Jagiellonian University
Department of Theoretical Computer Science

Zofia Glapa

Implementation of exact enumeration algorithms for vertex coloring

Bachelor Thesis

Supervisor: dr hab. Krzysztof Turowski

July 2023

List of definitions

List of algorithms

Contents

Chapter 1

Introduction

Graph coloring is a frequently studied problem of graph theory and it is also one of the oldest in this field. Its beginnings can be traced back to the 1850s when Francis Guthrie, a mathematics student, made an observation while studying the administrative map of England. He noticed that it was possible to color the counties on the map using only four colors in such a way that no two adjacent counties shared the same color. In 1879 the Four Color Problem, stating that any map on the plane can be colored with 4 colors, was proposed. Later that year Kempe [Kem79] gave the first proof, in which he designed a method of obtaining such coloring. This was proven to be false in 1890 by Heawood [Hea90], who however managed to prove the problem for five colors using Kempe's technique. The concept of the chromatic polynomial was introduced in 1912 by Birkhoff [Bir12] in an unsuccessful attempt to solve the problem. FCP was finally proved in 1976 by Appel and Hanken [AH76] . That proof was one of the first to strongly rely on computer assistance. Later more proofs for the problem were presented, requiring less computation like that by Seymour and Robertson [Rob+96]. Meanwhile, the general problem became more popular with Brooks [Bro41] bounding the chromatic number by the degree of the graph and Grotzsch [Gro59] showing that every planar graph not containing a triangle is 3-colorable.

The chromatic number problem was shown to be NP-complete by Karp [Kar10] in 1972 by a reduction from the boolean satisfiability problem, which was earlier proved to be NP-complete by Cook [Coo71] .

Over the years various practical applications were discovered for graph coloring. One of them is register allocation in compilers – i.e. the problem of optimally assigning variables used in a program to processor registers and managing the necessary data transfers between them. The first graph coloring approach was proposed by Chaitin et al. [Cha+81] in 1981. The idea was to represent the variables as vertices in a graph, creating an edge between each pair that need to live simultaneously during the program's execution. This was later improved by Briggs et al. [BCT94] in 1992 with the introduction of conservative coalescing and biased coloring.

Graph coloring also becomes useful in various scheduling problems such as timetabling (de Werra [Wer85]), assigning frequencies to mobile radio telephones (Roberts [Rob06]), printed circuit testing (Garey, Johnson [GJS76]) or pattern matching (Ogawa [Oga86]).

1.1 Basic graph definitions

Graph definitions as stated by Dietzel in Graph Theory [3rd edition] [Die05]

Definition 1.1 (graph). *A graph is a pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$; thus, the elements of E are 2-element subsets of V. The elements of V are the vertices of the graph G, and the elements of E are its edges.*

Definition 1.2 (subgraph). *If for two graphs $G = (V, E)$ and $G' = (V', E')$ $V' \subseteq V$ and $E' \subseteq E$ then G' is a subgraph of G, written as $G' \subseteq G$.*

Definition 1.3 (induced subgraph). *If for two graphs $G = (V, E)$ and $G' = (V', E')$ $G' \subseteq G$ and G' contains all the edges $xy \in E$ with $x, y \in V'$ then G' is an induced subgraph of G.*

Definition 1.4 (complete graph). *If all the vertices of G are pairwise adjacent, then G is complete. A complete graph on n vertices is a K^n.*

Definition 1.5 (clique). *A maximal complete subgraph of a graph is a clique.*

Definition 1.6 (vertex coloring). *A vertex coloring of a graph $G = (V, E)$ is map $c : V \to S$ such that $c(v) \neq c(w)$ whenever v and w are adjacent.*

Definition 1.7 (chromatic number). *The chromatic number of a graph $G = (V, E)$ denoted by $\chi(G)$ is the smallest integer k such that G has a k-coloring, a vertex coloring $c : V \to \{1, \ldots, k\}$.*

1.2 Different approaches to solving graph coloring related problems

Two main types of the problem can be recognized – the *decision* problem of finding whether a graph can be k-colorable or the *optimization* problem of finding the chromatic number. We may describe the ideal graph coloring algorithm as one that finds an exact solution for all graphs in a reasonable amount of time – polynomial in the graph's size. Since the problem is NP-hard we suppose that it does not exist and as such one of these requirements has to be broken to give a working algorithm.

Breaking the requirement for finding the exact solution yields an approximate algorithm. The approximate algorithms for optimization problems are polynomial time algorithms that produce a solution whose value is within a certain factor of the value of an optimal one. For graph coloring the performance guarantee is the worst-case ratio between the actual number of colors used by

the algorithm and the chromatic number of the graph [WS11]. In 1974 Johnson [Joh73] presented an algorithm that achieved the performance guarantee of $O(n/\log n)$. This result was later improved by Widgerson [Wig83] , who achieved the guarantee of $O(n(\log\log n)^2/(\log n)^2)$. The most recent improvement by Halldorsson [Hal15] achieved $O(n(\log\log n)^2/(\log n)^3)$. In 1995 Bellare et al. [BGS98] showed that for any arbitrary $\epsilon > 0$, there is no polynomial time approximation algorithm with the factor of $n^{\frac{1}{5}-\epsilon}$

The parameterized approximation algorithms allow relaxation on the time complexity – these algorithms produce an approximate solution to optimization problems in polynomial time in the input size and a function of a given parameter. The problems which require exponential time in the size of the parameter and polynomial in the size of the input are called fixed-parameter tractable, as they can be solved for small parameter values. Since *3-coloring* is known to be NP-complete, under the assumption that P = NP the graph coloring problem parametrized by the number of colors k cannot be in FPT (and even in XP) because otherwise there would exist an algorithm running in time $f(k)n^{g(k)}$ for $k = 3$. However, when parametrized by the treewidth of the graph the problem of vertex coloring is known to be W[1]-hard [Cyg+15].

However, if the exact solution is needed then we must fall back to exponential time algorithms. The dynamic programming algorithm presented in 1976 by Lawler [Law76] finds the optimal solution in $O(2.443^n)$. This was later improved in 2003 by Eppstein [Epp02] to $O(2.4150^n)$ and in 2004 by Byskov [Bys04] to (2.4023^n). In 2008 Björklund and Husfeldt [BH08] improved these results using fast matrix multiplication achieving the complexity of $O(2.2326^n)$. . The current best result of $O^*(2^n)$ was presented by Björklund et al. [BHK09] using the inclusion-exclusion principle and fast zeta transform. These algorithms however use exponential space which is not ideal in the real-world scenario, where although time can be spared the space is obviously limited. Therefore exact polynomial space algorithms are sometimes preferred, even though they come at a cost of increased time complexity. One of the first polynomial space algorithms was described by Zykov [Zyk49] in 1949 with the time complexity of $O(2^{n^2})$. In 1972 the idea of coloring with branch & bound algorithms was presented by Brown [Bro72] with time complexity of $O^*(n!)$. This algorithm was later modified by Christofides [Chr75], Brelaz [Bre+79], and Korman [Kor79]. In 2006 Bodleander and Kratsch [BK06] gave an $O(5.283^n)$ dynamic programming algorithm. The current best polynomial space results are achieved using the inclusion-exclusion principle. Bjorklund and Husfeldt [BH06] presented a method of converting an $O^*(x^n)$ algorithm for counting independent sets into $O^*((1+x)^n)$ time and polynomial space algorithm for vertex coloring by using the inclusion-exclusion principle. Recently, Gaspers and Lee [GL23] published an $O^*(1.2356^n)$ algorithm for the #IS problem yielding an $O^*(2.2356^n)$ time and polynomial space algorithm for graph coloring.

The main focus of this paper is the branch & bound approach. The four algorithms mentioned earlier are presented in detail in the next section. The implementations are based on the work of Kubale and Jackowski [KJ85] who

gathered the above and gave a general frame algorithm for finding the optimal coloring.

Chapter 2

Enumeration algorithms

In this section, we will describe a family of enumeration algorithms (also called branch and bound algorithms) that are used to solve graph coloring problems. These algorithms explore all possible solutions to find the optimal one. We will then cover four such algorithms in detail.

2.1 General idea

Let's imagine a depth-first search like traversal where the nodes are partial colorings and the leaves are total colorings of a given graph G. We will limit the tree only to contain proper colorings on its nodes, i.e. the algorithm will not follow the path that would lead to an invalid solution. Let $\{0, \ldots, n-1\}$ be a set of vertices and $\{1, \ldots, n\}$ be the set of all possible colors. The algorithm starts at the solution tree's root – a node with the vertex 0 given the color 1. Then at the ith level in the tree, the node will contain a partial solution up to the ith vertex. To decide which nodes should be created and/or traversed next, the algorithm will determine the set of *feasible colors* – denoted FC – values from the color set that can be assigned to the $(i+1)$th vertex without conflicting with the partial solution. The suitable edges and nodes will be created and the algorithm will follow by choosing the smallest colored edge and going down that branch. When the algorithm reaches the leaf it will compare the solution at the leaf to the best one so far and replace it in case of finding a better one. In that way, all valid colorings will be found and the best one will be returned. While this approach requires traversing the whole solution tree there are some improvements that can be made to omit some branches:

- the number of colored edges going down from a node can be decreased to contain only edges up to the current maximum color $m + 1$, since if the solution would need the color $m + 2$ while still using the $m + 1$ further in the path, then we could swap them,
- since we are only interested in the best solution then after reaching the leaf

node we can go back in the tree to the point where the current maximum color m was used and only consider the nodes before that one in the current path as only these may influence that vertex's color.

This description has a few gaps and the way how following algorithms fill them is what differentiates them. For each one, we will define four crucial features:

1. Initialization – lower and upper bounds on the chromatic number and the initial order of the vertices
2. Rearrangement – whether and how the vertices are rearranged in the course of the algorithm
3. Determining feasible colors for vertices – how to decide which edges should go down from the current node in the tree
4. Deciding which subtrees can be skipped in the backtrack because they would not produce a better solution

2.2 Brown's algorithm

The simplest algorithm, most similar to the idea presented above contains 2 main parts: the forward step (going down the solution tree) and the backward step (backtracking) which interchange in the main loop. The run procedure initializes the ordering on the vertices and sets trivial bounds on the chromatic number. This algorithm uses the Greedy Largest First ordering on the vertices, namely for ith in the ordering it has more neighbors in the set $\{0, \ldots, i-1\}$ than any of the vertices from $\{i+1, \ldots, n-1\}$. The vertex with the maximal degree is chosen as the first one and ties are broken by checking for a higher degree. Then, in repeat, the algorithm executes the forward step while searching for a total coloring. After the forward step, if the solution has been found the backtrack starting point is set to the first vertex with the maximal color, the best solution is replaced, and the upper bound is lowered to the highest used color. When the execution ends early and the solution has not been found, the starting point is set to the last seen vertex. Then the backward step is executed which corresponds to backtracking through the solution tree in pursuit of a possibly better solution. After the backtrack finds a possible resumption point of the Forwards procedure – a vertex that can be recolored while still keeping the upper bound – it returns execution back to the forward step. The algorithm ends when either the upper bound matches the lower bound or the backtrack reaches the first vertex completing the traversal.

Algorithm 2.2.1 (Brown's algorithm)

```
   Global variables:
   lb, ub: lower and upper bounds
 1: procedure RUN(G)
 2:     ordering ← GreedyLargestFirst(G)
 3:     lb ← 1
 4:     ub ← G.size
 5:     r ← 0
 6:     while true do
 7:         Forwards(G, r)
 8:         if ub = lb then
 9:             break
10:         end if
11:         Backwards(G, r)
12:         if r = 0 then
13:             break
14:         end if
15:     end while
16: end procedure
```

The Forwards procedure initiated at the rth vertex loops through all remaining vertices, updates their feasible colors, and assigns them the smallest possible color. If some vertex cannot be assigned any color, then the execution stops and it is returned to be used as an entry point for backtracking. After the algorithm successfully colors all vertices the new solution is saved as the best one and the upper bound is updated. The vertex which was the first to reach maximum color is returned for Backwards.

```
   Global variables:
   n: number of vertices
   lb, ub: lower and upper bounds
   CS: current solution
   BS: best solution
   FC: map of feasible colors sets
 1: procedure Forwards(G, r)
 2:     for i ← r to n − 1 do
 3:         FC[i] ← determineFeasibleColors(i)
 4:         if FC[i] = ∅ then
 5:             r ← i
 6:             return
 7:         end if
 8:         CS[i] ← FC[i].begin
 9:     end for
10:     ub ← getMaxColor(CS)
```

```
11:  |  r ← getMaxColorNode(CS)
12:  |  BS ← CS
13: end procedure
```

The Backwards procedure iterates back through CP vertices, checks whether it's possible to assign them a different color, and if so proceeds to Forwards with that color. If not then it continues with the next CP vertex. Since recoloring the first vertex will not make a difference, r is assigned 0 after finishing the loop to signal that the algorithm has ended.

```
    Global variables:
    CS: current solution
    FC: map of feasible colors sets
 1: procedure Backwards(r)
 2:  |  CP ← determineCurrentPredeccessors(r)
 3:  |  while CP ≠ ∅ do
 4:  |  |  i ← CP.front()
 5:  |  |  FC[i].erase(CS[i])
 6:  |  |  if FC[i] ≠ ∅ and FC[i].begin() < ub then
 7:  |  |  |  r ← i
 8:  |  |  |  return
 9:  |  |  end if
10:  |  end while
11:  |  r ← 0
12: end procedure
```

The GreedyLargestFirst procedure finds the GLF ordering on the vertices. It starts with adding the maximum degree vertex to the ordering, then it creates and maintains a queue of yet uncolored vertices sorted decreasingly by the number of already ordered neighbors with ties broken by taking the one with a larger degree. Subsequent vertices from the queue are added to the ordering, updating the remaining vertices' neighbors count after each pop.

```
 1: procedure GreedyLargestFirst(G)
 2:  |  ordering ← []
 3:  |  m ← GetMaxDegreeNode(G)
 4:  |  Q ← Queue()     \\ sorted decreasingly by (number of already ordered
    neighbors, degree) pair
 5:  |  for each u ∈ m.GetNeighbors() do
 6:  |  |  Q.push(u)
 7:  |  end for
 8:  |  while Q ≠ ∅ do
 9:  |  |  v ← Q.pop()
10:  |  |  ordering.push(v)
11:  |  |  for each u ∈ v.GetNeighbors() do
12:  |  |  |  u.NeighborsInOrdering++
```

```
13:     |   end for
14:     | end while
15:     | return ordering
16: end procedure
```

The set of feasible colors for each vertex is determined in the following way: $FC[i] = \{1 \dots \min(C+1, ub-1)\} \setminus NC[i]$, where C is the maximum color used in the coloring so far and $NC[i]$ is the set of colors of the neighbors of i.

```
    Global variables:
    lb, ub: lower and upper bounds
    CS: current solution
 1: procedure DetermineFeasibleColors(i)
 2:     | FC ← {}
 3:     | C ← getMaxColor(CS)
 4:     | for c = 1 to min(C + 1, ub − 1) do
 5:     |     | FC.push(c)
 6:     | end for
 7:     | for each u ∈ i.neighbors do
 8:     |     | FC.erase(CS[u])
 9:     | end for
10:     | return FC
11: end procedure
```

The set of current predecessors is the set of all vertices appearing in the order before the current entry point r.

```
 1: procedure DetermineCurrentPredeccessors(r)
 2:     | CP ← Queue()
 3:     | for i = 1 to r − 1 do
 4:     |     | CP.push(i)
 5:     | end for
 6:     | return CP
 7: end procedure
```

2.2.1 Proof of correctness

The algorithm always terminates

The main loop alternates between the Forwards and the Backwards procedures while traversing the solution tree. It follows down on a path to the leaf in Forwards and then backtracks to another path in Backwards. Since it never goes back to a previously explored branch the algorithm terminates when it reaches the root, finding no other possible recoloring of the second vertex.

The algorithm finds the best solution possible

Let's define a simplified Brown's algorithm as an algorithm that does not perform any checks on the lower or upper bound, and also does not perform the reorder at the start, but is otherwise the same as shown before. We state the following:

Theorem 2.2.1. *The simplified Brown's algorithm enumerates all proper colorings of the graph of n vertices.*

Proof. If $|V(G)| = 1$ then there is only one possible coloring, and it is found correctly by the algorithm.

Let G be a graph of $k > 1$ vertices. Let v be a vertex in G such that it appears last in the order. Let G' be a subgraph of G without the vertex v and all its incident edges. G' is a graph of $k - 1$ vertices and by induction, the algorithm enumerates all proper colorings of this subgraph. For each of the colorings found for G', we can extend it by checking all feasible colors for v in reference to the coloring of its neighbor and assigning one of them to it, i.e. performing another iteration of the loop in the FORWARDS procedure. Therefore the algorithm enumerates all proper colorings of G. □

Additionally, if Brown's algorithm works for any graph with any order of vertices, it will also work for a graph with GREEDYLARGESTFIRST ordering. Now let's prove that this simplification does not change the returned solution.

Theorem 2.2.2. *Checking for lower or upper bounds only eliminates solutions that either use more colors or are lexicographically larger (in reference to GLF order) than the ones found before checking lower and upper bounds.*

Proof. Since the algorithm always chooses the first color from the set of feasible colors, it is guaranteed to be lexicographically first with that number of colors used when a solution is found. Therefore any solution that may be found after it would have to be larger. Since the upper bound check only fails if the color to be assigned at this moment reaches or is larger than the current solution's largest color, then every coloring found after that assignment would be worse than the current one. Therefore at every moment if a better solution exists then we can be sure that it is not in an omitted subtree. □

By combining these two theorems we know that during its execution the algorithm finds a subset of all solutions that must include the best one, and given that it only substitutes the best solution if the new one has fewer colors than the previous one then the following is true.

Corollary 2.2.2.1. *Brown's algorithm correctly finds the lexicographically first proper coloring of the graph.*

2.3 Christofides algorithm

The Christofides modification to Brown's algorithm reduces the number of necessary backtracks by limiting the CP set to only include vertices laying on a monotonic path ending in r. A path is considered to be *monotonic* when it passes through increasingly indexed vertices. The *current predecessors* set is now defined as follows:

Let the predecessor set for v_i be defined as

$$P(i) = \{j < i\colon \text{ there exists a monotonic path from } v_j \text{ to } v_i\}$$

We now define current predecessors as $CP = CP \cup P(r)$ for any vertex v_r that is passed as an argument to the BACKWARDS procedure.

The new DETERMINECURRENTPREDECESSORS method will now only add the vertex to the CP set if it lies on a monotonic path ending in r. The monotonic paths between every 2 vertices can be precomputed before the actual coloring algorithm by calculating the transitive closure of the graph.

Algorithm 2.3.1 (Christofides algorithm)

```
1: procedure DetermineCurrentPredeccessors(r)
2:     CP ← Queue()
3:     for i = 1 to r − 1 do
4:         if TransitiveClosure[i][r] then
5:             CP.push(i)
6:         end if
7:     end for
8: end procedure
   Global variables:
   G: graph
1: procedure CalculateTransitiveClosure(r)
2:     for each (u, v) ∈ G.edges do
3:         TransitiveClosure[u][v] ← true
4:     end for
5:     for u = 0 to n − 1 do
6:         for v = 0 to n − 1 do
7:             if TransitiveClosure[u][v] then
8:                 for w = 0 to n − 1 do
9:                     if TransitiveClosure[v][w] then
10:                        TransitiveClosure[u][v] ← true
11:                    end if
12:                end for
13:            end if
14:        end for
```

15: | **end for**
16: **end procedure**

The rest of the algorithm remains the same as before.

2.3.1 Proof of correctness

In the previous algorithm, we showed that the enumeration approach always finds the best possible solution. The only change made was to limit the number of backtracks performed by the algorithm by restricting the CP set. Now, we just need to prove that we cannot find a better solution by starting the forward step from the omitted vertices.

Theorem 2.3.1. *Whenever a better solution is found after backtracking, the vertex whose color change triggered the forward step lies on a monotonic path ending in the previously largest colored vertex.*

Proof. Let v be the triggering vertex. Since r has the largest color its color has to get smaller to obtain a better solution. But for any vertex u its color can only be changed if the color of a smaller indexed neighbor changed in the previous step. (Otherwise, the algorithm would still have chosen the smallest available color). Since that goes for every vertex but v it is easy to notice that these changes happen along some path – more precisely a path starting in v and ending in u. Since the algorithm traverses the vertices in increasing order, such a path is monotonic. □

2.4 Brelaz algorithm

Brelaz brings two major modifications to the base algorithm

1. A new algorithm – Saturation Largest First with Interchange (SLFI) – is used for setting the initial ordering and bounds on the chromatic number
2. The CP set is reduced to limit the number of backtracks by focusing on the smallest indexed predecessors for every color.

2.4.1 Saturation Largest First with Interchange

The SLFI algorithm finds the saturation largest first ordering on the vertices and computes a base solution that determines the initial bounds. We define *saturation degree* of a vertex as the number of its distinctly colored neighbors. During the execution, the algorithm creates and maintains a queue of yet unordered vertices sorted decreasingly by saturation degree with ties broken by choosing a larger degree. At the same time, subsequent vertices are ordered and colored using a simple heuristic known from the FORWARDS procedure – for each vertex, a set of forbidden colors (the ones used by its neighbors) is created and the smallest possible color is chosen. The interchange mechanism is applied if a new color needs to be used for the first time. For every pair (α, β)

of neighbors' colors, a subgraph containing only that colored vertices is created and then divided into connected components. If all of the components neighbor the current vertex with vertices colored with at most one of $\{\alpha, \beta\}$ then we can flip the colors in the component neighboring with α thus making the current vertex free of α-colored neighbors. That interchange allows the current vertex to be given α – preventing from unnecessarily increasing the number of colors. If no possible interchange is found the current vertex is assigned a new color. Reusing any color for the first time means that all vertices up to that point were connected forming a clique, so when it happens the initial clique size is set to the number of colors used so far. When all vertices are ordered and colored the upper bound is set to the number of colors used and the lower bound is set to the initial clique size.

Algorithm 2.4.1 (Saturation Largest First with Interchange algorithm)

```
   Global variables:
   CS: current solution
 1: procedure SLFI
 2:     maxColor ← 1
 3:     Q ← Queue()        \\ sorted decreasingly by (saturation degree, vertex
   degree) pair
 4:     m ← GetMaxDegreeNode(G)
 5:     CS[m] ← 1
 6:     ordering.push(m)
 7:     for each u ≠ m ∈ G.nodes do
 8:         Q.push(u)
 9:     end for
10:     while Q ≠ ∅ do
11:         v ← Q.pop
12:         ordering.push(v)
13:         c ← GetFirstValidColor(v)
14:         if c ≤ maxColor then
15:             CS[v] ← c
16:             if IC = 0 then
17:                 IC ← maxColor
18:             end if
19:         else
20:             for each w ∈ G.neighbors(v) do
21:                 K.insert(CS[w])
22:             end for
23:             if Interchange(K, v, CS) then
24:                 if IC = 0 then
25:                     IC ← maxColor
26:                 end if
27:             else
28:                 CS[v] ← c
```

```
29:  |  |  |  |  maxColor++
30:  |  |  |  end if
31:  |  |  end if
32:  |  end while
33: end procedure
```

The INTERCHANGE procedure tries to perform a color interchange for each pair of colors of current vertex v neighbors. If possible it applies the flip where necessary and colors the current vertex with α. If no possible interchange was found it returns false to indicate that a new color will be needed.

```
1: procedure INTERCHANGE(K, v, CS)
2:  |  for each (α, β) ∈ K × K do
3:  |  |  G' ← GETSUBGRAPHONCOLORS(G, α, β)
4:  |  |  verticesToRecolor ← ISINTERCHANGEABLE(G', CS, v, α)
5:  |  |  if verticesToRecolor.SIZE() > 0 then
6:  |  |  |  for each u ∈ verticesToRecolor do
7:  |  |  |  |  FLIPCOLOR(u)
8:  |  |  |  end for
9:  |  |  |  CS[v] ← α
10:  |  |  |  return true
11:  |  |  end if
12:  |  end for
13:  |  return false
14: end procedure
```

The ISINTERCHANGEABLE procedure performs the BFS dividing the graph into connected components. If it finds a component in which two vertices of different colors are connected with the current vertex v (making it impossible to apply interchange) it stops the search early. Otherwise, all components connected with v by α are bundled and returned to be recolored by the calling method.

```
1: procedure ISINTERCHANGEABLE(G', CS, v, α)
2:  |  neighborColor ← −1
3:  |  verticesToRecolor ← {}
4:  |  for each u ∈ G'.NODES() do
5:  |  |  Q ← QUEUE()
6:  |  |  if not visited[u] then
7:  |  |  |  Q.PUSH(u)
8:  |  |  |  while Q ≠ ∅ do
9:  |  |  |  |  w ← Q.POP()
10:  |  |  |  |  if G.HASEDGE((v, w)) then
11:  |  |  |  |  |  if neighborColor = −1  or neighborColor = CS[w] then
12:  |  |  |  |  |  |  neighborColor ← CS[w]
13:  |  |  |  |  |  else
```

```
14:                                    return {}
15:                                end if
16:                            end if
17:                            COMPONENT.PUSH(w)
18:                            for each w' ∈ G'.NEIGHBORS(w) do
19:                                if not visited[w'] then
20:                                    visited[w'] ← true
21:                                    Q.PUSH(w')
22:                                end if
23:                            end for
24:                        end while
25:                        if neighborColor = α then
26:                            verticesToRecolor.INSERTALL(component)
27:                        end if
28:                    end if
29:                end for
30:                return verticesToRecolor
31:            end procedure
```

2.4.2 Reducing the CP set

The set of current predecessors for a vertex i is defined as follows – let AP be the set of *adjacent predecessors*

$$AP_c(i) = \{j < i \ : \ j \in N(i) \wedge C[j] = c\} \text{ for any } i \in \{1, \ldots, n\}, c < ub$$

Let's call $r_c = \min AP_c(i)$ a *representative* of c. Let R be the set of representatives:

$$R(i) = \begin{cases} \emptyset & \text{if } AP(i) = \emptyset, \\ \{r_a, r_b, ...\} \setminus IC & \text{otherwise.} \end{cases}$$

Finally, we define the current predecessors as

$$CP = CP \cup R(i)$$

Algorithm 2.4.2 (Brelaz algorithm)

```
    Global variables:
    CS: current solution
1:  procedure DETERMINECURRENTPREDECESSORS(i)
2:      for j ← 0 to n − 1 do
3:          R[j] ← ∞
4:      end for
5:      for j ← 0 to i − 1 do
6:          if (i, j) ∈ G and CS[j] < ub and R[CS[j]] > j then
7:              R[CS[j]] ← j
```

```
        end if
    end for
    for each u ∈ R do
        if u < ∞ then
            CP.PUSH(u)
        end if
    end for
end procedure
```

This modification creates the CP set incrementally with each iteration of the while loop in the BACKWARDS procedure.

```
    Global variables:
    lb, ub: lower and upper bounds
    FC: map of feasible colors sets
procedure BACKWARDS(r)
    CP ← DETERMINECURRENTPREDECCESSORS(r)
    while CP ≠ ∅ do
        i ← CP.FRONT()
        CP ← DETERMINECURRENTPREDECCESSORS(i)
        FC[i].ERASE(CS[i])
        if FC[i] ≠ ∅ and FC[i].BEGIN() < ub then
            r ← i
            return
        end if
    end while
    r ← 0
end procedure
```

2.4.3 Proof of correctness

This algorithm brings two major changes, the first being the initial ordering and the second the definition of CP. We will prove now that none of these changes have any effect on the overall correctness of the algorithm, therefore proving that it indeed finds the best possible solution.

SLFI algorithm

Since we have already proven that the overall algorithm is correct no matter the vertex order, then the only thing left to prove is that it correctly finds the base solution and thus correctly sets the bounds. Let's assume that the coloring is not a valid one. There exist then two vertices $i < j$ sharing an edge and $C[i] = C[j] = c$. If the j vertex was colored with c then two things could have happened. Either c was a newly created color or it already was used somewhere earlier. The first case is impossible since we know that i was colored with c

first. Therefore the coloring had to have happened during the interchange, as c would be in the forbidden colors set for j.

Theorem 2.4.1. *The interchange always gives a valid partial solution.*

Proof. Let us denote the current vertex as v, and the colors of the subgraph G' as α and β. Let's assume that after the interchange on this subgraph, the solution is no longer a proper coloring. Since it was correct before then either the flipped vertex not conflicts with an unflipped one or the current vertex conflicts with one of its neighbors. In the first case the conflict would mean that there exists an edge between the two – making it appear in the same component which contradicts the assumption that only one of them was flipped. In the second case that would mean that one of v neighbors is now colored with α. But all components containing such components have been flipped so the one it is in had to have also a vertex colored with β connected to v. However, such a situation would make the algorithm fail without a flip and therefore the interchange would not have happened at all. Therefore the interchange yields a valid partial solution. □

Reducing the CP set

Firstly let's remark that the recoloring of the initial clique will not produce better coloring as a simple switch of colors used in the clique would make it return to the original state. It can be therefore forever excluded from the CP.

Let's assume for a moment that when we determine the current predecessors for a vertex we take all the neighboring vertices, not just the representatives. Dynamically updating the CP in Backwards de facto creates the monotonic paths used in the Christofides algorithm. Because for every vertex processed, all its neighbors are pushed into the queue, then every vertex with the path to the r vertex will be eventually pushed and then processed. Since any vertex from any path will be pushed then surely a vertex starting a monotonic path to r will be pushed as well.

Going back to the representatives' approach we can easily see that if two vertices adjacent to r have the same color then changing the color of the one with a bigger index will not lead to that color being freed for r. If that change was however to influence some other neighbor to free its color then there are two options possible. Either that neighbor is the smallest representative of its color – therefore the path that had to be traversed for this change would be added either way as queuing the neighbor would trigger it, or that neighbor is not the smallest representative in which case the color would not be freed as the smallest representative would not be touched by this change. Therefore it is sufficient for the representative to be included – any path that would start by the greater one being changed would be started either way.

2.5 Korman's algorithm

Korman's modification adds the *dynamic reordering* of the vertices to the base algorithm. Since during FORWARDS some vertices have fewer *feasible colors* than others, Korman suggests changing the order so as to color them first. The rule for the ordering is simple – repeatedly choose the vertex with the fewest feasible colors from the set $\{i, \ldots, n-1\}$, change the order by placing it in the ith position and then continue with the coloring.

Algorithm 2.5.1 (Korman's algorithm)

The new FORWARDS procedure maintains a queue of yet uncolored vertices ordered decreasingly by the number of distinctly colored neighbors (making the ones with fewer feasible colors appear on top). While the queue is not empty it retrieves the top vertex, places it in the ordering, determines the feasible colors, and then colors it with the smallest available one. When the loop finishes the upper bound is determined and the starting point for BACKWARDS r is set.

```
   Global variables:
   lb, ub: lower and upper bounds
   CS: current solution
   BS: best solution
   FC: map of feasible colors sets
   NO: new ordering
   G: graph
 1: procedure FORWARDS(r)
 2:     for each u ∈ G do
 3:         blockedColors[u] ← {}
 4:     end for
 5:     Q ← QUEUE()  \\ sorted by the number of distinctly colored neighbors
 6:     while Q ≠ ∅ do
 7:         i ← Q.POP()
 8:         NO.PUSH(i)
 9:         FC[i] ← DETERMINEFEASIBLECOLORS(i)
10:         if FC[i] = ∅ then
11:             r ← i
12:             return
13:         end if
14:         isColored[i] ← true
15:         CS[i] ← FC[i].BEGIN()
16:         for each u ∈ G.NEIGHBORS(i) do
17:             blockedColors[u].INSERT(CS[i])
18:         end for
19:     end while
20:     ub ← GETMAXCOLOR(CS)
21:     r ← GETMAXCOLORNODE(CS)
```

```
22:  |  BS ← CS
23: end procedure
```

Since the set of colors blocked by neighbors has to be maintained in Forwards for the purposes of the queue, the DetermineFeasibleColors method just computes the complement of this set, adding only the color current max + 1 when necessary.

```
    Global variables:
    lb, ub: lower and upper bounds
    CS: current solution
    FC: map of feasible colors sets
    NO: new ordering
 1: procedure DetermineFeasibleColors(i, blockedColors)
 2:  |  m ← getMaxColor(CS)
 3:  |  for c = 1 to min(m + 1, ub − 1) do
 4:  |  |  if c ∉ blockedColors[j] then
 5:  |  |  |  FC[NO[i]].push(c)
 6:  |  |  end if
 7:  |  end for
 8: end procedure
```

The Backwards method now has to take the new ordering NO into the account while searching for the resumption point for Forwards. When the backtrack finishes and the resumption point is determined, the appropriate vertex is recolored and the part of the new ordering after that vertex is invalidated so it can be redefined in the forward step.

```
    Global variables:
    lb, ub: lower and upper bounds
    CS: current solution
    FC: map of feasible colors sets
    NO: new ordering
 1: procedure Backwards(r)
 2:  |  for i ← r − 1 to 0 do
 3:  |  |  j ← NO[i]
 4:  |  |  FC[j].erase(CS[j])
 5:  |  |  if FC[j] ≠ ∅ then
 6:  |  |  |  if FC[j].size() > 1  or FC[j].begin() < ub then
 7:  |  |  |  |  CS[j] ← FC[j].begin
 8:  |  |  |  |  while NO.size() > i + 1 do
 9:  |  |  |  |  |  NO.pop()
10:  |  |  |  |  end while
11:  |  |  |  |  r ← i return
12:  |  |  |  end if
13:  |  |  end if
```

```
    end for
    r ← 0
end procedure
```

2.5.1 Proof of correctness

Let's start with a few observations. The BACKWARDS method removes the new ordering information for all vertices which were colored after the resumption point r, so:

Observation 2.5.1. *The ordering created by the* FORWARDS *procedure is only saved for vertices below the resumption point defined in the* BACKWARDS *procedure.*

Which instantly gives the following

Observation 2.5.2. *If a vertex was permanently reordered, its position could only be changed when a vertex before it becomes a resumption point discarding the ordering of the following vertices and that only happens when the algorithm switches to a new branch (upon the recolor of the resumption vertex).*

This leads us to the conclusion that the algorithm must always terminate.

Corollary 2.5.2.1. *The same solution (i.e. the same branch) cannot be visited twice during the* FORWARDS *method – so the algorithm must terminate as the solution tree is finite.*

Theorem 2.5.1. *The algorithm returns the optimal coloring.*

Proof. The FORWARDS method of Korman's algorithm does not in fact differ much from the FORWARDS method in Brown's algorithm. If we assume that the vertices were originally ordered with the result ordering of Korman's FORWARDS then Brown's algorithm would behave exactly the same and would produce the same coloring. The best solution that Korman's algorithm has found also produced an ordering O on the vertices. Earlier we proved that the simplified Brown's algorithm produces the optimal coloring for every possible ordering on the vertices. Therefore the solution found optimal in Korman's would also be found optimal for the simplified Brown's algorithm with vertices preordered by π. □

2.6 Complexity

The upper bound on the number of leaves in the solution tree for graph coloring is $n!$ since in the worst-case scenario the graph is a clique and $n!$ is the number of permutations of the color set. Since one iteration of FORWARDS and BACKWARDS traverses one path to the leaf down and then up (and that path is never visited again) then the complexity is given by the number of paths ending in leaves in the tree – $O^*(n!)$. The polynomial factor can be given as

$f(n)+b(n)$ – the complexities of FORWARDS and BACKWARDS respectively. The described implementation gives the polynomial factor of $O(n^2 \log n)$ for Brown's and Christofides' and $O(n^2 \log^2 n)$ for Brelaz's and Korman's versions.

Chapter 3

Computational results

3.1 Technical details

All discussed algorithms were implemented according to the pseudocode in the previous section in C++20 using the NetworKit [BCc23] library as well as the Koala [Tur+23] library. One slight difference was made in implementing Christofides' algorithm as was noticed that preordering the vertices using GLF gave a visible speedup.

3.2 Results

For testing, pseudo-random graphs were generated using the *Erdős–Rényi* [ER59], *Barabási-Albert* [BA99] and the *Watts-Strogatz* [WS98] models random graph generators from NetworKit library. All further definitions of the model are given as stated in the NetworKit library documentation. For each algorithm and combination of input values an average coloring time over these 5 tries was calculated and given as a result.

3.2.1 Erdős-Rényi model

The Erdős-Rényi generator creates a random graph in the $G(n, p)$ model, i.e., a graph with n nodes connected randomly. Each edge is included in the graph with probability p independently from every other edge. For this model, five graphs were generated for each pair n, d with the number of vertices $n \in \{30, 40, 50, 60\}$ and densities $d \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$. Table 3.1 shows the generated results. Running time of ∞ means that the programs did not terminate within 10 minutes and the + sign preceding the value means that at least one of the tries did not terminate in that time frame. As can be seen in the table, the algorithms perform fairly well on low-density graphs but fail to do so for graphs with a medium density.

Table 3.1: Results for Erdős–Rényi generated graphs

n	d	Brown	Christofides	Brelaz	Korman
30	0.1	<0.001	<0.001	<0.001	0.004
	0.3	<0.001	<0.001	<0.001	0.010
	0.5	0.002	0.002	0.002	0.008
	0.7	0.006	0.004	0.006	0.012
	0.9	0.001	0.001	0.004	<0.001
40	0.1	<0.001	<0.001	<0.001	0.061
	0.3	0.007	0.005	0.008	0.212
	0.5	0.095	0.086	0.008	1.655
	0.7	0.078	0.066	0.012	0.463
	0.9	0.017	0.015	0.015	0.027
50	0.1	<0.001	0.001	<0.001	0.047
	0.3	0.460	0.475	0.797	43.471
	0.5	25.910	23.083	5.329	166.176
	0.7	9.852	11.156	9.087	123.832
	0.9	0.225	0.236	0.016	0.098
60	0.1	0.001	0.004	<0.001	9.127
	0.3	8.846	7.826	2.648	+ 47.189
	0.5	∞	∞	∞	∞
	0.7	∞	∞	∞	∞
	0.9	2.984	67.586	64.276	27.957

Table 3.2: Results for preferential attachment generated graphs in seconds

n	k	Brown	Christofides	Brelaz	Korman
40	5	0.007	0.002	0.001	0.004
	10	0.002	0.034	0.003	0.031
	15	0.005	0.006	0.003	0.006
	20	0.002	0.004	0.003	0.004
50	5	0.002	0.004	0.002	0.005
	10	0.177	0.049	0.004	0.086
	15	0.098	0.419	0.006	0.074
	20	1.415	1.994	0.008	0.073
	25	0.392	0.044	0.014	0.274
60	5	0.059	0.011	0.004	0.009
	15	2.155	1.379	0.009	2.024
	20	0.17	11.731	0.007	0.735
	25	125.869	24.508	0.017	0.715
	30	0.052	0.123	0.018	0.084
70	5	45.143	0.008	0.003	1.632
	10	54.606	0.266	0.005	0.036
	15	26.468	8.169	0.135	0.872
	20	1.725	1.326	0.017	0.328
	25	3.125	41.383	0.73	2.192
	30	0.984	1.215	0.032	53.543
	35	0.671	0.95	0.012	9.427

3.2.2 Barabási-Albert model

The Barabási–Albert model is an algorithm for generating random scale-free networks using a preferential attachment mechanism. The network begins with an initial connected network of nodes, and new nodes are added to the network one at a time. This generator implements the preferential attachment model as introduced by Barabási and Albert. For this model, five graphs were generated for each pair n, k with the maximum number of vertices $n \in \{40, 50, 60, 70\}$ and the number of attachments per node $k \in \{5, 10, 15, \ldots, \frac{n}{2}\}$. Table 3.2 shows the generated results. As can be observed Brown's algorithm generally performs the worst for this kind of graphs and the Brelaz method gives the most promising results in all cases.

3.2.3 Watts-Strogatz model

The Watts-Strogatz model produces graphs with high clustering and low average path length – first, a regular ring lattice is generated, then some edges are rewired randomly with the probability p. As before, for this model, five graphs were generated for each of three values n, k, p with the number of vertices $n \in \{40, 50, 60, 70\}$ and the number of neighbors with which each node is joined $k \in \{5, 10, 15, \ldots, \frac{n}{2} - 5\}$ and the probability of rewiring each edge $p \in \{0.1, 0.5, 0.9\}$. Table 3.3 shows the generated results.

Table 3.3: Results for Watts-Strogatz generated graphs in seconds

n	k	p	Brown	Christofides	Brelaz	Korman
40	5	0.1	0.006	0.006	0.004	0.005
		0.5	0.005	0.007	0.003	0.004
		0.9	0.005	0.007	0.002	0.007
	10	0.1	0.024	0.025	0.009	0.068
		0.5	0.022	0.018	0.031	0.036
		0.9	0.089	0.088	0.035	0.049
	15	0.1	0.003	0.008	0.008	0.028
		0.5	0.005	0.005	0.01	0.02
		0.9	0.004	0.007	0.012	0.025
50	5	0.1	0.126	0.092	0.079	0.006
		0.5	0.081	0.049	0.006	0.01
		0.9	0.027	0.023	0.02	0.004
	10	0.1	0.409	0.353	0.362	0.097
		0.5	0.609	0.489	0.303	0.913
		0.9	0.187	0.173	0.231	0.13
	15	0.1	0.084	0.079	0.03	0.173
		0.5	0.256	0.222	0.048	2.078
		0.9	1.424	1.467	0.057	1.57
	20	0.1	0.019	0.023	0.068	0.161
		0.5	0.015	0.023	0.065	0.045
		0.9	0.019	0.029	0.065	0.029
60	5	0.1	0.145	0.204	0.132	0.027
		0.5	0.36	0.228	0.068	0.031
		0.9	0.383	0.288	0.065	0.029
	10	0.1	∞	∞	∞	∞
		0.5	0.609	0.489	0.303	0.913
		0.9	11.006	11.697	9.783	3.293
	15	0.1	17.7	21.142	8.079	32.341
		0.5	∞	∞	∞	∞
		0.9	11.6	8.093	2.916	8.412
	20	0.1	1.042	1.345	0.087	17.299
		0.5	1.244	1.473	0.515	6.103
		0.9	0.202	0.202	0.079	35.715
	25	0.1	0.009	0.02	0.112	0.037
		0.5	0.011	0.02	0.118	0.072
		0.9	0.011	0.017	0.121	0.034

Bibliography

[AH76] Kenneth Appel and Wolfgang Haken. "Every planar map is four colorable". In: (1976).

[BA99] Albert-László Barabási and Réka Albert. "Emergence of scaling in random networks". In: *science* 286.5439 (1999), pp. 509–512.

[BCc23] Humboldt-Universität zu Berlin - Department of Computer Science - Modeling, Analysis of Complex Systems, and contributors. *NetworKit*. 2023. URL: `https://networkit.github.io/index.html` (visited on 06/2023).

[BCT94] Preston Briggs, Keith Cooper, and Linda Torczon. "Improvements to graph coloring register allocation". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.3 (1994), pp. 428–455.

[BGS98] Mihir Bellare, Oded Goldreich, and Madhu Sudan. "Free bits, PCPs, and nonapproximability—towards tight results". In: *SIAM Journal on Computing* 27.3 (1998), pp. 804–915.

[BH06] Andreas Björklund and Thore Husfeldt. "Inclusion-Exclusion Based Algorithms for Graph Colouring". In: *Electron. Colloquium Comput. Complex.* TR06 (2006).

[BH08] Andreas Björklund and Thore Husfeldt. "Exact algorithms for exact satisfiability and number of perfect matchings". In: *Algorithmica* 52 (2008), pp. 226–249.

[BHK09] Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. "Set Partitioning via Inclusion-Exclusion". In: *SIAM Journal on Computing* 39.2 (2009), pp. 546–563.

[Bir12] George Birkhoff. "A determinant formula for the number of ways of coloring a map". In: *The Annals of Mathematics* 14.1/4 (1912), pp. 42–46.

[BK06] Hans Bodlaender and Dieter Kratsch. "An exact algorithm for graph coloring with polynomial memory". In: *UU-CS* 2006 (2006).

[Bre+79] Daniel Brelaz et al. "New methods to color the vertices of a graph". In: *Communications of the ACM* 22.4 (1979), pp. 251–256.

[Bro41] Rowland Leonard Brooks. “On colouring the nodes of a network”. In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 37. 2. Cambridge University Press. 1941, pp. 194–197.

[Bro72] Randall Brown. “Chromatic scheduling and the chromatic number problem”. In: *Management science* 19.4-part-1 (1972), pp. 456–463.

[Bys04] Jesper Makholm Byskov. “Enumerating maximal independent sets with applications to graph colouring”. In: *Operations Research Letters* 32.6 (2004), pp. 547–556.

[Cha+81] Gregory Chaitin et al. “Register allocation via coloring”. In: *Computer languages* 6.1 (1981), pp. 47–57.

[Chr75] Nicos Christofides. *Graph theory: An algorithmic approach (Computer science and applied mathematics)*. Academic Press, Inc., 1975.

[Coo71] Stephen Cook. *The complexity of theorem-proving procedures, STOC'71: Proceedings of the third annual ACM symposium on Theory of computing*. 1971.

[Cyg+15] Marek Cygan et al. *Parameterized algorithms*. Vol. 4. 8. Springer, 2015.

[Die05] Reinhard Diestel. “Graph theory 3rd ed”. In: *Graduate texts in mathematics* 173.33 (2005), p. 12.

[Epp02] David Eppstein. “Small maximal independent sets and faster exact graph coloring”. In: *J. Graph Algorithms Appl* 7.2 (2002), pp. 131–140.

[ER59] Paul Erdős and Alfréd Rényi. “On random graphs I”. In: *Publ. math. debrecen* 6.290-297 (1959), p. 18.

[GJS76] Michael Garey, David Johnson, and Hing So. “An application of graph coloring to printed circuit testing”. In: *IEEE Transactions on Circuits and Systems* 23.10 (1976), pp. 591–599.

[GL23] Serge Gaspers and Edward Lee. “Faster graph coloring in polynomial space”. In: *Algorithmica* 85.2 (2023), pp. 584–609.

[Gro59] Herbert Grotzsch. “Ein Dreifarbensatz fur dreikreisfreie Netze auf der Kugel”. In: *Wiss. Z. Martin Luther Univ. Halle-Wittenberg, Math. Nat. Reihe* 8 (1959), pp. 109–120.

[Hal15] Magnús Halldórsson. “Progress (and Lack Thereof) for Graph Coloring Approximation Problems”. In: *International Conference on Current Trends in Theory and Practice of Informatics*. Springer. 2015, pp. 14–23.

[Hea90] Percy John Heawood. “Map color theorems”. In: *Quant. J. Math.* 24 (1890), pp. 332–338.

[Joh73] David Johnson. “Approximation algorithms for combinatorial problems”. In: *Proceedings of the fifth annual ACM symposium on Theory of computing*. 1973, pp. 38–49.

[Kar10] Richard Karp. *Reducibility among combinatorial problems*. Springer, 2010.

[Kem79] Alfred Kempe. "On the geographical problem of the four colours". In: *American journal of mathematics* 2.3 (1879), pp. 193–200.

[KJ85] Marek Kubale and Boguslaw Jackowski. "A Generalized Implicit Enumeration Algorithm for Graph Coloring". In: *Commun. ACM* 28.4 (1985), pp. 412–418. ISSN: 0001-0782.

[Kor79] Samuel Korman. "The graph-colouring problem". In: *Combinatorial optimization* (1979), pp. 211–235.

[Law76] Eugene Lawler. "A note on the complexity of the chromatic number problem". In: *Information Processing Letters* 5.3 (1976), pp. 66–67.

[Oga86] Hideo Ogawa. "Labeled point pattern matching by Delaunay triangulation and maximal cliques". In: *Pattern Recognition* 19.1 (1986), pp. 35–40.

[Rob+96] Neil Robertson et al. "Efficiently four-coloring planar graphs". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996, pp. 571–575.

[Rob06] Fred Roberts. "On the Mobile Radio Frequency Assignment Problem and the Traffic Light Phasing Problem". In: *Annals of the New York Academy of Sciences* 319 (Dec. 2006), pp. 466–483.

[Tur+23] Krzysztof Turowski et al. *koala-networkit*. 2023. URL: `https://github.com/krzysztof-turowski/koala-networkit` (visited on 06/2023).

[Wer85] Dominique de Werra. "An introduction to timetabling". In: *European journal of operational research* 19.2 (1985), pp. 151–162.

[Wig83] Avi Wigderson. "Improving the performance guarantee for approximate graph coloring". In: *Journal of the ACM (JACM)* 30.4 (1983), pp. 729–735.

[WS11] David Williamson and David Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

[WS98] Duncan Watts and Steven Strogatz. "Collective dynamics of 'small-world'networks". In: *nature* 393.6684 (1998), pp. 440–442.

[Zyk49] Alexander Aleksandrovich Zykov. "On some properties of linear complexes". In: *Matematicheskii sbornik* 66.2 (1949), pp. 163–188.