis *Marcel Roeloffzen*

Eindhoven University of Technology, Eindhoven, Netherlands

Submitted: February 2024 Accepted: July 2024 Published: August 2024

Article type: Regular paper Communicated by: H. Meyerhenke

Abstract. This paper focuses on the dynamic graph coloring problem, a dynamic variant based on the well-researched graph coloring problem. This variant of the problem not only considers the number of colors used in the coloring for a graph, but also how many nodes in this graph need to change their color when the graph is changed. The balance between these two measures of quality, as well as running time, creates an inherent trade-off, in which algorithms solving this problem often only focus on one or the other. A variety of such algorithms already exist and are compared, as well as improved upon, in this paper. Each of these algorithms uses different variables to measure its effectiveness, making it difficult to compare their advantages and disadvantages. Finding the right option for a practical application is thus unnecessarily difficult. By implementing the different algorithms and comparing them experimentally, we get a better insight of the strong and weak points of these algorithms. Using this knowledge we propose two new improved variants of these algorithms, obtained by combining aspects of the existing ones. We find that this approach of combining existing algorithms with different strong points often yields superior results and allows for a more versatile trade-off within the algorithm, making it suitable for a broader range of practical applications.

E-mail addresses: m.theunis.tue@protonmail.com (Menno Theunis) m.j.m.roeloffzen@tue.nl (Marcel Roeloffzen)

1 Introduction

1.1 Background and motivation

The graph coloring problem is one of the most well known problems in computer science and combinatorics. The aim is to find a way of coloring some graph G such that no two adjacent elements have the same color. Many variations of this problem exist, most of which are well studied. In this paper we focus on problems within the vertex coloring variant. Regardless of the variation, one aspect generally remains the same: the aim is to find a valid coloring that uses few colors. To this extent, various goals exist, ranging from finding the chromatic number of a graph (i.e. the minimum number of colors required) to finding out whether a graph can be colored with a predefined number of colors.

Apart from being a set of interesting theoretical problems, the graph coloring problem can also be used to solve real-world problems by modeling a situation using a graph and letting a coloring encode a solution. One example of such a model would be the frequencies of routers with overlapping ranges. Each node in the graph would represent a router, each edge showing that two routers have an overlap in range and each color would represent a specific frequency the routers can be set to. A valid coloring with few colors then represents a way of assigning the routers frequencies such that there is no interference between routers and only a few frequency channels are required.

Some problems also exist which can be modeled using the graph coloring problem, but for which a single static coloring is not a sufficient solution. One such example can be obtained by replacing the routers in the previous real-world problem with mobile phones trying to communicate with the nearest cell phone tower. Phones that are close to the same tower cannot use the same frequency. A single static coloring would be insufficient, however, since mobile phones are able to move around, into or out of range of certain towers. Apart from that, new phone calls may be initiated or ongoing ones may be ended. These changes in the real-world situation can be represented in a graph by adding or removing edges or nodes. Because the graph changes over time, with each update the coloring must also be adjusted in order to stay valid and use as few colors as possible. The variation of the graph coloring problem that deals with this specific situation is called the dynamic graph coloring problem, which will be the focus of this paper.

Aside from the number of colors used in the solution to a dynamic graph coloring problem, there is an additional measure of quality: the number of recolors done. Every time the graph changes, the previous coloring must be adjusted or recomputed in order to fit the new graph. Every node that changes its color when comparing the old and new coloring is counted as a recolor. Such a recolor always represents a change in the real-world counterpart of the model. In the case of the mobile phones a recolor represents a phone switching to a different frequency or tower. Such a change may also have a cost associated with it, either based on money, time, or other resources. An inherent trade-off thus arises between the number of colors used in a coloring and the numbers of recolors required to get to the new coloring. Our goal is to investigate this trade-off, both by comparing existing algorithms for dynamic coloring as well as developing a new algorithm.

1.2 Related work

Many research papers have been dedicated to the graph coloring problem, proving various aspects of this problem, and finding solutions for it. It has already been shown that finding the chromatic number of a graph and finding out whether it can be colored with $K > 3$ colors are both NP-hard problems [1, 2]. Despite this result, many practical algorithms have already been found for the static graph coloring problem that allow the coloring of graphs in various ways. Some of these solutions limit themselves to planar graphs or small inputs [3]. Other times these approaches use heuristics that oftentimes result in using fairly few colors, rather than finding an exact solution. Many heuristics depend on first ordering the vertices and then coloring them one by one in the selected order. It has been proven that there must exist at least one such ordering of vertices for every graph that results in an optimal coloring, but finding this ordering is still an NP-hard problem [4]. The first and most well known such ordering is based on the degree of the vertices and described in [5].

Unlike many of the other variations, not much research has been performed on the dynamic graph coloring problem. Some heuristic algorithms do exist that attempt to solve this problem, such as those by Bhattacharya et al. [6], Barba et al. [7], Solomon et al. [8] and Yuan et al. [9] These results mainly focus on provable bounds on the number of recolors or colors used, but their performance in practical settings is not evaluated. A paper by Bossek et al. [10] does contain a comparison of algorithms for the dynamic graph coloring problem, but focuses on theoretical bounds and bipartite graphs, rather than the general case. Apart from this lack of experimental data, each paper describing a solution to the dynamic graph coloring problem uses its own terminology when it comes to variable names, making it difficult to compare the results of each paper. In this work we summarize the most popular approaches to solve the dynamic graph coloring problem and compare their effectiveness and trade-offs by running multiple experiments. We also propose two competitive new algorithms created by combining the existing approaches.

1.3 Contributions and organization

The main contributions this work provides are a clear comparison of the existing algorithms solving the dynamic graph coloring problem and two new algorithms that combine ideas from the existing ones. The general comparison of the algorithms aims at stimulating further research into different algorithms and into when each variation should be used. Currently it is difficult to find the correct algorithm to use for a project, since the papers introducing them do not clearly state the advantages and disadvantages as compared to the other available algorithms. In comparing these algorithms and running the experiments, the ideas for two new algorithms presented themselves, formed by combining some of the investigated algorithms together. These algorithms turn out to be quite competitive and provide a trade-off for some of the considered algorithms that do not normally allow for a parameter to control the importance of the number of colors versus the number of recolors. The implementations of these new algorithms, as well as any other files used during the research for this paper, can be found in the accompanying GitHub repository [11]. This work separates itself from the existing research because of the wide spectrum of generated as well as the large real-life dataset used during the experiments. The generated datasets cover a wide variety of different sizes, densities and other graph properties that could affect the graph coloring problem. The various experiments performed with these datasets are described in more detail in Section 6. Details on the implementation of the dataset generation can be found in Appendix B. The real life dataset used in this work is a modified version of the one used in [12]. These modifications are necessary in order to make the dataset suitable for use in the dynamic graph coloring problem. Using the timestamps in this dataset we simulate a changing graph. This real-life dataset is much larger than the generated ones, and allows us to verify how well the generated experiments represent a real-life use case.

The rest of the work is structured as follows. Section 2 presents the terminology and prerequisite knowledge required to understand this work. In Section 3 each of the compared algorithms is introduced in short, after which Section 4 expands on the exact approach taken and specific implementation used for each algorithm. The new combination algorithms are introduced in Section 5. Section 6 introduces the experiments ran as part of this research and Section 7 includes the most important results. Finally, Section 8 and 9 draw conclusions based on the results and discuss what could be done to further this research.

2 Preliminaries

This section contains a detailed explanation of the terminology used in this work. All definitions of recurring terms are *italicized*. Apart from introducing this terminology this section also contains the exact definitions of the dynamic graph coloring problem that is used in this work.

2.1 Colorings and Recolors

Given a graph, a *coloring* is an assignment of colors to the vertices of the graph. Such a coloring is *valid* if all vertices have been assigned a color and no edge has two endpoints with the same color. Such a valid coloring uses a limited set of distinct colors. In many cases we are interested in the size of this set which we refer to as the *total number of colors used*. This total is lower bounded by the *chromatic number* C, which is the optimal, or lowest, number of colors required to get a valid coloring for a graph. Note that in general finding the chromatic number is NP-hard. In this work we therefore approximate C by running a static greedy coloring algorithm on a graph which does not guarantee, but is likely to produce a 'good' coloring with close to C colors. We call the approximation $\hat{C}$. The particular static greedy coloring we use is the one described in [5] and detailed in Section 4.1, and has been chosen for its simplicity and popularity in research papers and graph libraries.

Often we will use numbers to refer to the different colors as some algorithms require a strict ordering on the colors. That is, some algorithms use the concept of a *lowest color*.

Next to the total number of colors used we are also interested in the number of color changes with each graph update. When an edge is added to the graph it may cause a *conflict* when the two endpoints of this edge have the same color. This requires at least one, but often multiple *recolors* to again create a valid coloring.

2.2 Problem definition

With these definitions we can define the dynamic graph coloring problem considered in this work more precisely: The input of the problem will consist of two parts, the *initial graph* and the *update sequence*. The initial graph will consist of all nodes required at any point during the update sequence and a set of edges forming the initial connections. The initial graph is assumed to start with a good coloring, which we generate using the static greedy algorithm from Section 4.1. The second part of the input, the update sequence, is represented by a list of tuples, each tuple consisting of an edge and a Boolean stating whether that update corresponds to the *addition* or *removal* of that edge. Note that vertex additions and removals can also be performed so we will not consider them separately here.

The goal is then to compute a valid coloring on the initial graph and ensure that after each edge addition or removal in the sequence the coloring is valid, potentially. Note that we consider an online model where the algorithms do not have access to the entire sequence. We will evaluate the algorithms on (1) the total number of colors used, (2) the number of recolors and (3) the total computation time.

2.3 Trade-off and Variables

In many of the algorithms we discuss there is a trade-off between the total number of colors used and the number of recolors. In fact several algorithms will have a *parameter* that can be used to shift the balance between these optimization criteria.

Some additional important variables are N, representing the number of nodes in a graph and Δ, representing the *maximum degree* of a graph at some point in the update sequence. The maximum degree can change during an update sequence, which is difficult to track. Therefore, we decide not to use Δ in the algorithms, but rather replace it with δ where necessary, with δ representing the *local degree* of a node in the graph. While Δ is often used in bounds for the number of colors or recolors in graph coloring algorithms, the algorithms considered here also function properly by using the local degree δ. We thus decide to use δ in the algorithms, but keep Δ as an upper bound to reason with.

2.4 Black-box Algorithms

Finally, some algorithms make use of *black-box algorithms*, meaning the algorithm will use a different graph coloring algorithm to create a coloring for some (sub)graph, of which the result can later be used to create a more complete or efficient coloring. The input and output of such a black-box algorithm are accessible to the overarching algorithm but the fact that it is a black-box means the inner workings of the algorithm being used as a subroutine cannot be influenced, changed or observed. This also means that any algorithm with the correct input and output can be used as such a black-box, if a more efficient algorithm is found, the algorithms currently used as black-boxes can be swapped out for the more efficient variant, making the overarching algorithm more efficient as well without having to redesign anything.

3 Introduction to the Algorithms

In this section we introduce the algorithms by assigning them a unique name, providing an intuitive notion of their workings, stating known asymptotic bounds and presenting their potential strengths or weaknesses. Table 1 summarizes this section.

Static Greedy Algorithm (4.1) The static greedy algorithm, as described in [5] and [13] colors the nodes of a given graph using at most $\Delta + 1$ colors. When other algorithms require a subroutine for coloring a static graph we use the static greedy algorithm.

Random Warm-Up (4.2) This randomized coloring algorithm is a combination based on the two warm-up results from [6]. It uses $\Delta + 1$ colors where Δ is the maximum degree in the graph at the time of the coloring. The random warm-up uses randomness to recolor nodes and is expected to be the simplest and fastest algorithm discussed in this work. It is additionally expected to only use very few recolors, due to recoloring at most one node per update and low probability of causing conflicts. This random warm-up algorithm is also the algorithm that is used whenever another algorithm requires a dynamic black-box algorithm as a subroutine.

Small- and Big-Bucket Algorithms (4.3) The small- and big-bucket algorithms as described in [7] use sets of buckets, each with their own color palette, to color the nodes in a graph. By splitting the nodes up in such a way, the advantages of both static and dynamic graph coloring algorithms can be exploited. Both of the algorithms use a parameter d to manage the trade-off between number of colors used and number of recolors per update. The two algorithms specialize in using few colors or needing few recolors respectively, meaning they complement each other to cover a larger trade-off range.

Static-Dynamic Algorithm (4.4) The static-dynamic algorithm is described as the algorithm for general graphs in [8]. This algorithm uses a parameter l to manage its trade-off, and revolves around using a dynamic graph coloring algorithm to resolve conflicts in some cases and a static graph coloring algorithm in others. To obtain a valid coloring from this algorithm, each vertex is assigned a tuple consisting of both a static and dynamic color (c_1, c_2), making the expected total number of colors used in this algorithm comparatively high.

DC-Orient (4.5) The DC-Orient algorithm as described in [9] does not focus on the number of recolors per update, but rather aims at simulating the static greedy algorithm from Section 4.1 in a dynamic manner. By updating the colors in this way the coloring and thus also the total number of colors used is generally identical to the one that would be generated by the greedy static algorithm. As a trade-off, however, no bound is known on the number of recolors per update, and this value, as well as running time, are both expected to be relatively high when compared to the other dynamic algorithms.

Table 1: Known bounds and expectations of the various algorithms. *where $\beta = \frac{\log N}{l}$

Algorithm	Colors Used	Recolors / Update	Expectation
Static Greedy	$\Delta + 1$	-	Close to C colors, never recolors
Random Warm-Up	$\Delta + 1$	1	Very fast, focus on using few recolors
Small-Bucket	$\mathcal{O}(dN^{1/d}C)$	$\mathcal{O}(d)$	Wide trade-off with focus on recolors
Big-Bucket	$\mathcal{O}(d)$	$\mathcal{O}(dN^{1/d}C)$	Wide trade-off with focus on colors
Static-Dynamic*	$\hat{O}(\frac{C}{\beta} \log^2 N)$	$\mathcal{O}(\beta)$	Inefficient trade-off, high number of colors
DC-Orient	$\Delta + 1$	Unknown	Close to C colors, high recolors

4 Implementation Details

In this section the workings of the algorithms are expanded upon to allow for easier understanding and reproduction of the implementations used in this paper. Each algorithm is described textually as well as using pseudocode if none was present in the cited paper.

4.1 Static Greedy Algorithm

The static greedy algorithm used in this paper is the greedy coloring algorithm provided by the NetworX Python library [14], which corresponds to the Greedy-Color method described in [13].

It assigns each node a priority based on its degree, in which higher degree nodes gain higher priority, and arbitrarily orders the nodes with identical degree. Using this priority ordering the algorithm colors the nodes one by one, starting with the node that has the highest priority. Whenever a node is colored in this way, it is assigned the lowest color that does not cause conflicts with any of its neighbors. Since any node in the graph can have at most edges equal to the maximum degree Δ, there can be at most Δ colors that are already occupied by a node's neighbors. Each node can therefore be colored using a color value of $\Delta + 1$ or lower.

It is worth noting that, while the upper bound of colors used is $\mathcal{O}(\Delta{+}1)$, this number is often lower in practice, achieving total colorings with close to C colors.

4.2 Random Warm-Up

The random warm-up algorithm used in this work stems from the results described in [6]. The algorithm we use is similar to Bhattacharya's first warm-up result, but has been modified slightly to suit our needs. When a conflict occurs due to an edge insertion, the random warm-up algorithm recolors one of the nodes involved, specifically the one that has been recolored most recently. It does so by creating a set of colors not occupied by any neighbors and changing the color of the node in question to one of these colors, picked uniformly at random. This set of free colors is obtained by creating a set of $\delta + 1$ colors, where δ is the degree of the node to be recolored. This set represents the complete color palette, after which we remove the colors already occupied by at least one of the node's neighbors. The set of δ neighbors adjacent to the node in question can cover at most δ colors. The color set of $\delta + 1$ colors thus always includes at least one color after removing the occupied ones. This guarantees the conflict can be resolved by recoloring the chosen node.

The paper by Bhattacharya et al. [6] also describes methods to make the algorithm more resistant to malicious adversaries and make it easier to formally analyse. In the scope of this work, however, we will only experimentally analyse the algorithms and will not consider malicious algorithms. We thus opt for the simplest variant of the algorithm.

Note that the use of δ in this algorithm allows the algorithm to function without knowing the value of Δ, while still achieving a bound of $\Delta + 1$ colors used and at most 1 recolor per update.

4.3 Small- and Big-Bucket Algorithms

The small- and big bucket algorithms as proposed by Barba et al. in [7] are two complementary algorithms that allow for a trade-off between recolors and total number of colors used. Central to both of these algorithm is the idea of vertices being put in different buckets. Each time vertices are added to a bucket a static black-box coloring algorithm is used on the subgraph induced by the vertices in that bucket. Since each bucket uses its own independent color palette, if each bucket is validly colored this means the union of all these colorings is also a valid coloring for the entire graph.

The complementary aspect of the algorithms is that, depending on a parameter d, these algorithms can achieve different ranges of the trade-off between recolors and total number of colors used. The small-bucket algorithm uses many small buckets and favors fewer recolors. It uses only $\mathcal{O}(d)$ amortized recolors per update, but uses $\mathcal{O}(dN^{1/d}C)$ colors. The big-bucket algorithm uses fewer larger buckets and skews the other way. It favors fewer total colors, using only $\mathcal{O}(dC)$ colors but requiring $\mathcal{O}(dN^{1/d})$ amortized recolors per update. As parameter d increases, the small- and big-bucket algorithms produce more similar results up until they converge at $d = \log N$, after this point, higher values for d will have no effect.

The overall structure of the small- and big-bucket algorithms consists of a set of buckets divided over different levels. When a node is added to one of these buckets, that bucket's coloring is recomputed using the static black-box coloring algorithm. If the buckets on a level are deemed to be full, they are all grouped together and moved into the first bucket on the next level. The sizes of the buckets as defined in the algorithm ensure such a transfer is always possible. After moving up a level, the new bucket will statically recolor its nodes. This process of adding vertices to buckets and moving nodes from lower to higher levels continues until the end of the leveled structure is reached and a full reset occurs.

Depending on the parameter d the algorithms will use more or fewer levels of buckets, each with different capacity. A visual representation of the bucket levels and each bucket's capacity is shown in Figures 1 and 2.

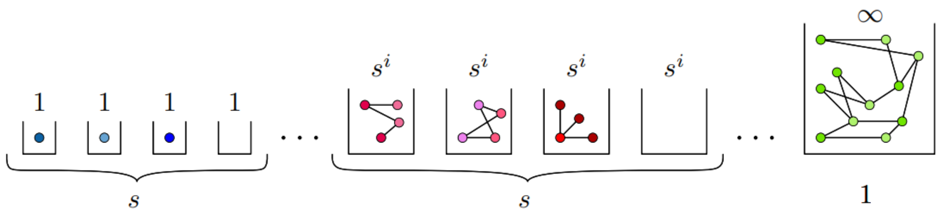

Figure 1: Visual representation of the buckets used in the small-bucket algorithm. Figure taken from [7].

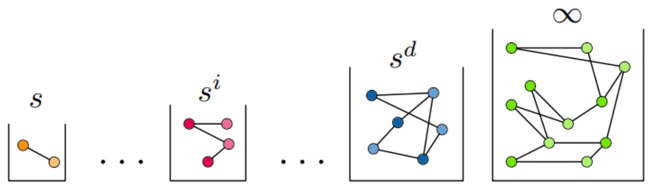

Figure 2: Visual representation of the buckets used in the big-bucket algorithm. Figure taken from [7].

Since the small- and big-bucket algorithms serve only as a comparison algorithm in this work, and no parts of this algorithm are reused in the new combination algorithms, we do not provide a more detailed description of its workings. More details about the algorithm can be found in [7] and the pseudocode used to implement both the small- and big-bucket algorithms can be found in algorithm 1 and 2, located in appendix C.

4.4 Static-Dynamic Algorithm

The static-dynamic algorithm for general graphs as described by Solomon et al. in [8] aims at combining static and dynamic black-box algorithms in a way that allows for a trade-off between the advantages of both. In order to achieve this the algorithm uses two representations of the same graph: a full graph G and a sparse variant G' of the same graph. The static-dynamic algorithm uses a leveled structure to keep track of the updates and, based on this structure, resolves a conflict with either a static black-box algorithm or a dynamic one. Static black-box recolors are performed on the full graph G, whereas dynamic black-box recolors are performed on the sparse graph G'. This method of recoloring can cause a conflict to only be resolved in one of the two graphs and remain in the other. In order to obtain a graph coloring without conflicts the static-dynamic algorithm combines the two colorings from G and G' into a single one, by defining a color within the new coloring as a tuple $(c1, c2)$, with $c1$ being the color a node has in G and $c2$ being the color a node has in G'. The number of times a static coloring step is taken instead of a dynamic one can be influenced by the parameter l, on which the authors base their bounds of $\hat{O}(\frac{C}{\beta}\log^2 N)$ total colors and $\mathcal{O}(\beta)$ expected recolors per update, where the total color bound suppresses polyloglog(N) factors and $\beta = \frac{\log N}{l}$.

We expect (and confirm in Section 7) that this approach requires many colors. For example, during a reset step, which occurs when the end of the layered structure is reached, the graph G is reset by running a static black-box algorithm on the entire graph. The coloring in G after this reset step is thus completely valid. The description of the static-dynamic algorithm does not mention resetting the graph G', however. This means that even after the reset step the combined coloring $(c1, c2)$ still uses many more colors than necessary, as the colors from G' could be left out completely.

From [8] the definition of the subgraphs on which static black-box algorithms should be ran during conflict resolution is unclear. The paper by Solomon et al. defines this set of nodes as those with the highest recent degree at the end of an update interval or subinterval, and while *recent degree* and *update intervals* are clearly defined, it is not clear what is intended with a *subinterval*. For the experiments in this paper the implementation of static-dynamic uses the following definition: A subinterval of an update interval is an interval of any size equal to or smaller than that of the original update interval, and starting at the same point the original interval does. This definition of subinterval falls within the general expected definition of subinterval.

In section 5.1 we propose a variation of this algorithm that solves some of the problems identified here.

An alternative version of this algorithm that does reset the dynamic graph during a full static recolor as well has been implemented too. A small comparison between this variation and the main version of the algorithm can be found in Appendix A, Section A.1. The variation is not used in the remainder of this paper, however.

The pseudocode for the static-dynamic algorithm can be found in Algorithm 3, located in appendix C. Note that we do not presume to know all update steps in advance and thus do not explicitly generate the 'update segments' in advance, but rather keep track of our position within the leveled structure by using counters.

4.5 DC-Orient

The final algorithm considered in this work is DC-Orient as described in [9]. What sets this algorithm apart from the rest is its aim to simulate the static greedy algorithm discussed in Section 4.1 in a dynamic manner. The authors observed that when a colored graph is recolored using the static algorithm after one update occurred, only few of the vertices change color in many of the cases. This observation is the only attempt at reducing the number of recolors in this algorithm, as there is no parameter available for a trade-off. It thus heavily leans towards a small total number of colors used and is practically the opposite of the random warm-up algorithm discussed in Section 4.2. DC-Orient does not provide a guarantee on the number of recolors but does achieve the same coloring as the static greedy algorithm.

The algorithm works by defining a priority ordering based on the degree of the vertices in the graph, much like the static greedy algorithm, where high degree nodes get priority over low degree nodes. A directed graph G^* is generated in which the edges of G are pointed from high priority to low priority nodes as can be seen in Figure 3.

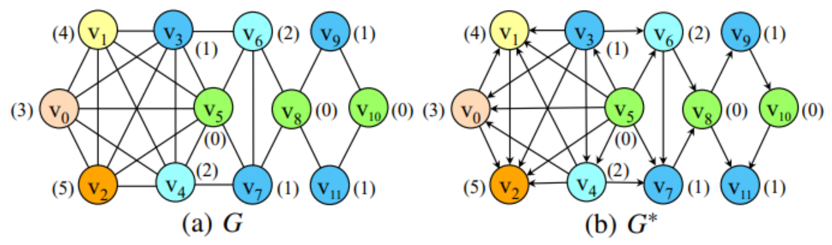

Figure 3: Example of a directed graph G^* created and used by DC-Orient. Figure taken from [9].

When an update occurs, G^* is first updated to reflect the new priority ordering of the vertices and potential conflicts are then resolved by executing a CAN step in which the vertex of highest priority involved in a conflict first *collects* the colors not occupied by it's in-neighbors in G^*, then *assigns* itself the 'lowest' color from the set of available colors and finally it *notifies* it's out-neighbors in G^* that they might have to do a CAN step as well, in case a color was chosen that one of these lower priority nodes had assigned to them. Because G^* is directed and acyclic this chain of CAN steps is guaranteed to finish.

Yuan et al. [9] presents several different versions of the DC-Orient algorithm. They all compute the same coloring, but differ in terms of complexity and running time. The basic version of the algorithm is the easiest to understand, but is very slow in practice. The authors therefore introduce different ways to make their algorithm more efficient. One of these additions is a Dynamic In-Neighbor Color Index (DINC-Index) to keep track of the colors of each node's in-neighbors. By maintaining this DINC-Index the algorithm no longer needs to access each node's neighbors to assign it a new color, which in practice turns out to be much more efficient. Furthermore, some pruning strategies are suggested to avoid unnecessary recursions to nodes that do not cause conflicts. Both the basic and the optimized versions of the algorithm have been implemented, but since the results, apart from the running time, are identical we have decided to use the optimized version in our experiments. A small experiment to compare the running times of both versions can be found in appendix A, Section A.2. The pseudocode for the basic algorithm, as well as for all the extensions, can be found in the original paper [9].

5 Additional Algorithms

In addition to the four classes of algorithms taken from the relevant papers, we have implemented two combinations of these algorithms that can overcome some of the issues the original versions face.

5.1 Static-Simple

During preliminary experiments it became apparent that the static-dynamic algorithm performs objectively worse than other algorithms in many cases. More specifically, the number of colors used was significantly higher than the alternative algorithms for almost all values of parameters. We hypothesise that the main reason for this issue is the multiplication in total number of colors used that occurs when combining the static coloring and the dynamic coloring into a single combined color $(c1, c2)$. We therefore introduce a simpler version of the algorithm named static-simple. The static-simple algorithm is identical to the static-dynamic algorithm when it comes to the static black-box component. The difference with static-dynamic is that there is no sparse graph G' on which a black-box dynamic algorithm is ran, instead, we simply use the same approach as the random warm-up algorithm from Section 4.2, when a conflict occurs that is not solved by any static black-box executions, simply pick one of the conflicting vertices and assign it a random color from the set $\{0, .., \delta\}$ that is not occupied by any of its neighbors yet. Such a color always exists, and by handling dynamic conflicts in this manner we prevent the need of creating combined colors $(c1, c2)$. Bounding this new algorithm is difficult, while the number of colors used is technically bounded by $\mathcal{O}((\Delta + 1)l)$, this is not representative of the actual colors used, since in practice this value is significantly lower when compared to static-dynamic. The number of recolors remains $\mathcal{O}(\frac{\log N}{l})$ on average, as it is in static-dynamic. Additionally, this randomized process for solving conflicts is very similar to that already used in static-dynamic, considering the random warm-up from Section 4.2 is used as the dynamic black-box in that algorithm.

5.2 DC-Random

The second new combination of algorithms aims at creating a trade-off between the two extreme algorithms DC-Orient and the random warm-up. These algorithms both provide excellent results in one aspect, but less than desirable results in the others. The random warm-up manages to handle updates with very few recolors in a very short time, but uses a lot of colors in the process. DC-Orient on the other hand achieves a high quality coloring with a low number of colors but needs many recolors and takes long to run. By combining the two algorithms and adding a parameter to control which of the two to use, we allow a trade-off between recolors, number of colors and running time. The resulting algorithm is DC-Random, which combines DC-Orient and the random warm-up algorithm in the simplest way possible. A parameter p between 0 and 1 is added to the algorithm, representing the probability of taking a random warm-up step rather than a DC-Orient step. A random step uses the same approach as in static-simple: one of the conflicting nodes is assigned a random color from the set $\{0, .., \delta\}$ not occupied by one of its neighbors. When a DC-Orient step is executed, the algorithm does exactly what DC-Orient would do, and thus intuitively 'overwrites' some of the randomly chosen colors in its CAN step chain. We thus expect the total number of colors used to remain quite low, while the average number of recolors should be reduced. Note that in the optimized version of DC-Orient the DINC-Index needs to be maintained whenever an update occurs, in the optimized version of DC-Random we therefore always maintain this DINC-Index, even if a random step is taken. A small comparison between the basic and optimized versions of DC-Random can be found in appendix A, Section A.3.

We note that both the random warm-up algorithm and DC-Orient have the same bound on total number of colors used, namely $\mathcal{O}(\Delta + 1)$. By combining these algorithms, knowing neither algorithm will ever assign a vertex a color outside of the color palette $\{0, .., \Delta\}$ we can conclude that DC-Random will also use at most $\mathcal{O}(\Delta + 1)$ colors in the worst case, although in practice it is likely to use fewer. The expected number of recolors per update is difficult to bound because neither of the original algorithms state a clear upper bound on the number of recolors.

6 Experiments

This section describes the experiments performed during this research. The variety of experiments presented here aims to represent many different use cases and find situations in which some algorithms might perform particularly well. Each experiment consists of a starting graph, an update sequence and a set of parameters the algorithms are ran with. This section contains an overview of how these components are chosen for the different experiments.

6.1 Starting graph

Most experiments consist of a generated graph and update sequence. The graph will function as a starting point and the update sequence will provide updates to this graph. The implementation of these allows us to generate graphs and update sequences with various properties. The graph can have a predetermined number of nodes or edges and allows for different distributions of the degrees in the graph.

The most basic graphs generated are simple random graphs with a set number of vertices and a parameter for edge density. This density parameter represents the probability that an edge is actually present in the graph. Apart from being able to change the density of the starting graphs, the generation of these graphs also allows for a skewed distribution of edges, in which nodes are assigned a priority, giving higher priority nodes more edges. The strength of this effect can be adjusted and thus allows for experimenting with the effect of this bias on the different algorithms. A more detailed description of the implementation of this skewed edge selection can be found in Appendix B, Section B.2.

6.2 Update Sequence

We use two separate methods of generating update sequences to emulate different scenarios. We name these two methods *increasing* and *stream based*. The increasing update sequence allows a lot of control over various aspects of the sequence, whereas the stream based sequence can consist of a larger amount of more random updates.

For the increasing update sequence we generate a sequence of updates that consists of edge additions only, we start with an empty graph that only has vertices, obtained by removing edges from the initially generated starting graph, and add one edge back at a time. This variation allows for the edge additions to occur in a randomized order or in a more organized 'expanding' or 'node focused' order. The expanding order allows us to simulate a breadth-first-search type behaviour, in which the edge insertion updates are ordered in such a way that edges adjacent to the connected component so far are added first with higher probability. Similarly the node focused approach assigns each node a priority and adds edges adjacent to nodes with higher priority first, this means the updates will likely be concentrated on one node at a time. More details about the expanding and node focused approaches can be found in Appendix B, Section B.3.

Alternatively, in the stream based approach, we can generate an update stream consisting of both edge insertions and deletions. In this case we start with the generated graph and add and remove edges with the same probability, such that the number of edges remains largely the same throughout the update stream. The edges that are removed over time can be picked at random or have an increased probability to be removed as they remain in the graph for a longer period. These two options are given the names random stream and decaying stream. Note that in the stream variant of the experiments, the generated graph, including its edges, is only the starting point. The graph only changes when the update sequence adds or removes edges. More information on the decaying approach is presented in Appendix B, Section B.1.

Optionally the generation of both the graph and update sequence allow for some randomized 'variation' such that two graphs or sequences generated with the same parameters will result in similar but different outputs. It is therefore possible that graphs intended to fall into the same range, size or density-wise, slightly differ from each other. This slight variation should help reduce the chance of a recurring outlier.

6.3 Performed Experiments

With these options in place to generate starting graphs and update sequences, there are many experiments that could be performed. In this paper, we focus on performing experiments that change only one component at a time. This allows us to better see the effect of that change on the various algorithms. The experiments considered in this work are as follows:

Small vs. Large The small vs. large experiment focuses on how algorithms perform on differently sized graphs. This experiment contains two phases, one in which the graph size is adjusted by changing the number of nodes in the starting graph and a second which changes the density of the graph. Both of these phases use the method of an increasing update sequence. Seeing how the algorithms perform on graphs with a different number of nodes or edges gives an idea of which of them will scale well to a real use case.

Update Stream Similarly the update stream experiment aims at simulating the algorithm being used continuously for a long time. An update sequence is created using the stream based approach. This sequence can thus be made very long, but will maintain the properties of the graph, such as density and size, during execution of the updates. This experiment will show whether algorithms can be used for extended periods of time, and how their performance may deteriorate over time. More details about how this update sequence is generated are provided in Appendix B, Section B.1

Degree Variation The degree variation experiment aims at finding out if certain algorithms perform better or worse in situations that are not uniform, but biased in some way. This experiment assigns some nodes a high priority, meaning more edges are generated adjacent to them. The increasing update sequence approach is then used to add these biased edges into the graph one by one. The skewed update sequence simulates one aspect of what a real world update sequence may look like. Details about generating such a skewed degree distribution can be found in Appendix B, Section B.2.

Update Spread The update spread experiment has a similar goal, but rather than changing the distribution of density it changes the order of the updates in the update sequence. It does so by using the increasing update sequence approach, where the edges to be added are, partially, sorted on which nodes they are adjacent to. Finding out whether the order of updates affects the algorithms in any way opens up the possibility of specializing algorithms for specific use cases in which this ordering is known or can be changed. Additional information on generating a partially ordered update spread are provided in Appendix B, Section B.3.

Reddit Dataset Apart from these generated graphs and update sequences we also consider a real world Reddit dataset. This dataset, as found in [12], models the hyperlinks between various subgroups called 'subreddits' on the social media platform Reddit. This dataset, contrary to most generated experiments, provides us with some real-world properties and biases, such as being much larger, but also rather sparse around some nodes and highly concentrated around others. The aim of this final experiment is to see whether the results obtained from the generated graphs are representative for real-world usage of the algorithms, and otherwise, what differences occur in their performance.

All algorithms were implemented in Python 3 using the NetworX library [14]. The experiments were executed using the notebook functionality in Visual Studio Code on a system with an octacore Intel i7-6700K CPU at 4 GHz and 16 GBs of RAM. The timing of the algorithms was performed using the perf_counter() functionality available in Python's time library.

6.4 Experiment Parameters

In each generated experiment the algorithms are ran multiple times on the same data. Each instance with different parameter values in order to get a better impression of how the algorithm trade-offs work. Each algorithm is executed with up to a hundred different parameter values and any algorithms using a lot of randomness are ran three times as often, using the same parameters, in order to find a more stable average performance. Table 2 provides the exact parameter ranges per algorithm and notes on why these ranges were chosen.

Table 2: Parameter ranges used for each algorithm during the experiments

Algorithm	Runs	Parameter	Reasoning
Random Warm-Up	3	-	Has no parameters
Small-Bucket	1	1 - 30	Converges with Big-Bucket at parameter 30
Big-Bucket	1	1 - 30	Converges with Small-Bucket at parameter 30
Static-Dynamic	3	1 - 200	Wide range because source provides little guidance
DC-Orient	1	-	Has no parameters
Static-Simple	3	1 - 200	Same as Static-Dynamic to compare easily
DC-Random	3	0.4 - 1	Total range is 0 - 1, higher values proved more interesting

7 Results

This section presents and discusses the results obtained from the experiments. The section is divided into three parts. First the generated figures and how to read the outcomes are explained. Secondly, the general observations are discussed, which are outcomes that are common over most of the experiments. Finally, the experiment specific results are presented, which are findings more specific to a specific attribute of the input or algorithm.

7.1 Reading the Results

The results obtained from the experiments are presented in this section. Most of these observations are described in natural language combined with a visual plot of the results. Each of these result plots consists of two graphs forming a graph pair, containing a plot displaying the relation between average number of colors used and average recolors on the left side, and a plot displaying the relation between total time taken per instance of the algorithm and average recolors on the right side. Because the x-axis for both plots are identical, the two plots can be combined in order to obtain all three components of the experiment outputs: number of colors, number of recolors and time taken.

The lines present in each plot represent the trade-off each algorithm can make. Each point a line passes through corresponds to one datapoint obtained by running the algorithm on the experiment input with a certain parameter. The arrow displayed on each line represents the direction of increasing parameters. Some of the algorithms are represented by a point rather than a line, this means the algorithm was not executed using multiple different parameters and no trade-off is therefore visible. In most cases this is due to the algorithms in question not supporting a trade-off, but in the case of the Reddit dataset the long running times prevented more datapoints from being generated.

Because the importance of number of colors, number of recolors and running time can vary from application to application, it is difficult to define when one algorithm performs better than another. Since for all aspects, colors, recolors and running time, lower values are preferred, it means algorithms with datapoints closer to the left-bottom corner are those that perform best. We will focus largely on whether lines or points lie left of or below other lines in order to compare them. Additionally, it is worth noting that the axis scales for each experiment are different, so we will compare relative locations between the datapoints of various algorithms within a plot, and how they compare to the ideal estimate for the average chromatic number C.

7.2 General Observations

Most of the plots generated by the experiments look rather similar, we therefore first discuss the aspects of the results that are generally the same, before going into detail per experiment. We discuss the general observations per algorithm below. Figure 4 in Section 7.3 and Table 10 in appendix A provide a visual and numerical representation of the most general experiment to support these observations.

Random Warm-Up The random warm-up algorithm uses randomness to ensure at most one node has to be recolored during an update step. This approach causes the number of recolors and the running time to be low, but sacrifices on number of colors used. In the plotted results we indeed see that the random warm-up algorithm uses the least number of colors and more colors than most other datapoints, with the exception of Static-Dynamic. We will thus consider the random warm-up one of the extremes when it comes to the trade-off between colors and recolors: namely the one focusing on low number recolors.

Small- and Big-Bucket Algorithms Together, the small- and big-bucket algorithms strike a balance between colors and recolors. Their total trade-off spans the distance between both extremes, and curves towards the bottom left corner in the middle. It thus seems like the small- and big-bucket algorithms are a competitive alternative to many of the others presented in this paper. The plots also confirm that as the parameter for these two algorithms increase, they produce more similar results before converging in the middle. Apart from the color and recolor trade-off, the small- and big-bucket algorithms also perform competitively when it comes to their running time. It is worth noting that seemingly the number of recolors done per update is related to the

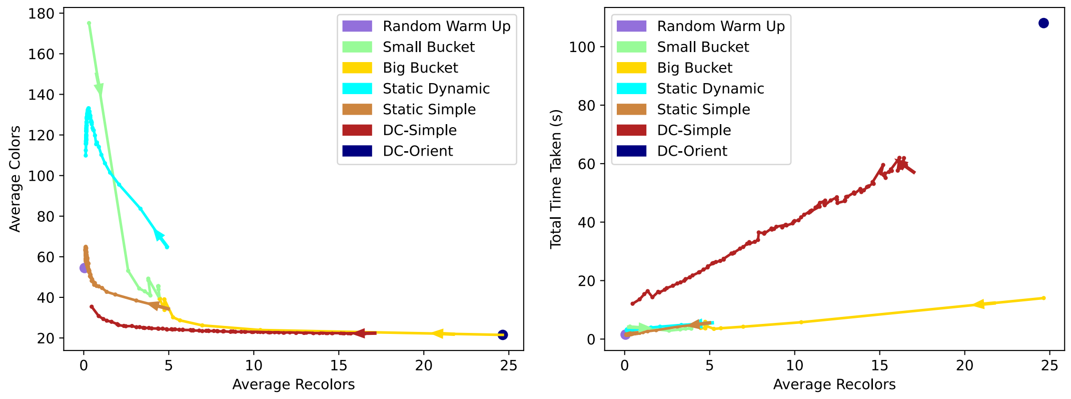

Figure 4: Results for a medium number of nodes with random edges and random update sequence. 200 nodes, 11940 edges and estimated average $\hat{C} = 22$. Representative of what most of the resulting plots look like.

running time. This can be explained by the algorithms having to perform more complex tasks when a recolor occurs when compared to no conflicts occurring during an update.

Static-Dynamic Algorithm As expected, the experiment plots show that, even in the same range of recolors, the static-dynamic algorithm requires many more colors than all other algorithms (with the exception of the small-bucket algorithm with parameter 1). This result is explained by the fact that this algorithm uses two different colorings, the static coloring and the dynamic coloring, which it combines by creating color tuples $(c1, c2)$. This process practically multiplies the number of colors used in the static coloring with those used in the dynamic coloring, creating an unnecessarily large color palette. This effect is most noticeable in the middle of the parameter range, when the static coloring and the dynamic coloring use more or less the same number of colors. Multiplying these together results in a peak in the total number of color tuples $(c1, c2)$ when compared to using mostly the static coloring or dynamic coloring at higher or lower parameter values. The experiment plots also show that the high number of colors does not get outweighed by a very small number of recolors or significantly better running time. Making this algorithm less suitable than the alternatives.

DC-Orient Almost all plots show DC-Orient requiring the most recolors and the least number of colors, making this algorithm the second extreme: the one focused on number of colors used. The high number of recolors, however, also seems to result in a very high running time.

Static-Simple Static-simple was designed to be an improvement over static-dynamic. Where static-dynamic combined the random warm-up algorithm and DC-Orient by using them as black boxes and later combining their separate colorings, static-simple instead uses only a single coloring. This simple change in the way these algorithms are combined removes the blow-up of colors used that was present in static-dynamic and, as can be seen in the experiment plots, reduces the number of colors significantly while retaining the same range when it comes to recolors. The running time, while differing somewhat, is still competitive with static-dynamic and the other algorithms and overall it thus seems that this improved variant of static-dynamic has succeeded in its goal. The relatively small range of recolors that was covered by static-dynamic, especially when compared to the small- and big-bucket algorithms, has not been increased however, meaning that while the trade-off is much more efficient, it is not as versatile as certain other options.

DC-Random Finally, DC-Random, created to be a of the random warm-up algorithm and DC-Orient is shown to have succeeded in its purpose. Not only does its trade-off line move accurately between both of these algorithm results, it does so with a bottom-left curve that is lower than the other algorithms considered so far. This means that the trade-off between colors and recolors offered by this algorithm is the most efficient one presented here, making this the ideal option when both colors and recolors are important for the use case in question. This efficient trade-off comes at a cost, however, since the running time aspect of this algorithm is less than desirable for most parameter values. While the running time of DC-Random does considerably improve upon that of DC-Orient, even for relatively low parameter values, it is only competitive with the other algorithms at high ones. This high running time is something to consider when deciding on an algorithm for a specific use case. Additonally, peaks in the trendlines for running time can be observed in some of the plots for DC-Random. These peaks are likely not due to the parameter values of the algorithm, but rather to background processes on the computer slowing the experiments down.

With these general observations about the different algorithms clarified, the rest of the experiments will focus on the differences in the results that occur when running the algorithms on different types of graphs and update sequences. These experiments will allow insight into different situations in which certain algorithms might have an advantage due to how they work internally.

7.3 Small vs. large graphs and edge density

This experiment shows the change in how the algorithms perform on smaller or larger graphs. The visual results for a small number of nodes and a large number of nodes can be found in Figures 5 and 6. In addition to this comparison, a zoomed-in version of the running time plot in Figure 6 is provided in Figure 7. The experiments on low and high density graphs show the same results. For brevity their results are omitted here, but can be found in Appendix A.

In the results for increasing number of nodes, it is clearly visible that at a small number of nodes, three different algorithms eventually converge at the random warm-up results. The static-dynamic, static-simple and DC-Random algorithms do indeed all make use of this random warm-up as a subroutine. An extreme value for their parameters can thus cause their results to become very similar. As the number of nodes increases, however, we see that the number of colors used by the random warm-up increases at a much faster rate than our estimate for $\hat{C}$. This acceleration could indicate that the random warm-up and the two combination algorithms, with high parameters, may not scale well on even larger graphs, potentially due to the lack of reset steps in the algorithms, resulting in more suboptimal colors being left in the coloring, or potentially due to the average degree being higher in larger graphs, thus increasing the palette size of these algorithms. The small- and big-bucket algorithms display the opposite effect: in the small example, the combined line of these algorithms is almost straight from the top-left to the bottom-right corner, only a small part of this range is interesting, as the other algorithms perform objectively better for most of the range.

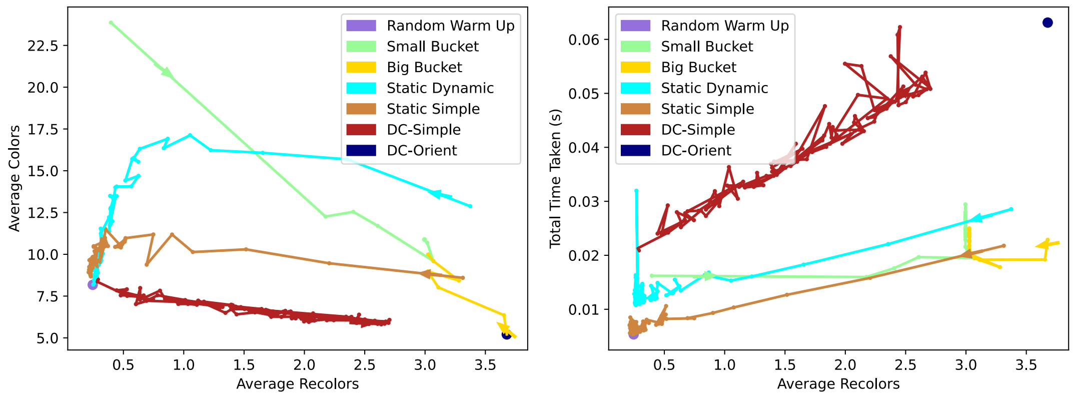

Figure 5: Results for a small number of nodes with random edges and random update sequence. 30 nodes, 217 edges and estimated average $\hat{C} = 5.21$.

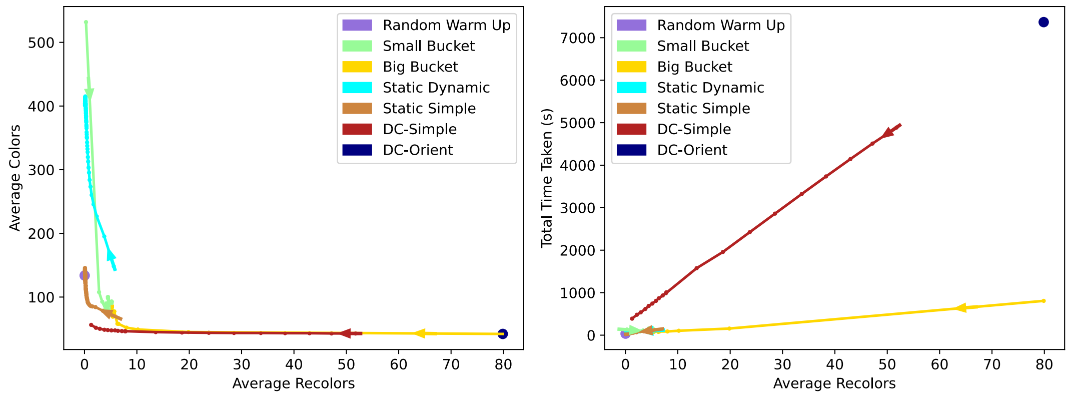

Figure 6: Results for a large number of nodes with random edges and random update sequence. 600 nodes, 89850 edges and estimated average $\hat{C} = 42$.

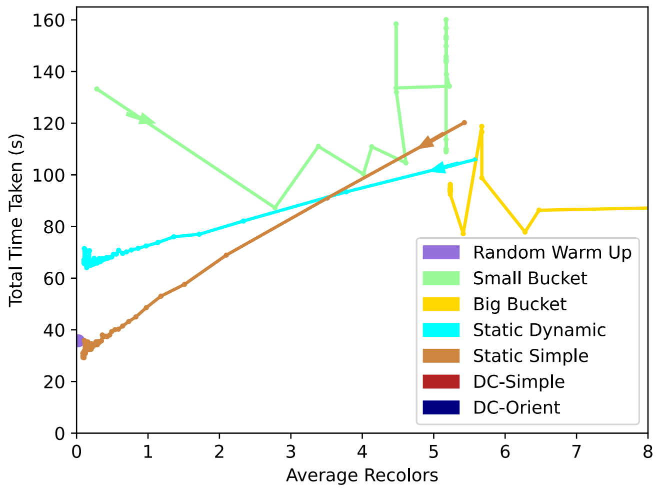

Figure 7: Zoomed-in plot of the running time from Figure 6.

As the number of nodes increases in the medium and large example, however, we find that this combined line curves to the bottom-left in an increasingly strong manner, making the algorithms competitive on a much larger part of the covered range. Finally, DC-Orient consistently performs well when it comes to number of colors used, but as the graphs become larger starts falling behind in number of recolors and running time quickly. This also explains why the running time line of DC-Random seems to become steeper as the graphs get larger: it traces a line between the random warm-up and DC-Orient, and DC-Orient increases its running time at a much faster rate than the random warm-up does.

Finally, in the zoomed-in plot of Figure 7 we can see that the random warm-up does seem to be the fastest option in general, allowing for some random variation when comparing it to the static-simple line. Apart from this result, however, it seems none of the algorithms consistently perform the quickest. The static-dynamic and static-simple algorithms are both almost as fast as the random warm-up when their parameters are high, but as their parameters decrease become on-par with the small- and big-bucket algorithms, at which point they would increase at a faster rate if not for the fact that the parameter range ends here. The small- and big-bucket algorithm seems the most consistent in its running time at this small scale, but Figure 6 shows that there is some increase in running time at the large scale, albeit much lighter than that of DC-Random.

7.4 Constant Update Stream

In this experiment we generate a stream of updates including both edge additions and removals, such that many updates are executed without changing the properties like size and density of the graph much. Such an update stream is meant to simulate prolonged usage of the algorithms. We generate two different update streams of a hundred thousand updates, one in which random edges are removed to maintain the same size and the other in which older edges are removed with a higher probability. The starting point of both update sequences is a random graph of a medium

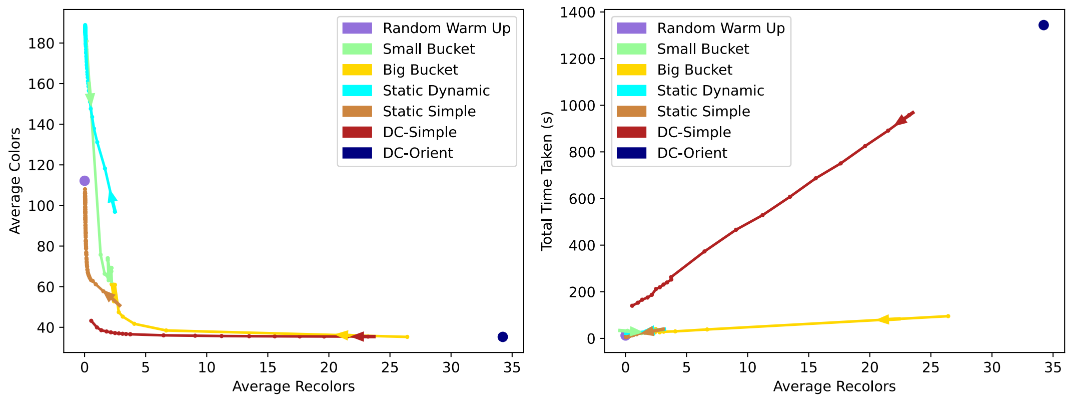

Figure 8: Results for a random stream of 100.000 updates, 200 nodes and estimated average $\hat{C} = 35$.

size, with 200 nodes and 11940 edges.

The variation in which older edges are removed with a higher probability, which we call the decaying update stream, produces results that are similar to those produced by the random update stream. We thus present only the random update stream results here. The results are provided in Figure 8.

From these results it is clear, when comparing them to the results from Section 7.3, that algorithms with reset functionality also perform better on long update streams than the more randomized algorithms do. This reinforces the belief that the algorithms with such a reset step are more scalable.

The random warm-up algorithm uses many more colors than the other algorithms, for example, whereas the small- and big-bucket algorithms remain in the same range as before. The static-dynamic algorithm performs relatively well. It seems the reset functionality of its static component may be countering the increase in colors used caused by the otherwise detrimental multiplication of static and dynamic colors. Surprisingly DC-Random does not produce significantly worse results when executing 100.000 updates, even when nodes are colored randomly with a very high probability parameter. This would indicate that 'incorrect' colors caused by taking random coloring steps are not often left behind in the coloring and are relatively quickly overwritten by a DC-Orient CAN step. This result, combined with the fact that at high parameter values the running time of DC-Random is almost competitive with the other algorithms, makes it a strong contender in the case of long update streams.

7.5 Degree Variation

The degree variation experiment runs the algorithms on similar graphs and update sequences with varying distributions of degrees. A random baseline, light and heavy skew of the degree distribution are generated and the algorithms are ran on all three update sequences. The random baseline and light degree distribution bias generate resulting plots that are very similar to the one in Figure 4, yet the heavily skewed degree distribution results, as displayed in Figure 9, shows an interesting

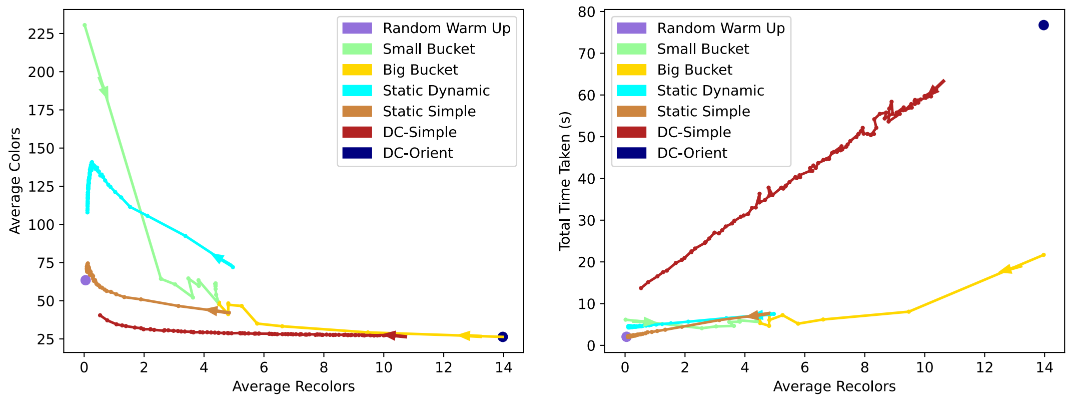

Figure 9: Results for a heavily skewed distribution of degrees. 250 nodes, 13769 edges and estimated average $\hat{C} = 26$.

effect. Apart from these three levels of degree skew, an update sequence is tested in which only a single node has an extremely high degree. This experiment does not appear to affect the algorithms significantly, however, and its results are thus not presented.

As can be seen when comparing Figure 9 to Figure 4 we can conclude that almost all algorithms have very similar performance to that in other experiments, regardless of the distribution of degrees. The one point of interest here is that DC-Orient, and to a lesser extent therefore also DC-Random, seem to require fewer recolors as the distribution of degrees becomes more skewed, an effect which causes the other lines in the plot to appear more spread-out horizontally. We thus conclude DC-Orient is the only algorithm performing more efficiently on a graph with skewed degree distribution. This aspect of DC-Orient might make it a more viable option in certain biased situations, rather than only being a slow algorithm focused on achieving a low number of colors. This effect does not seem present when only a single node of high degree is added to the graph, nor does this addition seem to significantly affect the other algorithms.

7.6 Update Spread

This section focuses not on the edges added during the update sequence, but on the order in which these updates are executed. The three methods used to generate this update order are: randomly, prioritized based on an expanding breadth-first-search principle, where all edges are likely to be part of a single connected component, and lastly prioritized on node, where updates adjacent to a node with high priority are more likely to occur early in the update sequence. Note that the node prioritized update sequence groups the updates together much more strongly than the update sequence ordered in an expanding fashion.

Comparing the results of the random and expanding update sequences displays an effect very similar to the one observed in the degree distribution experiment in Section 7.5. The node prioritized update sequence results, as displayed in Figure 10 takes this effect to a new extreme.

We once again see DC-Orient profit from a skewed update set. The average number of recolors for DC-Orient drops as low as the convergence point of the small- and big-bucket algorithms. This

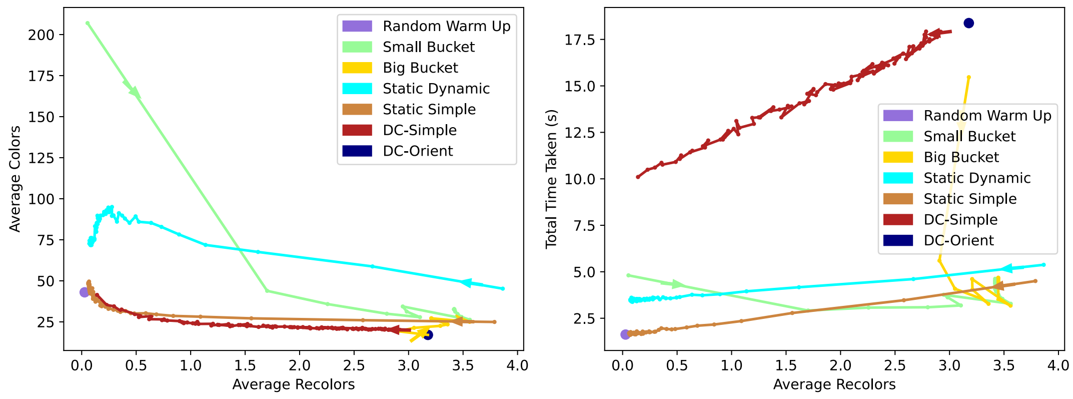

Figure 10: Results for an update sequence with strong node prioritization. 228 nodes, 12440 edges and estimated average $\hat{C} = 17$.

decrease in recolors also allows the running time of DC-Orient to be almost equal to the other, usually faster, algorithms. This surprisingly effective behaviour of DC-Orient, especially combined with the results from the degree distribution experiment in Section 7.5, make it a much more versatile option than initially expected. The DC-Random algorithm can be seen to profit to a lesser extend from these same skewed update sequences, as it only uses a DC-Orient step occasionally, depending on the chosen parameter. This surprising location of the DC-Orient datapoint also effects the big-bucket line, as the big-bucket algorithm with a very low parameter often uses a reset step, which has been implemented using the static greedy coloring algorithm that DC-Orient aims to simulate. It is therefore also visible in this plot that the big-bucket algorithm line moves towards the position of DC-Orient, which creates a significantly different curve compared to the other visualisations, as this is not where the DC-Orient datapoint would normally lie.

7.7 Reddit Dataset

For the final experiment, we use a real-life dataset representing hyperlinks between different 'subreddits' on the social media platform Reddit. This dataset originates from a Stanford University research paper by Kumar et al. [12]. Some preprocessing was done to make this dataset applicable to our work. The directed edges in the dataset were interpreted as undirected ones and any duplicate edges caused by this process were removed. Remaining are 35776 nodes and 124330 edges, which the greedy static coloring algorithm can color using only 34 colors. The graph is thus large, but not very dense. In this experiment we hope to see which algorithms work best in a real-life example, rather than generated graphs with specific properties.

The results of this experiment can be found both in Table 3 and Figure 11.

We find that most algorithms behave as expected. The random warm-up uses relatively many colors but has a very low number of recolors and running time. The small- and big- bucket algorithms do not provide the best results but do achieve a fairly efficient trade-off. Static-dynamic uses a total number of colors that is much higher than all other algorithms and static-simple simply improves upon static-dynamic in all aspects. What is surprising about these results is how well DC-

Table 3: The results of running the algorithms on a real-life Reddit dataset.

35776 Nodes; 124330 Edges; $\hat{C} = 23.05$	Average #Recolors	Average #Colors	Time Taken (s)
Random Warm-Up	0.11	78.3	3321.593
Small-Bucket algorithm (d = 5)	3.93	137.31	3494.585
Big-Bucket algorithm (d = 5)	5.39	51.1	3167.058
Static-Dynamic algorithm (l = 10)	1.24	283.54	7779.184
Static-Dynamic algorithm (l = 100)	0.22	343.64	7375.252
DC-Orient	2.48	23.05	7444.115
Static-Simple algorithm (l = 10)	1.13	94.65	2862.408
Static-Simple algorithm (l = 100)	0.2	93.35	2904.748
DC-Random (p = 0.8)	24.21	37.16	15795.713
DC-Random (p = 0.998)	2.9	61.8	6857.528

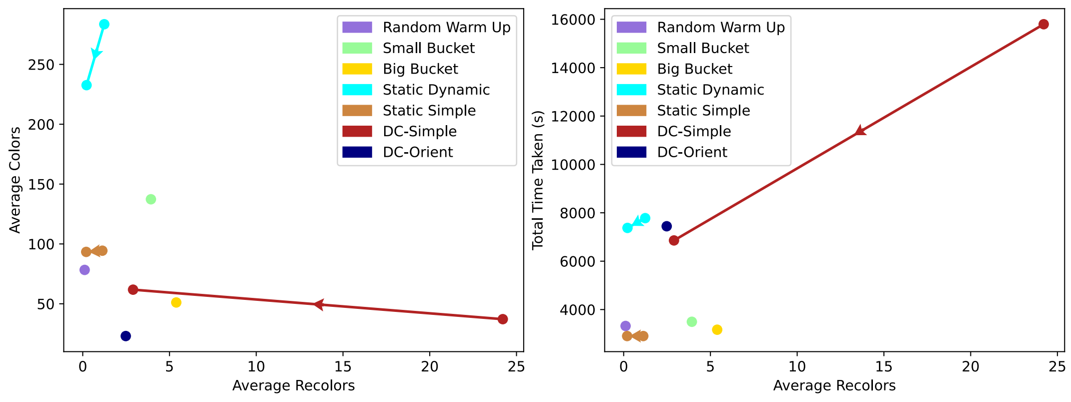

Figure 11: Visual plot of the results of the real-life Reddit dataset, based on the values of Table 3

Orient manages to perform on a large real-life dataset. Whereas previously DC-Orient has usually been the slowest algorithm with by far the most recolors, in this case the number of recolors is quite competitive with the other algorithms even when it has the lowest total number of colors used. Additionally the running time for DC-Orient is on par with static-dynamic and even lower than the DC-Random algorithm for low parameter values, this is unlike what could be observed with the generated graphs. In this case DC-Random is not able to outperform DC-Orient, although the final instance with $p = 0.998$ comes close. The efficiency of DC-Orient on this real-life dataset could be due to an inherent skew in either the edges or order of the updates in the dataset. As has been shown in Sections 7.5 and 7.6, DC-Orient performs better on datasets with some aspect that is not evenly distributed. The authors of the paper from which the dataset originates [13] do mention that many 'conflicts' between subreddits are caused by only 1% of the existing subreddits. These concentrated conflicts could be the cause of many cross-referencing hyperlinks between such 'aggressive' subreddits and make it likely for this dataset to indeed contain an unfair distribution of edges. Since the effect observed here is much stronger than seen during the generated experiments, including the experiment from Section 7.6, which included a very strong skew already, it seems that this real-life dataset may contain biases that are much stronger than anticipated.

8 Conclusion

From the results obtained in Section 7 a number of conclusions can be drawn. These conclusions are grouped per algorithm and presented in this section.

Random Warm-Up The random warm-up algorithm has shown to only be a good option when dealing with a rather small graph with a relatively short lifetime. If the graph grows too large or undergoes a large number of updates, the randomness of this approach will cause the total number of colors to be higher than necessary. Even so, it is known to theoretically be bounded by the maximum degree $\Delta + 1$, and if the number of recolors or running time is of most importance, there is no other algorithm that achieves a lower number of recolors or faster running time than this one.

Small- and Big-Bucket Algorithms These algorithms turned out to be the baseline, or middle of the road, when it comes to the trade-off between recolors and total number of colors used. The two complementary algorithms cover the entire range of the trade-off by using different parameters, as is evident from the figures in Section 7, and perform competitively in all except the most extreme cases. While not the most efficient solution for every case, these algorithms are a reasonable choice when looking for a middle ground between recolors and number of colors used, on top of which they also scale well with both graph size and update sequence length, performing just as well when either of these increases.

Static-Dynamic Algorithm The static-dynamic algorithm is somewhat of an exception in that the idea behind it is intuitively quite smart, but the execution causes its performance to be less than desirable. The multiplicative growth in number of colors used caused by using two separate color palettes that need to later be combined makes this algorithm the least attractive choice in many situations. It is therefore not recommended to use this algorithm in any of the cases considered in this work.

DC-Orient While DC-Orient was always expected to be the go-to algorithm in cases where the total number of colors used are most important, the experiments have shown a whole new property of this algorithm. Where normally DC-Orient would require many more recolors and time to run compared to other algorithms, this disadvantage seemingly disappears when ran on a highly skewed dataset. On datasets with unfair distribution of degrees or an ordered update sequence DC-Orient can compete with all other algorithms discussed in this work while still retaining the lowest number of colors used of them all.

Static-Simple The new static-simple algorithm has proven to be a strict improvement over the original static-dynamic algorithm. It manages to produce results with the same number of recolors, but a much lower number of total colors used. It also remains competitive with the small-bucket algorithm and the random warm-up, often outperforming the small-bucket algorithm and providing a reasonable trade-off in the range of low number of recolors. The static-simple algorithm does not manage to cover the same range as the big-bucket, DC-Orient or DC-Random algorithms when it comes to low number of colors, however. Additionally, the random element of this algorithm, that becomes especially important for higher parameter values can make the results for this algorithm quite unpredictable, which is something to consider when deciding to use static-simple.

DC-Random Possibly the most surprising and promising result of the experiments is the quality of the results produced by DC-Random. This simple combination of the random warm-up and DC-Orient is seemingly able to simulate either one of them, but is also able to produce competitive results anywhere in the range between them. Similarly to DC-Orient, DC-Random requires fewer recolors when the dataset is skewed somehow, but since the random-warm up algorithm displays no such behaviour, this effect is lessened with higher values for parameter p, in some cases, like the real-life Reddit dataset, it can thus occur that DC-Random with some parameter p is outperformed by the original DC-Orient. The biggest drawback of DC-Random remains its running time which, even though it considerably improves upon that of DC-Orient, is still not competitive with the alternative algorithms discussed in this work. This algorithm should thus only be used if the number of colors and recolors is sufficiently more important than the time it takes to run.

These conclusions indicate that, depending on the situation, almost all of the discussed algorithms are the best option in at least some cases. Aside from outlining what these cases are in order to improve decision making, we also introduced two new combination algorithms, proving that cooperation between multiple approaches can lead to results that surpass either approach by themselves. With this lesson and an extension to the available experimental data on the dynamic graph coloring problem, we hope this work will stimulate further research into this multi-faceted problem.

9 Future Work

Even though many experiments were performed over the course of the research for this work, there are many more imaginable. Apart from new combinations of properties or larger graphs and update sequences, future work may also consider running experiments on graphs with even more significantly skewed aspects such as uneven degree distribution or node focused update sequence ordering. These inputs have proven to considerably change the relative performance of the different algorithms discussed in this work, and it would be interesting to see how strong this effect can become, as well as finding practical applications in which such skewed graphs naturally occur. Additionally, the parameter ranges used in Section 6 could be extended by considering more extreme values for the parameters. More datapoints can also be computed for the Reddit dataset, allowing for further insight into the trade-off for these algorithms in a real-life application.

Another aspect that could prove interesting is the difference between measuring the average number of colors and recolors as compared to the maximum of these values. Since many of the algorithms perform an occasional reset step to reduce the number of colors used, the quality of the coloring for such an algorithm could vary quite significantly while still obtaining a reasonable average. Depending on the practical application this could be problematic behaviour. Future research could thus be conducted on which algorithms most clearly display this behaviour, how it could affect a practical application, and whether or not a trade-off may be possible between stability and quality.

Apart from these additional experiments, the combination algorithms presented in this thesis could also be extended upon. While both of the new algorithms introduced in this work are seemingly competitive with many of the original algorithms, no theoretical analysis has been performed to prove asymptotic bounds for either of the two algorithms. The conclusions in this work are merely based on an intuitive understanding of each algorithm and the experiments that were performed. As such, different methods of combining the original algorithms may exist that perform even better than those found here, or improvements upon these algorithms might exist that allow them to perform even more efficiently with only minor changes. These new algorithms, but especially the idea of combining existing algorithms, is therefore an interesting topic for further research.

References

[1] M. Garey, D. Johnson, and L. Stockmeyer, "Some simplified np-complete graph problems," *Theoretical Computer Science*, vol. 1, no. 3, 1976, pp. 237–267. doi: 10.1016/0304-3975(76)90059-1

[2] R. M. Karp, *Reducibility among Combinatorial Problems.* Boston, MA: Springer US, 1972, pp. 85–103. ISBN 978-1-4684-2001-2. doi: 10.1007/978-1-4684-2001-2_9

[3] L. Stockmeyer, "Planar 3-colorability is polynomial complete," *SIGACT News*, vol. 5, no. 3, jul 1973, p. 19–25. doi: 10.1145/1008293.1008294

[4] R. Lewis, J. Thompson, C. Mumford, and J. Gillard, "A wide-ranging computational comparison of high-performance graph colouring algorithms," *Computers & Operations Research*, vol. 39, no. 9, 2012, pp. 1933–1950. doi: 10.1016/j.cor.2011.08.010

[5] D. J. A. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, 01 1967, pp. 85–86. doi: 10.1093/comjnl/10.1.85

[6] S. Bhattacharya, D. Chakrabarty, M. Henzinger, and D. Nanongkai, "Dynamic algorithms for graph coloring," in *Proceedings of the 2018 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2018, pp. 1–20. doi: 10.1137/1.9781611975031.1

[7] L. Barba, J. Cardinal, M. Korman, S. Langerman, A. van Renssen, M. Roeloffzen, and S. Verdonschot, "Dynamic graph coloring," *Algorithmica*, vol. 81, no. 4, Apr. 2019, pp. 1319–1341. doi: 10.1007/s00453-018-0473-y

[8] S. Solomon and N. Wein, "Improved dynamic graph coloring," *ACM Trans. Algorithms*, vol. 16, no. 3, jun 2020. doi: 10.1145/3392724

[9] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang, "Effective and efficient dynamic graph coloring," *Proceedings of the VLDB Endowment*, vol. 11, 11 2017, pp. 338–351. doi: 10.14778/3157794.3157802

[10] J. Bossek, F. Neumann, P. Peng, and D. Sudholt, "Runtime analysis of randomized search heuristics for dynamic graph coloring," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1443–1451. ISBN 9781450361118. doi: 10.1145/3321707.3321792

[11] M. Theunis. (2022) Dynamic graph coloring thesisx. *GitHub repository.* Available: https://github.com/mtheunistue/DynamicGraphColoring

[12] S. Kumar, W. L. Hamilton, J. Leskovec, and D. Jurafsky, “Community interaction and conflict on the web,” in *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2018, pp. 933–943. doi: 10.1145/3178876.3186141

[13] A. Kosowski and K. Manuszewski, “Classical coloring of graphs,” *Graph Colorings*, 2004, p. 1–19. doi: 10.1090/conm/352/06369

[14] A. Hagberg, D. Schult, and P. Swart. Networkx reference release 2.8.4. *NetworkX*. Available: https://networkx.org/documentation/stable/_downloads/networkx_reference.pdf

A Comparison Tables

This appendix consists of experimental results obtained during preliminary experimentation on the various algorithms to find out which versions to focus on. The results in this appendix are provided in the form of tables in which the version of the algorithm, average number of recolors per update, average number of colors used in the coloring after each update and total time taken for all updates are provided as columns. Additionally, in the top left cell of each table, some more information is provided about the graph used in the preliminary experiment. The number of nodes, edges and average estimate for the chromatic number $\hat{C}$ after each update are given. The experiments provided here follow the same set-up as described in Section 6, we thus consider increasing graphs and stream experiments.

A.1 Static-Dynamic Version Comparison

In Tables 4 and 5 we provide a small comparison between the two different implementations for the static-dynamic algorithm: one with and one without resetting the dynamic graph whenever a full reset is called. The version without a dynamic reset is the one presented in [8] and the main version considered in this work.

Table 4: Comparison of the static-dynamic algorithm with and without dynamic reset on random graph and update sequence.

235 Nodes; 18528 Edges; $\hat{C}$ = 27.17	Average #Recolors	Average #Colors	Time Taken (s)
Static-Dynamic (l = 10) no dynamic reset	1.06	132.49	6.902
Static-Dynamic (l = 10) dynamic reset	1.19	108.14	6.929
Static-Dynamic (l = 100) no dynamic reset	0.18	163.82	6.158
Static-Dynamic (l = 100) dynamic reset	0.2	155.13	6.494

Table 5: Comparison of the static-dynamic algorithm with and without dynamic reset on a random update stream of length 10.000.

207 Nodes; 7315 Edges; $\hat{C}$ = 24.54	Average #Recolors	Average #Colors	Time Taken (s)
Static-Dynamic (l = 10) no dynamic reset	0.51	124.94	2.513
Static-Dynamic (l = 10) dynamic reset	0.56	108.34	2.814
Static-Dynamic (l = 100) no dynamic reset	0.11	157.7	2.421
Static-Dynamic (l = 100) dynamic reset	0.11	157.13	2.336

From these results it appears the dynamic reset variant provides a slight skew towards fewer total colors used and more recolors. While this could be an interesting trade-off, especially since the effect on number of colors used seems to be much larger, it is not significant enough to make this algorithm competitive with the others considered in this thesis. We therefore decide to use the original version as described in [8] in the rest of the work, in order to get a clearer comparison between the algorithms from the different papers.

A.2 DC-Orient Version Comparison

A comparison between the performance of the basic and optimized versions of DC-Orient can be found in Tables 6 and 7.

Table 6: Comparison of the basic and optimized versions of DC-Orient on a random graph and update sequence.

178 Nodes; 6332 Edges; $\hat{C} = 14.3$	Average #Recolors	Average #Colors	Time Taken (s)
DC-Orient without optimizations	17.22	14.3	108.322
DC-Orient with optimizations	17.22	14.3	21.71

Table 7: Comparison of the basic and optimized versions of DC-Orient on a random update stream of length 10.000.

225 Nodes; 13645 Edges; $\hat{C} = 40.06$	Average #Recolors	Average #Colors	Time Taken (s)
DC-Orient without optimizations	38.3	40.06	2073.542
DC-Orient with optimizations	38.3	40.06	172.858

As is apparent from these results, both versions of DC-Orient are identically effective when it comes to recolors and total number of colors used. The only difference between the two lies in the running time. Because of this, the optimized DC-Orient version is used in the rest of this work. It is also worth noting that indeed the number of colors used by DC-Orient are identical to those used by the static greedy approach used to approximate C, as is apparent from these values being identical in both experiments.

A.3 DC-Random Version Comparison

Tables 8 and 9 show the difference in performance between the basic and optimized versions of DC-Random.

Table 8: Comparison of the basic and optimized versions of DC-Random on a random graph and update sequence.

231 Nodes; 8383 Edges; $\hat{C} = 14.25$	Average #Recolors	Average #Colors	Time Taken (s)
Basic DC-Random ($p = 0.8$)	5.63	16.27	52.536
Optimized DC-Random ($p = 0.8$)	5.47	17.04	13.881
Basic DC-Random ($p = 0.998$)	0.2	30.93	6.145
Optimized DC-Random ($p = 0.998$)	0.25	30.03	5.307

A similar result as for DC-Orient can be observed here: the results of both versions are very similar while the running time of the optimized version is much lower, especially for the iteration with $p = 0.8$. The reason the results are not identical for the different versions is that the random warm-up element included in DC-Random causes an element of randomness to affect the final results, it is therefore highly unlikely that even the same algorithm produces the exact same result twice. We thus also use the optimized version of DC-Random in the main experiments.

A.4 Density Experiment Results

The increasing density experiment shows similar occurrences to the Small vs Large experiment from Section 7.3, albeit in a much more minimal manner. The visual plots have therefore only been added in this Appendix, in Figure 12 and 13 The static-dynamic number of colors still blow

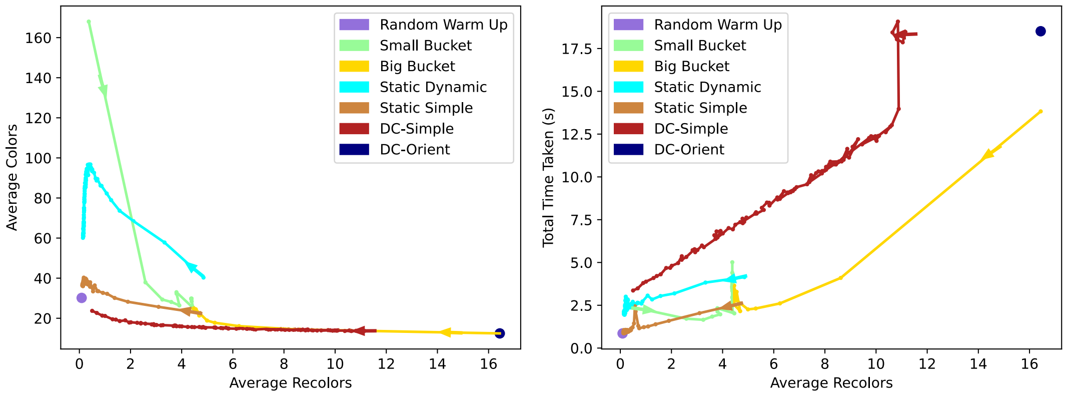

Figure 12: Results for a sparse graph with a random update sequence. 200 nodes, 5970 edges and estimated average $\hat{C} = 12.45$.

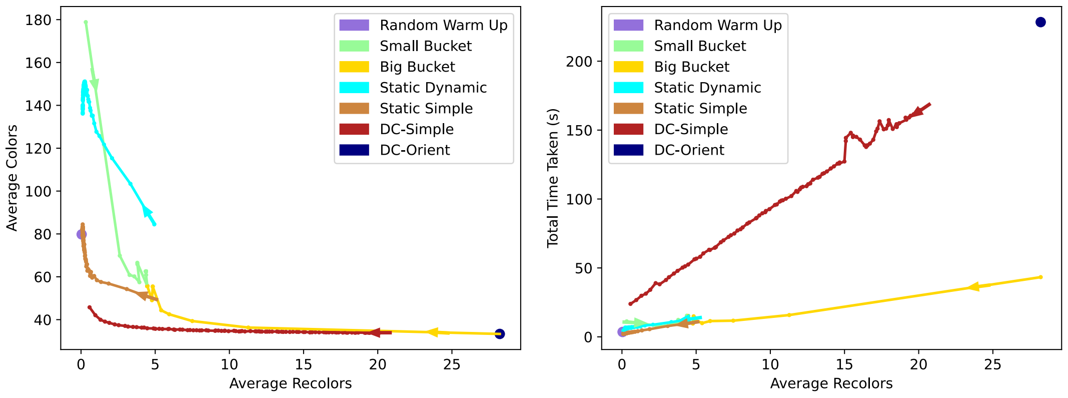

Figure 13: Results for a dense graph with a random update sequence. 200 nodes, 17910 edges and estimated average $\hat{C} = 33.33$.

Table 9: Comparison of the basic and optimized versions of DC-Random on a random update stream of length 10.000.

225 Nodes; 9148 Edges; $\hat{C} = 27.37$	Average #Recolors	Average #Colors	Time Taken (s)
Basic DC-Random ($p = 0.8$)	9.32	27.76	277.483
Optimized DC-Random ($p = 0.8$)	9.24	27.86	39.038
Basic DC-Random ($p = 0.998$)	0.26	38.2	17.299
Optimized DC-Random ($p = 0.998$)	0.22	42.37	11.105

up, but the random warm-up number of colors used increases at a similar rate to the estimate of $\hat{C}$, and the number of recolors for DC-Orient barely changes at all between the medium and large density tests. Additionally, the strength of the curve for the small- and big-bucket algorithms remains largely the same over the course of the different experiments, indicating that the density of the graph has little to no effect on its performance.

A.5 Parameter Comparison

Table 10 shows a comparison of the different parameters used while running the algorithms.

Table 10: The results of the algorithms ran using various parameters on the same graph and update sequence as Figure 4.

200 Nodes; 11940 Edges; $\hat{C} = 21.5$	Average #Recolors	Average #Colors	Time Taken (s)
Random Warm-Up	0.05	54.64	1.521
Small-Bucket algorithm ($d = 1$)	0.31	175.12	4.427
Small-Bucket algorithm ($d = 3$)	3.26	44.35	3.326
Small-Bucket algorithm ($d = 5$)	3.92	40.9	3.75
Small-Bucket algorithm ($d = 10$)	4.39	45.37	4.793
Small-Bucket algorithm ($d = 20$)	4.39	45.43	5.356
Big-Bucket algorithm ($d = 1$)	24.64	21.5	14.318
Big-Bucket algorithm ($d = 3$)	6.97	26.19	4.267
Big-Bucket algorithm ($d = 5$)	5.25	30.16	3.773
Big-Bucket algorithm ($d = 10$)	4.47	39.34	5.205
Big-Bucket algorithm ($d = 20$)	4.47	39.34	5.206
Static-Dynamic algorithm ($l = 1$)	4.9	64.88	5.885
Static-Dynamic algorithm ($l = 2$)	3.33	83.26	4.906
Static-Dynamic algorithm ($l = 5$)	1.77	100.34	4.086
Static-Dynamic algorithm ($l = 10$)	1.04	108.64	3.803
Static-Dynamic algorithm ($l = 50$)	0.29	130.59	3.274
Static-Dynamic algorithm ($l = 100$)	0.17	127.95	3.141
DC-Orient	24.64	21.5	83.747
Static-Simple algorithm ($l = 1$)	4.76	34.88	5.76
Static-Simple algorithm ($l = 2$)	3.09	38.33	4.349
Static-Simple algorithm ($l = 5$)	1.56	42.32	3.052
Static-Simple algorithm ($l = 10$)	0.89	45.55	2.453
Static-Simple algorithm ($l = 50$)	0.24	56.7	1.674
Static-Simple algorithm ($l = 100$)	0.14	60.37	1.629
DC-Random ($p = 0.2$)	20.22	21.96	71.565
DC-Random ($p = 0.5$)	14.23	22.38	54.849
DC-Random ($p = 0.8$)	6.84	23.63	31.353
DC-Random ($p = 0.998$)	0.31	40.33	10.059
DC-Random ($p = 0.9999$)	0.06	54.5	8.768

B Experiment data generation

B.1 Constant Update Stream

The constant update stream experiment makes use of two types of update streams. The random update stream is simple to implement and merely picks a random edge to add or remove. For the update sequence in which older edges are more likely to be removed, which we call the decaying update stream, generating it is a bit more involved. Initially each edge in the graph has the same probability to be removed once an edge removal step occurs. To signify this, all edges are assigned a weight of 1. With each update that is performed, all edges in the graph increase their weight by 1. As such, older edges will have a higher weight. When a new edge is added to the graph, its weight is initialized at 1, making it much less likely to be removed during an edge removal step than those already present in the graph for longer. When an edge removal step needs to occur in order to keep the number of edges in the graph stable, a weighted random selection is done, selecting one edge from the set of edges currently present in the graph, based on their current weight.

B.2 Degree Variation

For the light and heavily skewed degree variation experiments, two graphs are generated in which edges are unevenly distributed. To achieve this, we use a process that makes edges adjacent to some nodes more likely to occur in the initial graph. A process which we call node prioritization. This prioritization is achieved by assigning each node a priority between 0 and 1 during the graph generation phase. Whenever an edge is considered for being part of the initial graph the edge priority is obtained by combining the node priorities of the two adjacent nodes. This edge priority indicates the probability it is indeed added to the graph, decided by a weighted coin toss. If the result of this coin toss is negative, a new edge, which could potentially be the same one, is selected for consideration until the target number of edges is obtained. The distribution of these edge priorities thus influences how skewed the degrees in the initial graph become. If the edge priorities are fairly similar, only a light skew occurs, whereas if edge priorities differ more, a heavier skew will occur in the resulting graph. To achieve both these options we combine the node priorities into edge priorities in two different ways. In order to get a light skew we combine node priorities into edge priorities by taking their average, whereas in order to obtain a heavy skew we combine node priorities by squaring them both and multiplying them together. This square and multiplication enlarges the difference between the original node priorities and thus cause the edge priorities to be further apart. As such, we obtain both a graph with lightly skewed degree distribution and one with heavily skewed degree distribution.

B.3 Update Spread

The update spread experiment makes use of an expanding and node prioritized order. The expanding order is achieved by initially assigning each node a weight value of 1. The first edge in the update sequence is then selected by a weighted random sample using calculated edge weights, which are obtained by summing the node weights of the edge's endpoints. The nodes adjacent to the selected edge increase their weight by 1 before selecting the edge to be added next in the update sequence. This makes edges adjacent to a node that already has an edge in the update sequence more likely to be added sooner, but does not have a strong enough effect as to focus edge additions around a specific node. This process continues until all edge addition updates are ordered and provides us with an order reminiscent of a breadth-first-search approach.

The node prioritized update sequence is generated by assigning each node a priority between 1 and 1000, after which these priorities are taken to the power of some parameter x. This parameter can be adjusted to increase or decrease the strength of the prioritization. As found in preliminary experiments, however, this approach requires the parameter to be quite large, only clearly showing the intended effect of focusing mostly on one node at a time when using a value of 100. The result of this computation is thus that the priorities of the nodes are very spread out. After this computation each node has a set priority and edges are selected as follows: first a node is selected using a weighted random sample using the node priorities. From the set of available edge additions, a list is created of edges that are adjacent to this selected node, and from this list one edge is selected at random. This edge is put first in the update order and the whole process is repeated for the next position. If a node has no more adjacent edges in the available edge update set, it is removed from the random sample pool. This manner of ordering the edges is very likely to first add all available edges adjacent to the node with the highest priority and after that continue with adding all available edges that are adjacent to the node with the second highest priority, and so on. The ordering resulting from this method is thus grouped strongly.

C Pseudocode

In this appendix, the pseudocode for the small-bucket algorithm, the big-bucket algorithm and the static-dynamic algorithm are provided. This pseudocode represents the manner in which the algorithms have been implemented during this thesis and may thus vary slightly from the algorithms as described in the original papers [7] [8]. The textual description of the small- and big-bucket algorithms can be found in Section 4.3 and the description of the static-dynamic algorithm in Section 4.4.

Algorithm 1 Small-Bucket Algorithm

```
**Parameters:** $d$, $G$
Initialization:
resetBuckets($G$)
**Return**;

Update graph:
**if** Edge or Vertex removed **then**:
    Update all relevant subgraphs without changing colors
**if** Vertex added **then**:
    Add vertex to an empty bucket $b$ on the first level
    updateBuckets($b$);
**if** Edge added **then**:
    Choose one of the endpoints of the edge to be $v_e$
    Remove $v_e$ from whichever bucket it is in and add it to an empty bucket $b$ on the first level
    updateBuckets($b$);

updateBuckets($b$):
$i := 0$;
**while** $i < d$ **do**
    **if** Still an empty bucket at level $i$ **then**:
        Let $b_g$ denote the subgraph of the nodes in $b$
        staticColoring($b_g$);
        **Return**;
    **else**:
        Empty all level $i$ buckets into a single level $i$+1 bucket, update $b$ to point at the new bucket
        $i$++;
resetBuckets($G$);
**Return**;

resetBuckets($G$):
$N_R$ := number of vertices in $G$
$s := N_R^{1/d}$
Create $d$ levels of $s$ buckets each, having $s^i$ capacity for level $i$, starting at 0
Create a final level $d$ with a single bucket without a limit on capacity
staticColoring($G$);
**Return**;

staticColoring($g$):
Use the available static graph coloring algorithm to color subgraph $g$
**Return**;
```

Algorithm 2 Big-Bucket Algorithm

```
**Parameters:** $d$, $G$
Initialization:
resetBuckets($G$)
**Return**;

Update graph:
**if** Edge or Vertex removed **then**:
    Update all relevant subgraph without changing colors
**if** Vertex added **then**:
    Add vertex to bucket $b$ on the first level
    updateBuckets($b$);
**if** Edge added **then**:
    Choose one of the endpoints of the edge to be $v_e$
    Remove $v_e$ from whichever bucket it is in and add it to bucket $b$ on the first level
    updateBuckets($b$);

updateBuckets($b$): $i := 1$;
**while** $i < d + 1$ **do**
    **if** Less than or equal to $s^i - s^{i-1}$ vertices in $b$ **then**:
        Let $b_g$ denote the subgraph of the nodes in $b$
        staticColoring($b_g$);
        **Return**;
    **else**:
        Empty $b$ into the level $i$+1 bucket, update $b$ to point at the new bucket
        $i$++;
resetBuckets($G$);
**Return**;

resetBuckets($G$):
$N_R$ := number of vertices in $G$
$s := N_R^{1/d}$
Create $d$ levels of a single bucket each, having $s^i$ capacity for level $i$, starting at 1
Create a final level $d + 1$ with a single bucket without a limit on capacity
staticColoring($G$);
**Return**;

staticColoring($g$):
Use the available static graph coloring algorithm to color subgraph $g$
**Return**;
```

Algorithm 3 Static-Dynamic Algorithm for General Graphs

```
 1: Parameters: $l$, $G$
 2: Initialize:
 3: let $n$ be the number of vertices in $G$
 4: $c := 0$; Counter variable
 5: staticBlackBox($G$, 0);
 6: let $G'$ be $G$ without edges
 7: Initialize all colors in $G'$ as 0
 8: let $r_0 ... r_{\log n}$ be an empty set of nodes
 9: let R($r$) be a subgraph of $G$ depending on a selection of nodes $r$
10: Activate levels 0..$\log n$;
11:
12: Update graph:
13: if Edge or Vertex removed then:
14:     Update $G$ and $G'$ without changing colors
15: if Vertex added then:
16:     Add vertex to $G$ and $G'$
17:     Assign new vertex arbitrary colors $c_1$ and $c_2$
18: if Edge added then:
19:     Let $e$ be the added edge
20:     Increase recent degree of endpoints by 1
21:     Add edge to $G$ and $G'$
22:     Add node with the highest recent degree to $r_{level}$ for each active level
23: $c := c + 1 \bmod l$;
24: if $c \bmod l == 0$ then:
25:     Set $level$ to be the highest of the active levels
26:     if $level == 0$ then:
27:         staticBlackBox($G$, $level$)
28:     else
29:         staticBlackBox(R($r_{level}$), $level$);
30:         Deactivate $level$
31:     Activate all levels higher than $level$
32:     Return;
33: if Edge added then:
34:     if $e$ still in $G'$ then:
35:         dynamicBlackBox($G'$, $e$);
36: Return;
37:
38: staticBlackBox($g$, $level$):
39: Compute valid coloring for $g$ using the colors from $level$
40: Assign vertices in $g$ the computed color $c_1$
41: Reset recent degree for all vertices in $g$ to 0
42: Remove all edges adjacent to vertices in $g$ from $G'$
43: Remove all nodes from $r_{level}$
44: Return;
45:
46: dynamicBlackBox($g$, $e$):
47: Compute valid coloring for nodes adjacent to $e$ in $g$
48: Assign vertices in $g$ the computed color $c_2$
49: Return;
```