

Total coloring of cubic graphs

Mathieu Vavrille *

M1 internship supervised by Łukasz Kowalik †

28 August 2018

Abstract

We show two algorithms improving the current best algorithm for total coloring of cubic graphs, both using polynomial space. The best of the two algorithms has time complexity $\mathcal{O}(2^{1.1943n})$ for an n -vertex graph, using a branching algorithm. This running time bound is obtained using the "measure and conquer" technique.

Contents

1	Introduction	2
1.1	Problem definition and motivation	2
1.2	Previous results	3
1.3	Result of the internship	4
2	First approach	4
2.1	The algorithm	4
2.2	Branching algorithms analysis	7
2.2.1	Usual analysis	7
2.2.2	Measure and conquer	7
2.3	Time complexity analysis	7
2.3.1	Recurrence cases	7
2.3.2	Algorithm to find good weights	8
2.3.3	Final complexity	9
3	Second approach	10
3.1	The algorithm	10
3.1.1	Idea behind it	10
3.1.2	Improved approach	11
3.2	Time complexity	12
3.2.1	Linear program analysis	12
3.2.2	Coefficients of the linear program	12
3.2.3	Constraints of the linear program	14
3.2.4	Solution and time complexity	14
3.3	For subcubic graphs	14
3.3.1	New linear program, and time complexity	15
4	Conclusion	16
A	Remarks on the internship	17
B	Coefficients of the linear program	17
C	Linear program	20
C.1	For bridgeless cubic graphs	20
C.2	For general subcubic graph	20

*ENS de Lyon, 46 allée d'Italie, 69007 Lyon, France

†Institute of Informatics, University of Warsaw, Banacha 2, 02-097, Warsaw, Poland

1 Introduction

The study of **NP**-hard problems is one of the biggest domains in computer science. On a theoretical point of view, one may want to find properties of the solutions, or find the instances of the problem that can be solved efficiently. Exponential algorithms solving **NP**-hard problems are also well studied because **NP**-hard problems appears in real life applications. The algorithms raising one optimal solution (called exact algorithms) are not the only ones. The goal is sometimes to find an algorithm running faster than an exact algorithm, but that will raise an approximation of the optimal solution. Such algorithms are called approximation algorithms and there are often theoretical guarantees on the solution. Fixed parameter tractability is a domain where we want to find some parameters of the instances and an algorithm with running time parametrized by this parameter. This is for example what happens with treewidth: [Courcelle, 1990] shows that some problems have a linear time algorithm when the input graph have bounded treewidth.

Exact algorithms are interesting on the practical and on the theoretical point of view. Improvements on the running time will allow to solve bigger instances. On a theoretical point of view, a naive approach is often easy to find, but an improved algorithm may need to find properties of the input instances and of the problem that are not clear beforehand. Some results have been found where it was really unclear if there was an algorithm better than the naive approach. Here are some of the famous examples. For the dominating set, [Van Rooij and Bodlaender, 2011] showed an algorithm with running time $\mathcal{O}(1.4969^n)$, that is better than the naive algorithm with running time $\Omega(2^n)$. For the hamiltonian cycle problem, [Bellman, 1962, Held and Karp, 1962] found an algorithm using dynamic programming running in time $\mathcal{O}(n^2 2^n)$, whereas the naive approach uses $\mathcal{O}(n!)$ operations.

Graph coloring problems are well studied (and often **NP**-hard) problems that arise in many real-life applications, often involving constraints between different objects. Work has been done on the theoretical point of view (for example, bounds on the chromatic number, i.e., the number of colors needed to color the graph), as well as on the algorithmic point of view (searching for the fastest algorithm). On the algorithmic point of view, one of the main results is due to [Koivisto, 2006, Bjorklund and Husfeldt, 2006] that independently found a $\mathcal{O}^*(2^n)$ time algorithm computing the chromatic number.

1.1 Problem definition and motivation

In this report, we use standard notation and concepts for graphs. $G = (V, E)$ will be an undirected graph, in the instances of the problem it will be cubic (all the vertices have degree three) or subcubic (the maximum degree is three), without multi-edges or loops. Unless redefined, we write $n = |V|$ and $m = |E|$. We denote by $\Delta(x)$ the degree of a vertex $x \in V$, and $\Delta(G)$ the maximum degree of the graph.

Definition 1 (Neighbouring element). u and $v \in V \cup E$ are two neighbouring elements if:

- $u, v \in V$ and $(u, v) \in E$ (adjacent vertices)
- $u \in V, v = (u, w) \in E$ (vertex and incident edge)
- $u = (x, y), v = (x, w) \in E$ (incident edges)

Definition 2 (k -total coloring). Let $G = (V, E)$ a cubic graph. A k -total coloring of G is a function $\mathcal{C} : V \cup E \rightarrow \{1, \dots, k\}$ such that every two neighbouring elements have different colors (i.e. if $a, b \in V \cup E$ are neighbouring elements, $\mathcal{C}(a) \neq \mathcal{C}(b)$).

There are three main algorithmic problems connected with k -total coloring: deciding if a given graph is totally colorable in k colors, counting the number of coloring or enumerating (i.e. listing) them. The algorithms solving the decision or counting problem can usually be modified to return a coloring.

Total coloring is related to vertex coloring and edge coloring problems. As for the other coloring problems, we have some bounds on the minimum number of colors needed noted to totally color the graph (denoted $\chi''(G)$).

Property 1. Let G be a graph, then $\chi''(G) \geq \Delta(G) + 1$

This property immediately follows from the fact that if a node has degree d , then there are $d + 1$ neighbouring elements (the vertex and all the incident edges), so at least $d + 1$ colors are needed.

The following theorem is analogous to Brooks' theorem ([Brooks, 1941]) and Vizing's theorem ([Vizing, 1964]).

Theorem 1 ([Rosenfeld, 1971, Vijayaditya, 1971, Kostochka, 1996]). *If $\Delta(G) \leq 5$, $\chi''(G) \leq \Delta(G) + 2$*

It is conjectured that Theorem 1 holds for arbitrary maximum degree.

Conjecture 1 (Total coloring conjecture, Behzad and Vizing). *For every graph G , $\chi''(G) \leq \Delta(G) + 2$*

The $(\Delta(G) + 1)$ -total-coloring problem is **NP**-complete in the general case and this statement even holds for bipartite cubic graphs [Sánchez-Arroyo, 1989]. In this internship we are interested in the 4-total coloring problem on cubic graphs. This is almost the simplest class of graphs for which the problem is **NP**-complete, and the hope is that it has enough structure to get algorithms that are much faster than the exhaustive search. Although cubic graphs might seem rather restricted, it should be straightforward to generalize our results to subcubic graphs, i.e., graphs of maximum degree three.

A lot of work has also been done on total coloring of planar graphs (see [Kowalik et al., 2008] or [Shen and Wang, 2009]).

1.2 Previous results

Now we focus on 4-total coloring of cubic graphs.

Remark. As we are dealing with exponential algorithms, we are not considering the polynomial factors in the complexity. To point this fact, we write by \mathcal{O}^* the complexity where we erased the polynomial terms. A reason behind is that $\forall k, c, \epsilon > 0, \mathcal{O}(n^k 2^{cn}) = \mathcal{O}^*(2^{cn}) = o(2^{(c+\epsilon)n})$ so a rounding up of the factor c at the decimal we want will outdo any polynomial factor.

[Golovach et al., 2010] showed a $\mathcal{O}^*(2^{13n/8}) = \mathcal{O}^*(2^{1.625n})$ branching algorithm to enumerate the 4-total colorings of a connected cubic graph using polynomial space. They also show an algorithm running in time $\mathcal{O}(12^{(1/6+\epsilon)n})$, $\forall \epsilon > 0$, which can be bounded by $\mathcal{O}(2^{0.5975n})$ using exponential space. They use a dynamic programming algorithm over a path decomposition for the last one, counting the number of 4-total coloring of any input graph. More generally, they showed that given a path decomposition of size p , all the k -total coloring can be counted in time $\mathcal{O}\left(\left(k \binom{k-1}{\lfloor (k-1)/2 \rfloor}\right)^{p+1} \cdot k \cdot k! \cdot \log k \cdot n^2\right)$.

[Bessy and Havet, 2013] showed a $\mathcal{O}^*(2^{3n/2})$ algorithm enumerating the colorings of a cubic graph in polynomial space. They used an (s, t) -ordering of the graph to color the elements in a certain order.

Even though the complexity of branching algorithms compared to dynamic programming over path decomposition is worse, branching algorithms prove to be more efficient in practice (see [Berglin, 2014] for a practical evaluation of algorithms for the related problem of 3-edge coloring).

We can easily create a general algorithm solving total coloring using the adjacency graph.

Definition 3 (Adjacency Graph). Let $G = (V, E)$ a graph. The adjacency graph of G is the graph $\mathcal{T}(G) = (V' = V \cup E, E')$ where $(\alpha, \beta) \in E'$ if α and β are two neighbouring elements in G .

Remark.

- $|V(\mathcal{T}(G))| \leq |V(G)| + |E(G)| \leq |V(G)| + |V(G)| \cdot \Delta(G) \leq \left(\frac{\Delta(G)+1}{2}\right) |V(G)|$
- G is k -totally colorable iff $\mathcal{T}(G)$ is k -vertex colorable

This graph allows us to move the problem of total coloring to the one of vertex coloring. This leads to a simple algorithm for total coloring: create the adjacency graph and apply a vertex coloring algorithm. The k -vertex colorability of a graph can be checked in time $\mathcal{O}(2^n)$ (see [Björklund et al., 2009]). The final algorithm given a graph G have time complexity $\mathcal{O}(2^{(\Delta(G)+2)n/2})$. For small number of colors (i.e. small degree), we can use faster vertex coloring algorithms such as [Fomin et al., 2007]. For four colors, that leads to an algorithm in time $\mathcal{O}(2^{1.9711n})$ for the decision problem, and time $\mathcal{O}(2^{2.4021n})$ for the counting problem.

1.3 Result of the internship

We show two different approaches leading to polynomial space algorithms which improve the current best known polynomial space algorithm. In branching algorithms, we build a recursion tree where the instances are simpler and simpler as we go down the tree. The goal is to arrive at polynomial time solvable instances in the leaves. Both algorithms presented rely on this approach, with two different kind of instances to solve polynomially at the leaves.

The first algorithm relies on the fact that checking bipartiteness (i.e. 2 colorability) is polynomial. [Byskov et al., 2005] observed that a 4-vertex colorable graph can be partitioned into two bipartite subgraphs (each one corresponding to a pair of colors). We apply this property to the adjacency graph of the input graph. The algorithm will find all such bipartite subgraph decompositions, using some properties on the total coloring (that can be seen as properties on the structure of the adjacency graph).

To estimate the complexity of this algorithm, we use the "measure and conquer" approach. In this approach we need to tune some coefficients: this is done using a script that we implemented (see section 2.2 for the details).

We finally get an algorithm of time complexity $\mathcal{O}(2^{1.1943n})$ using polynomial space to find and count the 4-total colorings of a n -vertex cubic graph.

The second algorithm uses a completely different approach. We rely on the fact that checking total-list colorability of graph of bounded treewidth can be done in polynomial time (due to Courcelle's theorem, [Courcelle, 1990]). The understanding of treewidth is not necessary for this report, more details about treewidth can be found in [Bodlaender, 1994].

Definition 4 (Total-list coloring). Let $G = (V, E)$ a graph, and l a function associating to each vertex and edge a list of allowed colors ($\forall x \in V, l(x) \in \mathcal{P}(\mathbb{N})$). A total-list coloring of G with respect to l is a function $\mathcal{C} : V \cup E \rightarrow \mathbb{N}$ where $\forall e \in V \cup E, \mathcal{C}(e) \in l(e)$ and two neighbouring elements do not have the same color.

The outerplanar graph have bounded treewidth ([Scheffler, 1986]), so checking total-list colorability on them is polynomial. The insight of our second algorithm for 4-total coloring is the following: find a matching in the graph that matches all the degree three vertices, color in every possible way each edge of this matching with one incident vertex, and check if the remaining graph (which is outerplanar) is totally-list colorable. In reality, the algorithm is a little more complicated: we do not color the edges one by one, but many at once, because a lot of local coloring are forbidden, and we do not want to explore impossible solutions.

The time complexity analysis of this algorithm is done using linear programming. The exponent of the complexity (coefficient c in $\mathcal{O}(2^{cn})$) is expressed as a maximization function, and constraints express the flow of the algorithm.

We finally get a time complexity of $\mathcal{O}(2^{1.2893n})$ and polynomial space.

Remark: As the main result was found on the last part of the internship, I will not be able to give the most formal proof of the correctness and complexity of the algorithms.

To find the complexity of the algorithms, or to speed up some analysis, I implemented some scripts that can be found online ¹.

2 First approach

2.1 The algorithm

Here we show the algorithm using bipartite decomposition of the adjacency graph. Let $G = (V, E)$ be a cubic connected graph, and $\mathcal{T}(G) = (V', E')$ its adjacency graph. Then G is 4-totally colorable is equivalent to the fact that $\mathcal{T}(G)$ is 4-vertex colorable, which is equivalent to G' being decomposable in $\mathcal{T}(G) = G'_1 \cup G'_2$, two disjoint bipartite subgraphs.

¹<https://github.com/MathieuVavrille/total-coloring-cubic>

Lemma 1. Let $G = (V, E)$ be a subcubic graph and $\mathcal{C} : V \cup E \rightarrow \{1, 2, 3, 4\}$ a total coloring of G . Let $i, j \in \{1, 2, 3, 4\}$ two different colors. Consider a connected component Q of the subgraph of $\mathcal{T}(G)$ induced by the two colors i and j , i.e. $\mathcal{T}(G)[\mathcal{C}^{-1}(\{i, j\})]$.

Let $S = V(Q)$, $S_V = S \cap V$ and $S_E = S \cap E$. Then either

- S_E forms a cycle of even length in G and $S_V = \emptyset$, or
- S_E forms a path in G and S_V is a subset of endpoints of the path.

Moreover, if for v an endpoint of S_E , $\Delta(v) = 3$ then $v \in S_V$. Also, if both endpoints are in S_V , then $|S_E| \geq 2$

Proof. We use the fact that three elements neighbouring each other can't be colored in two colors (so they can't be in S at the same time).

First let $v \in V$. There can't be three incident edges to v in S_E because these three incident edges would be neighbouring each other, so can't be colored in two colors. So every vertex is incident to at most two edges in S_E , so S_E is a path or a cycle.

If S_E is a path, let $v \in S_V$. If v is not an endpoint, it is necessarily incident to two edges in S_E (because S is connected). These three elements are neighbouring each other, so they can't be colored in two colors. v is necessarily an endpoint of the path S_E .

If S_E is a cycle, by the same argument, there can't be any element in S_V . Also, it has even length because it is impossible to edge-color an odd length cycle with two colors.

If S_E is a path and v one of its endpoint with $\Delta(v) = 3$. If $v \notin S_V$, then v and the two edges incident to v are not colored with the same colors as S_E , so are colored with the other two colors, which is impossible because it is three neighbouring elements. So $v \in S_V$.

If both endpoints are in S_V , suppose that $|S_E| = 1$. Then the elements in $S_V \cup S_E$ are three neighbouring elements that can't be colored in two colors. So necessarily, $|S_E| \geq 2$. □

In what follows, we focus on cubic graphs. By applying Lemma 1 first on colors 1, 2 and next to colors 3, 4, we get the following corollary.

Corollary 1. Let G be a cubic graph. If G is totally 4-colorable, then $E(G)$ can be partitioned into two subsets E_R (red edges) and E_B (blue edges) so that both E_R and E_B are collections of paths of length at least two, and even cycles.

We call the edge coloring of $E(G)$ described in Corollary 1 a red-blue edge coloring.

The algorithm is as follows. We enumerate the red-blue colorings of G , and for each of the color $Z \in \{E_R, E_B\}$ we check in linear time if the corresponding subgraph of $\mathcal{T}(G)$ induced by Z and the endpoints of paths in Z is bipartite. This part is done by an algorithm CHECKBIPARTITE(G, Z) that we don't explain more in details.

To simplify the algorithm, we introduce the notion of "technical edge".

Definition 5 (Technical edge). An edge $xy \in E$ is technical if

- it is not colored,
- there is only one edge incident to x that is colored (say in red),
- there is only one edge incident to y that is colored by the same color (in red).

The branching algorithm is presented in Algorithm 1.

This algorithm answers to the decision problem, but it can easily be extended to an algorithm that counts the number of coloring, without any slow-down in the running time.

Proof. Explanation of the recursive calls I don't prove that the algorithm is correct but instead I give the intuition about it. I will also explain the recursive calls. A detail of the recursive calls and an illustration can be found in Table 2.

The last two conditions of the algorithm (8 and 10) are the two cases to deal with technical edges, and the other edges are treated on the first two recursive calls, that is why the algorithm will color all the edges. Now I need to explain why we do these recursive calls.

On Line 4 there is only one recursive call. This is due to the fact that we can't color the three incident edges of a vertex by the same color, so there is only one color possible for the edge.

Algorithm 1 BIPARTITESPLIT(G_0, G, E_R, E_B)

```

1: Input: The initial cubic graph  $G_0$ , the current red-blue decomposition  $E_R, E_B$ , and the re-
   remaining graph  $G = (V, E)$ 
2: Output: True if  $G$  is totally 4-colorable, False otherwise
3: if  $\exists xy \in E$  s.t.  $x$  is incident to two edges colored with the same color, say blue then
4:   return BIPARTITESPLIT( $G_0, G - xy, E_R + xy, E_B$ )
5: else if  $\exists xy \in E$  s.t.  $xy$  is not technical then
6:   return  $\begin{cases} \text{BIPARTITESPLIT}(G_0, G - xy, E_R + xy, E_B) \\ \vee \text{BIPARTITESPLIT}(G_0, G - xy, E_R, E_B + xy) \end{cases}$ 
7: else if  $\exists xy, yz, zx \in E$  s.t.  $x, y$  and  $z$  are touched by edges colored by the same color, say blue
   then ▷ It is a triangle
8:   return  $\begin{cases} \text{BIPARTITESPLIT}(G_0, G - \{xy, yz, wx\}, E_R + \{wx, yz\}, E_B + xy) \\ \vee \text{BIPARTITESPLIT}(G_0, G - \{xy, yz, wx\}, E_R + \{xy, wx\}, E_B + yz) \\ \vee \text{BIPARTITESPLIT}(G_0, G - \{xy, yz, wx\}, E_R + \{xy, yz\}, E_B + wx) \end{cases}$ 
9: else if  $\exists wx, xy, yz \in E$  s.t.  $x$  and  $y$  are incident to edges colored by the same color, say blue
   then
10:  return  $\begin{cases} \text{BIPARTITESPLIT}(G_0, G - \{xy, yz, wx\}, E_R + \{wx, yz\}, E_B + xy) \\ \vee \text{BIPARTITESPLIT}(G_0, G - \{xy, yz, wx\}, E_R + \{xy, wx, yz\}, E_B) \\ \vee \text{BIPARTITESPLIT}(G_0, G - \{xy, yz, wx\}, E_R + \{xy, wx\}, E_B + yz) \\ \vee \text{BIPARTITESPLIT}(G_0, G - \{xy, yz, wx\}, E_R + \{xy, yz\}, E_B + wx) \end{cases}$ 
11: else ▷ Every edge is colored
12:   CHECKBIPARTITE( $G, E_B$ )  $\wedge$  CHECKBIPARTITE( $G, E_R$ )
13: end if

```

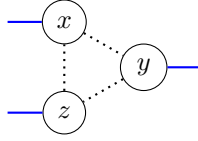


Figure 1: Case treated by line 8

Line 6 is the natural branching. We do two recursive calls: one with the edge colored in red, the other in blue.

On Line 1 there is the case where we have an uncolored triangle xyz where x, y and z are incident to an edges colored by the same color (say blue), as we can see on Figure 1. Then we want to know how to color the edges xy, yz and zx . The only possibility is to have only one edge colored in blue amongst the three. This is because if we have the three or two edges colored in blue, then we have a vertex surrounded by blue edges, which is forbidden, and if there is three edges colored in red, we have a cycle of odd length which is forbidden by Lemma 1. That is why we have three recursive calls, each one corresponding to giving the color blue to one edge.

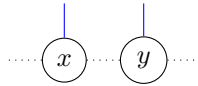


Figure 2: Case treated by line 10

On Line 10, we have the last recursive calls, and the case is illustrated in Figure 2. If we color xy in blue, then we can immediately apply the reduction rule by coloring wx and yz in red. If we color xy in red, there is more possibilities. We use the property that we can't have a path of length 1 colored in red. That means that there is necessarily an edge colored in red amongst wx and yz . The possibilities are to color both in red, or only one, which leads to three cases in total when coloring xy in red. \square

2.2 Branching algorithms analysis

2.2.1 Usual analysis

The algorithm we have shown as well as many other branching algorithms used for exponential algorithms, uses two kind of recurrence rules: the *reductions* and the *branchings*. The reductions are cases where there is only one recurrence call to a smaller instance (e.g. line 4). The branching rules recursively call the function on some other instances of smaller size. The flow of the algorithm can be seen as a recursion tree, where reduction rules have only one child, and branching rules have more than one. This recursion tree is of polynomial depth, so the number of reduction rules is polynomially bounded by the number of branching rules.

In order to bound the complexity up to polynomial factors, we only need to focus on branching rules. For each rule, we act like it was the only one in the algorithm, so that the complexity follows a recurrence of the form $T(n) = T(n - a_1) + T(n - a_2)$ (in the case of 2 branches). The complexity of this kind of recurrence is found by finding the biggest real zero of the function $X \mapsto 1 - X^{-a_1} - X^{-a_2}$. This zero is called a work factor, and noted $\lambda(a_1, a_2)$, gives us the complexity of the algorithm as $T(n) = \mathcal{O}^*(\lambda(a_1, a_2)^n)$.

To find the complexity of the whole algorithm, we compute the work factor of all the branching rules, and the biggest one will give us a bound on the total complexity. This is due to the fact that in the worst case, the worst branching rule may apply all the time, giving the complexity depending on the worst work factor.

2.2.2 Measure and conquer

We apply the "measure and conquer" ([Fomin et al., 2009]) approach to have a better bound on the complexity. The main idea behind measure and conquer is that in the previous approach, the worst recurrence was the only one taken into account, even if it cannot happen all the time. What we do to prevent this fact is that we define a new measure of the instance. The measure defined in the previous analysis is just the number of nodes or edges, but we can do better. If we put weights on the different cases that can appear (for example degree 3 or degree 2 vertices), we can reflect in the recurrences the fact that some vertices are easier to deal with than other. For example, vertices touched by two edges both colored in red (or blue), as in rule 4, should not appear in the size of the instance, because dealing with them is "easy", i.e. they will trigger a reduction rule. We also want to say that the vertices with one incident edge already colored are easier to deal with than vertices without incident edge colored, because we already have some constraint on them. To this extent, we define weights w_i on the different cases.

The recurrence relations become now $T(n) = T(n - a_1) + T(n - a_2)$ but now the parameters a_1 and a_2 depend on the weights of the instance (they are functions $a_i(w_1, \dots, w_k)$). The biggest work factor are computed according to these weights. Now we want to find the weights that minimize this work factor. The resulting complexity will just be $\mathcal{O}(\lambda^s)$ where λ is the biggest work factor according to the weights, and s is the size of the instance. Usually we want to find a nice way to express the complexity, as a function of the number of vertices or edges, so we want some bound on s . In our algorithm, the initial size of the instance will be the number of nodes, so the final complexity will be $\mathcal{O}^*(\lambda^n)$.

2.3 Time complexity analysis

Here we present the time complexity analysis, the choice of the weights, and the script implemented to find them.

2.3.1 Recurrence cases

We choose to put weights on nodes according to how adjacent edges are colored. We have a simple representation of these weights: the name of the weight is $a_{x,y}$ when the node have x adjacent edges colored by red, and y by blue. Table 1 shows the eight different colorings of edges incident to a vertex, each corresponding to a different weight.

We want to say that the vertices that will trigger a reductions and the vertices already treated (the three incident edges are colored) do not increase the complexity, so the weights $a_{0,2}, a_{1,2}, a_{2,0}, a_{2,1}$ are equal to zero. On the other hand, $a_{0,0}$ should be equal to 1 because initially the graph contains only uncolored edges, so the initial size is $\sum_{v \in V} a_{0,0} = n$. By symmetry we

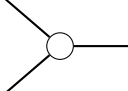
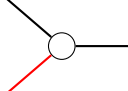
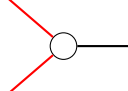
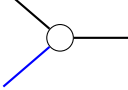
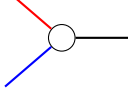
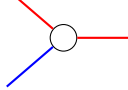
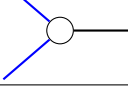
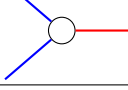
$a_{x,y}$	$x = 0$	$x = 1$	$x = 2$
$y = 0$			
$y = 1$			
$y = 2$			

Table 1: Table linking the vertices to their coefficient

have $a_{0,1} = a_{1,0}$ because red and blue play the same role. We have only two weights to tune : $a_{0,1}$ and $a_{1,1}$. Before getting started to show the recurrences, we need to understand how the size of the instance decreases when coloring an edge. Take a vertex whose weight is $a_{x,y}$. Its weight will become $a_{x+1,y}$ after coloring in red an edge adjacent to it (same for blue by increasing y). It can be seen graphically in the table as going to the box on the right (down when increasing y). The total weight of the instance decreases by $a_{x,y} - a_{x+1,y}$. We will use this fact, and even more: when we color an edge, we will decrease the weights of the incident vertices. When we don't know this weight, we can just look at the worst case. The worst case is the one that decrease the less the size of the instance, which is $\min(a_{x,y} - a_{x+1,y})$ (for the case of coloring with red), where $x, y \in \{0, 1\}$. The restriction on x and y is due to the fact that we will only reduce (and not branch) when x or y are equal to 2. Knowing this we can find the recurrence relation one by one. Table 2 shows all the recurrences of the algorithm used in the analysis. In the end, there are twelve different recurrences, but four of them are symmetrical to some of the other eight (they need to be present, but we won't talk about them, as it is only switching the colors).

In Table 2, we can see on which line of the algorithm the case happen. Sometimes, there are more than one case for the same algorithm line, it is because we have to distinguish cases in the analysis. On the illustration of each case we marked in dotted the edges that will be colored. The last two cases are the only ones that differ on this point, because we need to color three edges. The column "symmetry" tells if the case is symmetrical (it means that the same case will exist by swapping red and blue). The last column gives the size of the instances for each recursive call. The last two cases have more than two calls.

Proof. To explain the size of the recursive calls, we just need to find the weights that are changing. With this we can easily prove the lines 1, 3, 4 and 7 of the table.

Some harder cases are the ones where a min function appears in the recursive calls. This is due to the fact that the branching will trigger a reduction rule. As the reduction rule will only reduce the size of the instance, we consider it right now. As we don't know what is the weight of the vertex that will decrease, we take the worst case. This explains the lines 2, 5 and 6.

The last case in line 8 is a little different. Recall that in this case, wx, xy and yz are uncolored, and x and y are touched by edges colored by the same color, say blue (by definition of technical edge). What we also know is that there are not any non technical edges, so w and z are also touched by blue edges. Knowing this fact, it is easy to find the size of the recursive calls. \square

2.3.2 Algorithm to find good weights

Given the weights $a_{0,1}$ and $a_{1,1}$, we can now find the complexity of the algorithm by computing the roots of the functions associated to the recurrences (work factors), and taking the maximum one. The power of measure and conquer lies in the fact that we choose the weights that we want ($\in [0, 1]$), so we will choose the ones that minimize the work factor.

Work has already been done by [Fomin et al., 2009] to find these optimal weights. As we only need to find 2 weights, we can design a simple script.

The idea is to apply a kind of Monte-Carlo search: we generate all the pairs of weights up to a certain precision, we compute the work factor, and we take the weights that minimize the work factors. With my laptop, this method can only find weights up to 2 digits precision. We need to

case	illustration	line	symmetric ?	size of recursive calls
1		Line 6	No	$s - 2a_{0,0} + 2a_{0,1}$ $s - 2a_{0,0} + 2a_{1,0}$
2		Line 6	Yes	$s - a_{0,0} - \min(a_{x,y} - a_{x+1,y})$ $s - a_{0,0} - a_{0,1} + a_{1,0} + a_{1,1}$
3		Line 6	No	$s - a_{0,0} - a_{1,1} + a_{0,1}$ $s - a_{0,0} - a_{1,1} + a_{1,0}$
4		Line 6	No	$s - 2a_{1,1}$ $s - 2a_{1,1}$
5		Line 6	Yes	$s - a_{1,1} - a_{0,1} - \min(a_{x,y} - a_{x+1,y})$ $s - a_{0,1}$
6		Line 6	Same case	$s - a_{0,1} - a_{1,0} + a_{1,1} - \min(a_{x,y} - a_{x+1,y})$ $s - a_{0,1} - a_{1,0} + a_{1,1} - \min(a_{x,y} - a_{x,y+1})$
7		Line 8	Yes	$s - 3a_{0,1}$ $s - 3a_{0,1}$ $s - 3a_{0,1}$
8		Line 10	Yes	$s - 4a_{0,1} + 2a_{1,1}$ $s - 4a_{0,1} + 2a_{1,1}$ $s - 4a_{0,1} + a_{1,1}$ $s - 4a_{0,1} + a_{1,1}$

Table 2: Recurrences for each case of the algorithm, dotted segments denote the edges being colored

have a better precision to be able to find a better work factor. To this extent, we improve the procedure by searching further near the optimal point that we found.

Example: We apply a first search by sampling 10 values for each weight (so 100 values in total) between b and 1. We find the weights (0.6, 0.4), and now we will apply the same procedure but but with $a_{0,1} \in [0.5, 0.7]$, $a_{1,1} \in [0.3, 0.5]$ and sampling the same number of values.

This method allows us to find the best weights at the precision we want. The code is available online ²

It might seem to be an empirical method, but there is a theoretical background behind all of this, that we will not prove, but that can be found in [Eppstein, 2004]. The main theorem we need is that the work factors are semi-convex functions of the coefficients.

2.3.3 Final complexity

As we said at the beginning we are only interested in non polynomial computations. All the computations within each call of the main function is linear, and the final bipartite check is also polynomial. After running our script, we find that good coefficients are $a_{0,1} = 0.5813$, $a_{1,1} = 0.4187$, giving a final complexity of $\mathcal{O}(2^{1.1943n})$ (by searching for better coefficients, we can get $\mathcal{O}(2^{1.194242n})$), but such improvement starts to be irrelevant to the complexity).

The algorithm can be extend to counting the number of coloring. To our knowledge, there is no algorithm solving only the decision or the counting problem, so the best algorithm known is

²https://github.com/MathieuVaville/total-coloring-cubic/blob/master/edge_choosing.py

the one for enumeration by [Bessy and Havet, 2013]. We improve this algorithm but only for the decision and counting problem as we can't solve the enumeration problem with our algorithm.

3 Second approach

3.1 The algorithm

This algorithm solves only the decision problem.

3.1.1 Idea behind it

For the second algorithm, we have to distinguish on the input graph. One analysis will be done with a bridgeless cubic graph, and the other one will be on a general subcubic graph. For now we focus on bridgeless cubic graphs

This approach uses the same method of finding a some graphs on which we can solve polynomially some problem. The main idea of the algorithm is to find a perfect matching in the graph that matches all the degree three vertices, and color this matching such that the only elements remaining are of degree 2, so the remaining graph is an union of cycles. For the matching, we use the following theorem.

Theorem 2 ([Petersen, 1891]). *Let G be a bridgeless cubic graph, then G admits a perfect matching.*

This theorem ensures that we will be able to find a perfect matching in our graph. We can improve a little this algorithm by not coloring some edges of the matching. With this improvement (and by choosing the right edges) the remaining graph is outerplanar. After coloring the matching, we simply have to check the total-list colorability of the remaining graph, which can be done in polynomial time.

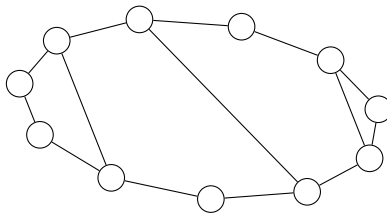


Figure 3: An example of outerplanar graph

Definition 6 (Outerplanar graph).

An outerplanar graph is a graph that has a planar drawing for which all vertices belong to the outer face of the drawing. There is an example in figure 3.

We are interested in outerplanar graphs because it is a class of graphs with bounded treewidth, also called partial- k -trees (proved by [Scheffler, 1986]). Many **NP**-hard combinatorial problems can be solved in polynomial time on families of graphs of bounded treewidth. We will use the following theorem that shows the power of bounded treewidth graphs.

Theorem 3 ([Courcelle, 1990]).

Every graph property definable in the monadic second-order logic of graphs can be decided in linear time on graphs of bounded treewidth.

As total-list coloring can be expressed in the monadic second-order logic, we can check it in linear time on outerplanar graphs. This gives us the last step of our algorithm (we will not give pseudo code for this, we are only interested in the fact that this polynomial algorithm exists).

The main algorithm is as follows: find a perfect matching, color each edge with one of its endpoint in every possible way, check total-list colorability of the remaining graph. We have to color one of the endpoints because if we don't, we don't really have an outerplanar graph to total-list color. The fact is that each edge creates a constraint between the 2 endpoints, and if such constraint is still present, we can't apply total-list coloring. Coloring one endpoint will allow to remove this constraint so that the remaining graph will be outerplanar.

We can easily find the time complexity of this algorithm. There are $\frac{n}{2}$ matched edges, and each time we color an edge and an incident vertex, we need 12 branchings (4 colors possible for the edge, and 3 for the vertex). The last step being polynomial, we don't consider it in the complexity. The time complexity is then $\mathcal{O}^*(12^{n/2})$ which can be bounded by $\mathcal{O}(2^{1.8n})$. As it is, this algorithm does not improve the current best algorithm by [Bessy and Havet, 2013]. We have to improve it.

3.1.2 Improved approach

Before giving the algorithm, we need to define some notions. We suppose that we already found a matching called M .

Definition 7 (Allowed path, cycle and chordal path). An allowed path of size n is a sequence x_1, \dots, x_l of vertices such that $(x_1, x_2), \dots, (x_{l-1}, x_l) \in E \setminus M$, and every matched edge touching x_i are different and not colored.

It is an allowed cycle when $(x_1, x_l) \in E \setminus M$.

It is an allowed chordal path when $(x_1, x_l) \in M$. For an allowed chordal path, we call inner vertices the vertices $x_i, 2 \leq i \leq n-1$

When we say that we want to color an allowed path or cycle, we want to color the vertices x_1 to x_l and the matched edges incident to them in every possible way. For an allowed chordal path, we don't color the elements x_1, x_l and (x_1, x_l) . Not coloring the last matched edge of an allowed chordal path will create some "marked" edges. These marked edges will be treated by the algorithm checking total list coloring on outerplanar graphs.

Coloring every edge (with an incident vertex) of an allowed path (in every possible way) needs less branching than coloring each edge separately. We now suppose that we have an algorithm COLORREC that can color in every possible way an allowed path, cycle or chordal path, and check if can be extended (locally) to a legal coloring. If the coloring can be extended, it will call recursively the main function. This function will also mark the edges of the allowed chordal paths that will not be colored. We also suppose that we have a function ISLISTCOLORABLE that given an outerplanar graph and a partial coloring will return True if G is totally list colorable.

Algorithm 2 ISTOTALLYCOLORABLE(G, col, M)

```

1: Input: The initial graph  $G$ , the current coloration  $col$ , and a matching that matches all degree
   three vertices  $M$ .
2: Output: True if  $G$  is totally 4-colorable, False otherwise
3: if There is an allowed path  $x_1, \dots, x_l, l > 7$  then                                ▷ Take the biggest path
4:   return COLORREC( $G, col, M, x_1, \dots, x_l$ )
5: else if There is an allowed cycle  $x_1, \dots, x_l$  then
6:   return COLORREC( $G, col, M, x_1, \dots, x_l$ )
7: else if There is an allowed chord  $x_1, \dots, x_l$  then
8:   return COLORREC( $G, col, M, x_2, \dots, x_{l-1}$ )
9: else if There is an allowed path  $x_1, \dots, x_l$  then                                ▷  $l < 7$ 
10:  return COLORREC( $G, col, M, x_1, \dots, x_l$ )
11: else
12:  return ISLISTCOLORABLE( $G, col$ )
13: end if

```

The algorithm always takes the biggest allowed path, cycle or chordal path first. Intuitively, the bigger paths will have less allowed local colorings that are in the end forbidden. This allows us to speed up the algorithm by treating many matched edges at once.

The constant 7 that appears in the algorithm is not chosen at random, it is due to the fact that it is the one that makes our estimation of the complexity the better. It will be explained in the next part.

In facts, for the purpose of the time complexity analysis I had to implement a script that can easily be transformed into the function COLORREC. The script works as follows : given a graph, you can tell what are the edges already colored and it will generate all the legal colorings of the remaining graph. Then we are able to see all the legal coloring of a subset of vertices and edges of this graph (the matched edges for example). This way, we will be able to call recursively the main function on every locally legal coloring, or we can count them (for the purpose of the time complexity analysis).

3.2 Time complexity

3.2.1 Linear program analysis

To estimate the running time of the algorithm, we introduce a new way to analyse branching algorithms. Recall that the idea behind measure and conquer is to say that the worst case can't happen all the time, that is why we define a new size of the instance.

Here the algorithm is a little different. When considering an allowed path (or something else), we will always do a certain number of recursive calls, and this number does not depend on the other colors already chosen. Sometimes it depends, but to simplify the analysis we look at the worst case, the one with the most number of branches.

Knowing this allows us to do a finer analysis where we can count exactly for each case the number of branches. Then we can parametrize the problem. For each case, we create coefficient w_i that will be the proportion of the time this case i happen. With this coefficient it is easy to find the final complexity. Assuming that the case i happen w_i times (in proportion), and each time it happen the algorithm makes z_i branches, the final complexity is $\mathcal{O}(z_1^{w_1} \dots z_r^{w_r}) = \mathcal{O}(2^{\sum_i \log_2(z_i)w_i})$. This is the analysis of a single execution, in the worst case we look at the worst coefficients, which consists in maximizing the linear function $\sum_i \log_2(z_i)w_i$ where z_i are given and $\sum_i w_i = 1$.

This way we can estimate the complexity, but we have even more power. We can add (linear) constraints to this maximization function to say that some cases can't happen some times. For example, if in one case we already have an edge colored, it means that beforehand there was one case that colored this edge. This type of constraint can be added to the maximization function. In the end, we can solve a linear system and the optimal solution will be the an estimation of the complexity of the algorithm.

This way of analyzing the running time of an algorithm has (to our knowledge) never been done. It seems to be a more powerful way to estimate the complexity than measure and conquer on this particular algorithm. Yet, it can't be applied all the time. I tried to apply the same analysis on the previous algorithm (with red-blue coloring) but it was not really suited, and measure and conquer gave a better estimation. It seems that the two methods can not really be compared to each other, it depends on the algorithm that is being analyzed.

3.2.2 Coefficients of the linear program

The creation of the linear program is done in two parts. We find the cases on which we will put coefficients, and we find the number of branches each case creates. The linear program is also a good way to know what case increases the complexity: when solving the linear program, the cases that have non zero coefficients are the ones that appear in the worst case. To improve the algorithm, or the estimation of the running time we know that we will have to look in detail in these cases. I did this a lot of time during the analysis, and now I will present the final result.

I will not explain all the constraints in detail, but instead I will give the idea behind them. All the coefficients are listed in the Appendix B with an illustration, and I explain here what they represent.

To have a better insight of the algorithm, we can see it in two parts: the first part (first three conditions) will deal with long allowed paths, cycles and chordal path, the second part will be dealing with the remaining small allowed paths. I will not enter in the details, but we can assume (in the worst case) that all the big allowed paths considered are of length exactly seven. The name of the coefficient for the big paths will be `bigpath`. We also have the coefficients `cycle_i` and `chord_i` for $3 \leq i \leq 6$ for the allowed cycles and chordal paths. The constraint on i is due to the fact that there are no cycles or chordal path of length 2 because there are no double edges, and $i \leq 6$ because otherwise, we would have an allowed path of length 7 which is impossible because this case would have been already treated before in the algorithm.

Recall that when dealing with allowed chordal paths, the algorithm will mark some edges that will not be colored. Marking the edges will decrease the number of branches the algorithm makes (because we don't color an edge), but the counter part is that it gives less information and constraints for the next steps of the algorithm.

The last condition is the one where we have to distinguish a lot of cases. There are three main cases, depending on why we are not able to extend the path. If we are not able to get a bigger path, it means that the next edges are either already colored or marked. There are three possibilities

for this: either the two matched edges surrounding the path are already colored, or one is marked and the other is already colored, or the two are marked. The paths of length 1 and 2 are treated separately (to improve the analysis). For the allowed paths of length 3, 4, 5 and 6 surrounded by already colored edges, we have the coefficients `surr3`, `surr4`, `surr5` and `surr6`. For the paths that are surrounded by one edge marked and one edge colored we have the coefficients `semi3` and `semi4`. There is no coefficient `semi5` because it would mean that we have a path of length 5 next to a marked edge (so a chordal path), so that in a previous step of the algorithm we had a path of length at least 7, which would have been treated on its own. By the same argument, there can only be path surrounded by marked edges of length less than 3.

Now for paths of length one we can look a little bit more in details. As it can't be extended in a path of length two or more (because we deal with the bigger paths first) we have more information.

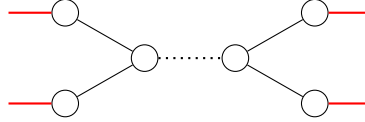


Figure 4: Illustration of paths of length 1 (in dotted)

The Figure 4 shows that on both sides of the matched edge (the dotted edge) there are already colored or marked edges. This fact will allow us to have more constraints on the linear program, and to have a better estimation of the complexity.

There are six coefficients for the paths of length one. We distinguish on whether the four surrounding edges are colored or marked. The possibilities are : four colored edges (coefficient `f4`), three colored edges (and one marked, coefficient `f3`), two colored edges that are on the same side of the matched edge (i.e. on the right of the illustration of figure 4, coefficient `f2o`), two colored edges that are on both sides of the matched edge (coefficient `f2c`), one colored edge (coefficient `f1`) and finally four marked edges (coefficient `f0`).

For the paths of length two, the principle is the same. We first remark that every matched edge of the path has to be surrounded by two colored/marked edge (one on each side of the edge). This is due to the fact that there are no allowed paths of length three in the graph because we deal with the bigger paths first. This way, we can also distinguish on whether the edges are already colored or marked.

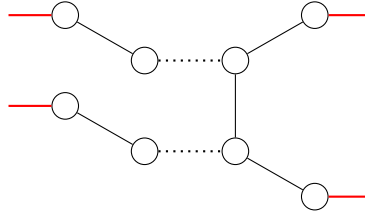


Figure 5: Illustration of paths of length 2 (in dotted)

The Figure 5 shows that the path is surrounded (on the right hand side) by colored or marked edges, and each matched edge of the path is near another colored or marked edge. There are nine coefficients to represent all the cases possible, noted `g..` where the first dots is a digit between 0 and 2, and the second one is a letter between `a` and `c` (for example `g2b`). The digit correspond to how many colored edges are surrounding the path (right hand side of the illustration), and the letters correspond to how many colored edges are near each matched edges (left hand side of the illustration), where $a = 0$, $b = 1$ and $c = 2$.

This concludes the explanation of the coefficients, for a representation of the cases (because they are easy to see graphically), one can refer to the Appendix B.

The number of branches is also shown in the appendix. Finding the number of branches was mostly done with a script that I implemented. On some cases where there are more than one possibility for the case, I take the worst case. There is only two cases that are done by hand, where we can find a closed formula for an arbitrary length of path.

3.2.3 Constraints of the linear program

The full linear program can be found in the Appendix C.1 The maximization function is found using the number of branches of each case. For the constraints, we have to look at when the cases can happen, and what is needed for them to happen. The first step is to see that the small allowed paths ($l < 7$) are the only ones that requires some edges to be already colored or marked. For example the case `surr3` need two edges to be colored (one on each side of the path), but each edge already colored can appear in two cases. Then, we know that the number of cases `surr3` is less than the number of already colored edges. We can easily know the number of already colored edges just by looking at the cases that will color edges. At the beginning, edges can be colored with the big allowed paths, cycles or chordal paths.

We create the variables y that represent the number of marked edges, and z that represent the number of colored edges that are colored by big allowed paths, cycles or chordal paths. For example, each chordal path will create one marked edge, and each path, cycle, and chordal path will also color some edges.

Then with y and z we can get the constraints on the paths of length greater or equal than three. We also have to update the number of colored and marked edges that can be used in the rest of the algorithm. Each case will "use" some colored and marked edge, but it will also color some edges. We create the variables y_2 and z_2 that are the number of marked and colored edges before starting to color the paths of length two.

We apply the same method on paths of length two, and get y_1 and z_1 that are the number of marked and colored edges before starting to color the paths of length one. Then finally we color the paths of length one.

At the end, we ensure that the sum of the coefficients is $1/2$ because we color (or mark) $n/2$ edges in total (every perfect matching in a cubic graph contain $n/2$ edges).

I will not detail the linear program more because it is mainly a case analysis, just looking at what is necessary for each case, and what is created.

3.2.4 Solution and time complexity

The goal of the linear program is to find an estimation of the complexity of the algorithm. The optimal solution gives us the exponent of the complexity. I used an online solver³ to solve the linear program. The optimal solution is 1.2982. The only nonzero coefficients are `semi3=0.08`, `f4=0.14`, `chord_4=0.04` (recall that we have to multiply by two these coefficients to know the proportion of the cases appear).

To conclude, we have an algorithm running in time $\mathcal{O}(2^{1.2982n})$ and polynomial space.

3.3 For subcubic graphs

If the graph is not bridgeless and cubic, we first apply an algorithm reducing the graph to a bridgeless graph without paths of degree two vertices longer than two or vertices of degree zero or one. The algorithm given by Algorithm 3 will take a general subcubic graph as input, simplify it and call the main function on a graph which is 4-totally colorable if and only if the input graph was 4-totally colorable. We can also cause every degree three vertex to be adjacent to another degree three vertex. This allows to get constraints on the number of degree two edges.

We have the issue that there is maybe no perfect matching in the new graph. We can instead find a matching with the property that deleting the edges of the matching will create a graph of bounded treewidth.

Let G be a bridgeless subcubic graph. We create the graph G' by contracting the paths of vertices of degree two (if there is $\deg(x) = 2$, and $(y, x), (x, z) \in E$ then the new graph has the edge $(y, z) \in E'$). This graph is bridgeless and cubic, we can apply the Petersen's theorem. To come back to the original graph, for each matched edge that exists in G' but not in G , we choose one of its endpoints, and take the edge to this endpoint in G to be matched. Figure 6 shows the steps of this construction.

With this construction, deleting the matched edges will give a graph of bounded treewidth, but that is not outerplanar this time.

³<https://www.zweigmedia.com/RealWorld/simplex.html>

Algorithm 3 $\text{SIMPLIFY}(G)$

```

1: Input: The initial subcubic graph  $G = (V, E)$ 
2: Output: True if  $G$  is totally 4-colorable, False otherwise
3: if  $\exists x \in V, \deg(x) \leq 1$  then
4:   return  $\text{SIMPLIFY}(G - x)$ 
5: else if  $\exists x \in V, s.t. \forall (x, y) \in E, \deg(y) = 2$  then
6:   return  $\text{SIMPLIFY}(G - x)$ 
7: else if  $\exists e \in E$  s.t.  $G - e$  has two connected components  $G_1$  and  $G_2$  then
8:   return  $\text{SIMPLIFY}(G_1) \wedge \text{SIMPLIFY}(G_2)$ 
9: else if  $\exists xy, yz \in E, \deg(x) = \deg(y) = \deg(z) = 2$  then
10:  return  $\text{SIMPLIFY}(G - \{x, y, z\})$ 
11: else
12:   return  $\text{ISTOTALLYCOLORABLE}(G)$ 
13: end if

```

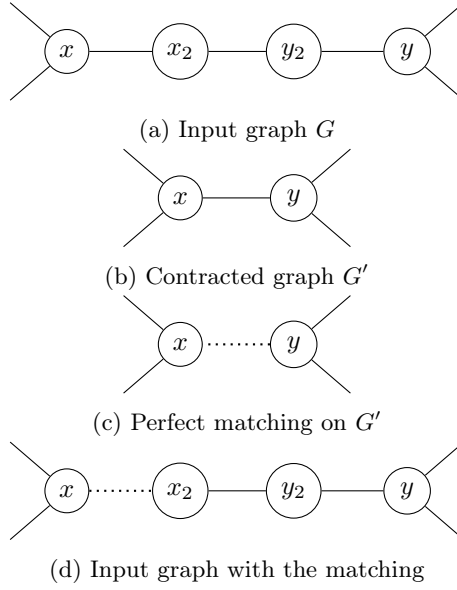


Figure 6: Procedure to find the matching

3.3.1 New linear program, and time complexity

I will not explain the linear program, because the analysis is the same than with the previous one. We create new variables for the cases with degree two vertices. The linear program is presented in the Appendix C.2. The reason behind the fact that I do not explain it a lot is because the algorithm is worse than the previous one, and I just stopped working on the linear program to focus on the faster algorithm.

The new coefficients created are **d1i**, **d1o**, **d2i**, **d2o** to represent the cases where we have degree two vertices. Respectively, they represent a degree two vertex adjacent to two degree three vertices, and that has one incident edge matched (**d1i**), the same case without matched edge (**d1o**), and the cases where two degree two vertices are adjacent, and there is a matched edge (**d2i**), or not (**d2o**).

We also create the coefficients **free3**, **free4**, **free5** and **free6** that represent the paths surrounded by marked edges.

For the paths of length two, we add a digit after the coefficient that tells how many vertices of degree two are incident to the matched edges. For the paths of length one, the coefficients **g2**, **g1** and **g0** are the ones where the matched edge is incident to a degree two vertex, and the digit indicates the number of surrounding colored edges.

In conclusion, the linear program gives us that the running time of the algorithm is $\mathcal{O}(2^{1.3373n})$. This is worse than previously, we can conclude that dealing with degree two edges is harder with

this method. This is due to the fact that degree two edges gives less constraint, and then we have to branch more to color everything.

4 Conclusion

During this internship, I found two algorithms that are really different to solve the same problem of total coloring on cubic graphs. For the problem of decision and counting we improve the best known polynomial space algorithm of [Bessy and Havet, 2013]. In the first approach, the algorithm is simple, and the analysis is done using the measure and conquer method. The second algorithm is more complicated, and the analysis is done using a linear program, which, to our knowledge, has never been done. The main parts missing for the first algorithm is a clear proof of the correctness, and the extension to subcubic graphs. This is a work in progress, as well as the writing of an article for the first algorithm (with measure and conquer analysis).

References

- [Bellman, 1962] Bellman, R. (1962). Dynamic programming treatment of the travelling salesman problem. Journal of the ACM (JACM), 9(1):61–63.
- [Berglin, 2014] Berglin, E. (2014). On the performance of edge coloring algorithms for cubic graphs. Student Paper.
- [Bessy and Havet, 2013] Bessy, S. and Havet, F. (2013). Enumerating the edge-colourings and total colourings of a regular graph. Journal of Combinatorial Optimization, 25(4):523–535.
- [Bjorklund and Husfeldt, 2006] Bjorklund, A. and Husfeldt, T. (2006). Inclusion–exclusion algorithms for counting set partitions. In Foundations of Computer Science, 2006. FOCS’06. 47th Annual IEEE Symposium on, pages 575–582. IEEE.
- [Björklund et al., 2009] Björklund, A., Husfeldt, T., and Koivisto, M. (2009). Set partitioning via inclusion-exclusion. SIAM Journal on Computing, 39(2):546–563.
- [Bodlaender, 1994] Bodlaender, H. L. (1994). A tourist guide through treewidth. Acta cybernetica, 11(1-2):1.
- [Brooks, 1941] Brooks, R. L. (1941). On colouring the nodes of a network. In Mathematical proceedings of the cambridge philosophical society, volume 37, pages 194–197. Cambridge University Press.
- [Byskov et al., 2005] Byskov, J. M., Madsen, B. A., and Skjernaa, B. (2005). On the number of maximal bipartite subgraphs of a graph. Journal of Graph Theory, 48(2):127–132.
- [Courcelle, 1990] Courcelle, B. (1990). The monadic second-order logic of graphs. i. recognizable sets of finite graphs. Information and computation, 85(1):12–75.
- [Eppstein, 2004] Eppstein, D. (2004). Quasiconvex analysis of backtracking algorithms. In Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, pages 788–797. Society for Industrial and Applied Mathematics.
- [Fomin et al., 2007] Fomin, F. V., Gaspers, S., and Saurabh, S. (2007). Improved exact algorithms for counting 3-and 4-colorings. In International Computing and Combinatorics Conference, pages 65–74. Springer.
- [Fomin et al., 2009] Fomin, F. V., Grandoni, F., and Kratsch, D. (2009). A measure & conquer approach for the analysis of exact algorithms. Journal of the ACM (JACM), 56(5):25.
- [Golovach et al., 2010] Golovach, P. A., Kratsch, D., and Couturier, J.-F. (2010). Colorings with few colors: counting, enumeration and combinatorial bounds. In International Workshop on Graph-Theoretic Concepts in Computer Science, pages 39–50. Springer.
- [Held and Karp, 1962] Held, M. and Karp, R. M. (1962). A dynamic programming approach to sequencing problems. Journal of the Society for Industrial and Applied Mathematics, 10(1):196–210.

- [Koivisto, 2006] Koivisto, M. (2006). An $o^*(2^n)$ algorithm for graph coloring and other partitioning problems via inclusion–exclusion. In *Foundations of Computer Science, 2006. FOCS’06. 47th Annual IEEE Symposium on*, pages 583–590. IEEE.
- [Kostochka, 1996] Kostochka, A. V. (1996). The total chromatic number of any multigraph with maximum degree five is at most seven. *Discrete Mathematics*, 162(1-3):199–214.
- [Kowalik et al., 2008] Kowalik, Ł., Sereni, J.-S., and Škrekovski, R. (2008). Total-coloring of plane graphs with maximum degree nine. *SIAM Journal on Discrete Mathematics*, 22(4):1462–1479.
- [Petersen, 1891] Petersen, J. (1891). Die theorie der regulären graphs. *Acta Mathematica*, 15(1):193.
- [Rosenfeld, 1971] Rosenfeld, M. (1971). On the total coloring of certain graphs. *Israel Journal of Mathematics*, 9(3):396–402.
- [Sánchez-Arroyo, 1989] Sánchez-Arroyo, A. (1989). Determining the total colouring number is np-hard. *Discrete Mathematics*, 78(3):315–319.
- [Scheffler, 1986] Scheffler, P. (1986). The graphs of tree-width k are exactly partial k -trees. *manuscript*, 1986 *Linear-Time Algorithms for NP-Complete Problems Restricted to Partial k -Trees*.
- [Shen and Wang, 2009] Shen, L. and Wang, Y. (2009). Total colorings of planar graphs with maximum degree at least 8. *Science in China Series A: Mathematics*, 52(8):1733–1742.
- [Van Rooij and Bodlaender, 2011] Van Rooij, J. M. and Bodlaender, H. L. (2011). Exact algorithms for dominating set. *Discrete Applied Mathematics*, 159(17):2147–2164.
- [Vijayaditya, 1971] Vijayaditya, N. (1971). On total chromatic number of a graph. *Journal of the London Mathematical Society*, 2(3):405–408.
- [Vizing, 1964] Vizing, V. G. (1964). On an estimate of the chromatic class of a p -graph. *Diskret analiz*, 3:25–30.

A Remarks on the internship

I did my internship in Warsaw, Poland. The city is great, and really different from other European capital cities (due to the fact that it was almost completely destroyed during second world war). I had an office in the university with four Phd students. There was some seminars, I attended to a seminar about neural networks, one about matrix multiplication, and another about the implementation of algorithms, and a comparative study of the efficiency on different graphs.

B Coefficients of the linear program

This section shows all the cases and the coefficients of the linear program. In each table (Tables 3, 4, 5, and 6), there is the name of the coefficient, an illustration of the case it represents (the dotted edges are the matched edges, red edges are the ones already colored, and the blue ones are the marked ones) and the number of branches needed to color the matched edges.

There are two cases where we can have a closed formula for the number of branches the algorithm will do.

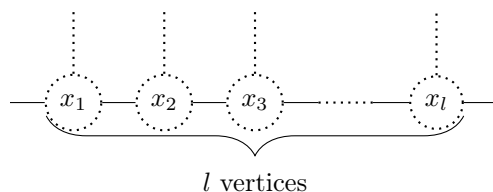


Figure 7: Allowed path with l vertices

Name of the coefficient	Illustration	Number of branching
bigpath		$12(2 \cdot 4^6 - 1)$
chord3		12
chord4		84
chord5		372
chord6		1524
cycle3		96
cycle4		156
cycle5		1200
cycle6		3948

Table 3: Number of branching to color the big allowed paths, the cycles and the chordal paths

On Figure 7 we have an allowed path of l vertices, and there are no colored edges surrounding this path (they can be marked). To color all the matched edges and the vertices x_1 to x_l we need $12 \cdot (2 \cdot 4^{l-1} - 1)$ branches.

Name of the coefficient	Illustration	Number of branching
surr6		8412
surr5		2114
surr4		540
surr3		138
semi4		892
semi3		220

Table 4: Number of branching to color the allowed paths of length greater or equal than 3

Proof. With one edge, we need $12 = 4 \times 3$ branches, for two edges we need 12×7 branches (12 branches for the first edge, and 7 for the second) because there are restrictions due to the fact that the first edge is already colored. What we remark is that on 6 of the branches for the second color, the color of the edge (x_1, x_2) is forced (there is only one choice). Then we have two choices for x_3 and two for its matched edge (4 branches in total), and again the edge (x_2, x_3) is forced. When one unmatched is forced then all the remaining matched edges need only 4 branches to be colored. On the case where the edge (x_1, x_2) is not forced (the last branch of the seven), there is seven possibilities to color x_3 and its matched edge and again on six of them the edge (x_2, x_3) is forced. This gives that the number of branches is equal to $12 \cdot (6 \cdot 4^{l-2} + 6 \cdot 4^{l-3} + \dots + 6 \cdot 4 + 7) = 12 \cdot (2 \cdot 4^{l-1} - 1)$. \square

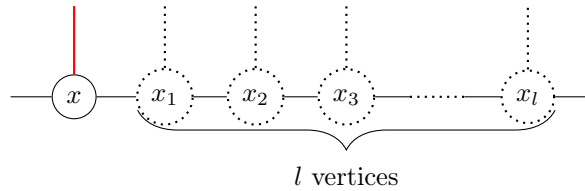


Figure 8: Allowed path with l vertices and one colored edge nearby

Figure 8 shows the second case where we have a closed form formula. To color the matched edges, we need $14 \cdot 4^{l-1} - 4$ branches for $l \geq 2$.

Proof. The edge incident to x being colored, we only need ten branches to color x_1 and its incident matched edge. On six of them, the color of the edge (x, x_1) is forced, so each of the other matched edges will be colored in four branches. Otherwise we apply the same reasoning than for the previous case. Finally there are $6 \cdot 4^{l-1} + 4(2 \cdot 4^{l-1} - 1) = 14 \cdot 4^{l-1} - 4$ branches to color the matched edges. \square

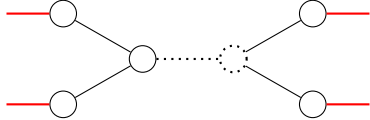
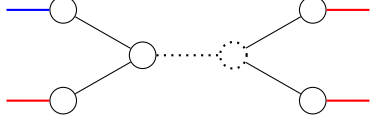
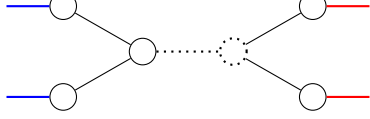
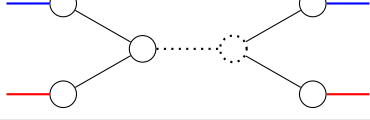
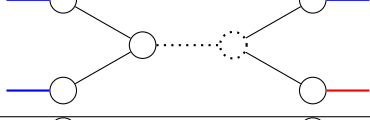
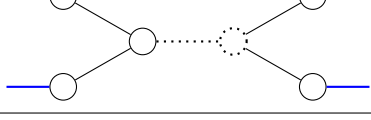
Name of the coefficient	Illustration	Number of branching
f4		8
f3		8
f2c		8
f2o		9
f1		10
f0		12

Table 5: Number of branching to color the allowed paths of length 1

C Linear program

C.1 For bridgeless cubic graphs

The linear program is given in Figure 9.

C.2 For general subcubic graph

For the algorithm where the instances are subcubic graphs, the linear program is given in Figure 10

Name of the coefficient	Illustration	Number of branching
g2a		36
g2b		36
g2c		36
g1a		52
g1b		52
g1c		50
g0a		84
g0b		77
g0c		71

Table 6: Number of branching to color the allowed paths of length 2

```

Maximize 7.10852surr3 + 9.07682surr4 + 11.04576surr5 + 13.03823surr6 + 7.78136semi3
+ 9.80090semi4 + 3.58496chord_3 + 6.39232chord_4 + 8.53916chord_5 + 10.57365chord_6
+ 16.58479bigpath + 6.58496cycle_3 + 7.28540cycle_4 + 10.22882cycle_5 + 11.94691cycle_6 + 3f4
+ 3f3 + 3f2c + 3.16993f2o + 3.32193f1 + 3.58496f0 + 5.16993g2a + 5.16993g2b + 5.16993g2c
+ 5.70044g1a + 5.70044g1b + 5.64386g1c + 6.39232g0a + 6.26679g0b + 6.14975g0c subject to

chord_3 + chord_4 + chord_5 + chord_6 - y = 0
7bigpath + chord_3 + 2chord_4 + 3chord_5 + 4chord_6
+ 3cycle_3 + 4cycle_4 + 5cycle_5 + 6cycle_6 - z = 0

0.5semi3 + 0.5semi4 + surr3 + surr4 + surr5 + surr6 - z <= 0
y - 0.5semi3 - 0.5semi4 - y2 = 0
z + 2.5semi3 + 3.5semi4 + 4.5semi5 + 2surr3 + 3surr4 + 4surr5 + 5surr6 - z2 = 0

y2 - 2g0a - 1.5g0b - 1g0c - 1.5g1a - 1g1b - 0.5g1c - 1g2a - 0.5g2b - y1 = 0
2g2c + 1.5g2b + 1g2a + 1.5g1c + 1g1b + 0.5g1a + 1g0c + 0.5g0b - z2 <= 0
z2 - 3g2c - 2.5g2b - 2g2a - 2.5g1c - 2g1b - 1.5g1a - 2g0c - 1.5g0b - g0a - z1 = 0

f3 + 2f2c + 2f2o + 3f1 + 4f0 - 2y1 <= 0
4f4 + 3f3 + 2f2c + 2f2o + f1 - 2z1 <= 0

f4 + f3 + f2c + f2o + f1 + f0 - path1 = 0
g0a + g0b + g0c + g1a + g1b + g1c + g2a + g2b + g2c - path2 = 0

y + z + path1 + 2path2 + 3semi3 + 4semi4 + 3a3 + 4a4 + 5a5 + 6a6 = 0.5

```

Figure 9: Linear program for the bridgeless cubic graphs

```

Maximize 5.16993surr2 + 7.10852surr3 + 9.07682surr4 + 11.04576surr5 + 13.03823surr6 + 5.70044b20
+ 5.70044b21 + 5.70044b22 + 7.78136semi3 + 9.80090semi4 + 11.80574semi5 + 6.39232c20 + 6.39232c21
+ 6.39232c22 + 8.53916c3 + 10.57365c4 + 12.58214c5 + 14.58426c6 + 3.58496chord_3
+ 6.39232chord_4 + 8.53916chord_5 + 10.57365chord_6 + 16.58479bigpath + 6.58496cycle_3
+ 7.28540cycle_4 + 10.22882cycle_5 + 11.94691cycle_6 + 3f4 + 3f3 + 3f2c + 3.16993f2o + 3.32193f1
+ 3.58496f0 + 3g2 + 3.32193g1 + 3.58496g0 + 0d1i + 0d1o + 0d2i + 0d2o subject to

2d1i + 2d1o + 3d2i + 3d2o <= 1

0.5d1i + 0.5d2i - x = 0
chord_3 + chord_4 + chord_5 + chord_6 + bigpath1i + d1o + d2i + d2o - y = 0
7d + chord_3 + 2chord_4 + 3chord_5 + 4chord_6 + 3cycle_3 + 4cycle_4 + 5cycle_5 + 6cycle_6 - z = 0

0.5semi2 + 0.5semi3 + 0.5semi4 + 0.5semi5 + surr2 + surr3 + surr4 + surr5 + surr6 - z <= 0
y - free3 - free4 - free5 - free6 - 0.5semi3 - 0.5semi4 - 0.5semi5 - y2 = 0
z + 3free3 + 4free4 + 5free5 + 6free6 + 0.5semi2 + 2.5semi3 + 3.5semi4 + 4.5semi5
+ 2surr3 + 3surr4 + 4surr5 + 5surr6 - z2 = 0
free3 + free4 + free5 + free6 + chord_3 + chord_4 + chord_5 + chord_6 - y <= 0
x - x2 = 0

surr2 + 0.5semi20 + semi21 + 1.5semi22 + free22 + 0.5free21 - z2 <= 0
free20 + free21 + free22 + 0.5semi20 + 0.5semi21 + 0.5semi22 - y2 <= 0
y2 + x2 - 2free20 - 1.5free21 - free22 - 1.5semi20 - semi21 - 0.5semi22 - y1 = 0

z2 + free20 + 0.5free21 + 0.5semi2 - z1 >= 0

0.5f3 + f2c + f2o + 1.5f1 + 2f0 + 0.5g2 + g1 + 1.5g0 - y1 <= 0
2f4 + 1.5f3 + f2c + f2o + 0.5f1 + g2 + 0.5g1 - z1 <= 0

f4 + f3 + f2c + f2o + f1 + f0 + g2 + g1 + g0 - path1 = 0
surr2 + semi20 + semi21 + semi22 + free20 + free21 + free22 - path2 = 0
2chord_3 + 3chord_4 + 4chord_5 + 5chord_6 - chord = 0
3cycle_3 + 4cycle_4 + 5cycle_5 + 6cycle_6 - cycle = 0

chord + cycle + 7bigpath + path1 + 2path2 + 3free3 + 4free4 + 5free5 + 6free6 + 3semi3 + 4semi4
+ 5semi5 + 3surr3 + 4surr4 + 5surr5 + 6surr6 + 0.5d1i + 0.5d1o + 1d2i + d2o = 0.5

```

Figure 10: Linear program for general graphs