Computing the Directed Cartesian-Product Decomposition of a Directed Graph from its Undirected Decomposition in Linear Time*†

Christophe Crespelle‡ Eric Thierry§

Abstract

In this paper, we design an algorithm that, given a directed graph G and the Cartesian-product decomposition of its underlying undirected graph $\tilde{G}$, produces the directed Cartesian-product decomposition of G in linear time. This is the first time that the linear complexity is achieved for this problem, which has two major consequences. Firstly, it shows that the directed and undirected versions of the Cartesian-product decomposition of graphs are linear-time equivalent problems. And secondly, as there already exists a linear-time algorithm for solving the undirected version of the problem, combined together, it provides the first linear-time algorithm for computing the directed Cartesian-product decomposition of a directed graph.

Introduction

The general idea of graph decompositions is to describe a graph as the composition, through some operations, of a set of simpler (and usually smaller) graphs. This framework has turned out to be very useful both for proving theorems (see e.g. [2]) and for solving efficiently difficult algorithmic problems using the "divide and conquer" approach (see [11]). This is the reason why, in the last decades, a lot of effort have been made for computing efficiently the decomposition of a graph with respect to a given operation. The Cartesian product of undirected graphs (graphs for short) and directed graphs (digraphs for short), usually denoted by $\Box$, is a classical and useful decomposition operation that allows to factorise some specifically structured redundancy in a graph. Such redundancy naturally appears in various contexts, both theoretic and practical, such as accountability, databases or programming. In those contexts, revealing and factorising these redundancy has a great impact on the efficiency of the solutions proposed to manage these systems.

*This work was partially supported by the Vietnam Institute for Advanced Study in Mathematics (VIASM).

†This article is a complete and improved version of the extended abstract appeared in [3].

‡Université Claude Bernard Lyon 1, DANTE/INRIA, LIP UMR CNRS 5668, ENS de Lyon, Université de Lyon – christophe.crespelle@inria.fr

§ENS de Lyon, LIP UMR CNRS 5668, Université de Lyon – eric.thierry@.ens-lyon.fr

Cartesian product has been used from the early times of graph theory, but the first intensive studies were provided by Sabidussi [10] and Vizing [12]. Both of them proved independently that any connected graph admits a decomposition into prime factors (graphs that can not be expressed as product of non-trivial graphs) which is unique up to the order of factors ($\Box$ is commutative) and up to isomorphisms (see an example on Figure 1). The unicity of the decomposition into prime factors was later extended to weakly connected digraphs by Feigenbaum [5] and Walker [13] (when graphs or digraphs are not connected, this prime decomposition is not necessarily unique). From the 80s, a series of papers have investigated the complexity of computing this prime decomposition. For connected undirected graphs, after the first polynomial algorithm by Feigenbaum et al [6] in $O(n^{4.5})$ time, where n is the number of vertices, the complexity was improved by Winkler [14], Feder [4] and Aurenhammer et al [1], until Imrich and Peterin [8] finally designed a linear-time algorithm (this is so far the only linear algorithm).

Related works. For weakly connected digraphs, the best complexity is achieved by Feigenbaum's algorithm [5] which has two steps: first it calls any algorithm that computes the prime decomposition of the underlying undirected graph (i.e. the graph obtained by replacing directed arcs by undirected edges), then by merging some factors it provides the decomposition for the digraph. The first step can be achieved with Imrich and Peterin's algorithm in time $O(n+m)$, where m is the number of edges. Then the time complexity of the second step is $O(n^2 \log^2 n)$. The problem has also been considered for restricted classes of digraphs. For connected partially ordered sets (posets), Walker describes a polynomial algorithm in [13] which also starts by factorising the graph once arcs are made undirected. Its complexity is not analysed precisely in [13], but it is not linear: it has two nested loops leading to, at least, a $\Theta(m \log^2 n)$ complexity. For sake of completeness, let us mention that, for posets, Krebs and Schmid considered a restricted version of the prime decomposition problem that only asks whether the input poset P admits a factorisation $P = P_0 \Box \mathbf{2}$, where P_0 is an arbitrary poset and $\mathbf{2}$ is the chain with two vertices. In [9], they obtain an $O(n^7)$ algorithm for solving this problem.

Our contributions. We give an $O(n+m)$ time algorithm for solving the following problem: given a directed graph G and the prime decomposition of its underlying undirected graph $\tilde{G}$, compute the prime decomposition of G with regard to the Cartesian product of *directed* graphs. This problem is at the core of the approach developed in [5] and [13] to compute the prime decomposition of a directed graph, and one of the key difficulty which did not allow them to obtain linear-time complexity. The improvement of the complexity we achieve here relies on 1) the incremental structure of our algorithm that fully takes advantage of the product structure of $\tilde{G}$ by parsing it layer by layer, and 2) a careful implementation and choice of the data-structure.

One of the most immediate benefit of our work is that, combined with the algorithm of [8] which gives a linear-time solution to the undirected problem, our algorithm provides a linear-time solution for computing the prime decomposition of directed graphs, therefore improving the complexities[1] of [5, 9, 13].

But beyond this fact, the solution we describe here has a stronger meaning. By re-

[1] And, unlike [9, 13], solving the problem not only for posets but for arbitrary digraphs G.

ducing the directed problem to the undirected one, we show that the two versions of the problem are linear-time equivalent[2], therefore clarifying the computational relationship between these two widely investigated problems. Moreover, this reduction makes the solution for the directed problem somehow independent from the solution used for the undirected one, in the sense that the algorithm computing the undirected decomposition can be modified or replaced without impacting the algorithm computing the directed decomposition: the part dealing with the orientation of arcs remains unchanged.

Outline of the paper. Section 1 presents the main definitions and theorems that we need about the Cartesian product. Section 2 describes the general scheme of our algorithm and the lemmas that will ensure its correctness. Section 3 describes the algorithm and goes into the details of data-structures and routines that are used to run it in linear time.

1 Preliminaries

In this article, we consider both directed graphs (also called digraphs), denoted $G = (V, A)$, and undirected graphs, denoted $G = (V, E)$, where V is the set of vertices of G, A is the set of arcs of G when G is a directed graph and E is the set of edges of G when G is an undirected graph. When we want to make explicit the graph or digraph referred to, we use the notations $V(G)$, $E(G)$ and $A(G)$. The neighbourhood of a vertex x in a directed or undirected graph will be denoted by $N(x)$. For a graph or a digraph G and a subset $S \subseteq V(G)$ of its vertices, we denote by $G[S]$ the graph induced by S in G. For any digraph G, we define its *underlying undirected graph* $\tilde{G}$ as the graph over the same vertices, where two vertices are adjacent if and only if there exists at least one arc between them in G. A digraph G will be called *weakly connected* if $\tilde{G}$ is connected.

After the formal definition of the Cartesian product, we state the factorisation theorem ensuring the unicity of the prime decomposition. This theorem applies to both graphs and digraphs, but it was first proved for graphs [10, 12] and then extended to digraphs [5, 13].

Definition 1 (Cartesian product of digraphs) *The Cartesian product $G = \prod_{1 \leq i \leq p} G_i$ of p directed graphs $(G_i)_{1 \leq i \leq p}$ is the directed graph $G = (V(G), A(G))$ whose vertex set is $V(G) = \prod_{1 \leq i \leq p} V(G_i)$ and such that for all $x, y \in V(G)$, with $x = (x_1, \ldots, x_p)$ and $y = (y_1, \ldots, y_p)$, we have $xy \in A(G)$ if and only if there exists $i \in [\![1, p]\!]$ such that $\forall j \in [\![1, p]\!] \setminus \{i\}, x_j = y_j$ and $x_i y_i \in A(G_i)$. The p-uple $(x_1, \ldots, x_p)$ associated with each vertex x is called the* Cartesian labelling *associated with this decomposition.*

Notation 1 ($x(i)$) *For any vertex x which received a Cartesian label, we denote by $x(i)$ the i-th coordinate of its label, that is, if x is labelled $(x_1, \ldots, x_p)$, $x(i) = x_i$.*

Definition 2 (Prime graph) *A digraph G is prime with regard to the Cartesian product iff for all digraphs G_1, G_2 such that $G = G_1 \square G_2$ then G_1 or G_2 has only one vertex.*

[2]In the sense that any of the two problems, let say P_1 (we denote P_2 the other one), can be solved by an algorithm using i) standard constant-time computation operations, a linear number of times along the whole algorithm, and ii) one single call to an algorithm solving P_2 along the whole algorithm.

The two preceding definitions are stated for digraphs but they also apply for graphs with edges instead of arcs. We now give the fundamental theorem of Cartesian product of graphs.

Theorem 1 (Unicity of the prime decomposition of digraphs [5, 13] and graphs [10, 12]) *For any weakly connected directed graph (resp. connected graph) G, there exists a unique $p \geq 1$ and a unique tuple $(G_1, \ldots, G_p)$ of digraphs (resp. graphs), up to reordering and isomorphism of the G_i's, such that each G_i has at least two vertices, each G_i is prime for the Cartesian product and $G = \prod_{i \in [\![1,p]\!]} G_i$. $(G_1, \ldots, G_p)$ is called the* prime decomposition *of G.*

A key property of the prime decomposition, stated by Theorem 2 below, is that it properly refines all other decompositions.

Definition 3 (Refinement of a decomposition) *Let G be a graph or a digraph and let $(G_1, \ldots, G_p)$, with $p \geq 1$, and $(H_1, \ldots, H_k)$, with $k \geq 1$, be two decompositions of G. We say that decomposition $(G_1, \ldots, G_p)$* refines *decomposition $(H_1, \ldots, H_k)$ iff $k \leq p$ and there exists a partition $\{I_1, \ldots, I_k\}$ of $[\![1, p]\!]$ such that $\forall j \in [\![1, k]\!], H_j = \prod_{i \in I_j} G_i$.*

Theorem 2 (Finest decomposition [7]) *Let G be a weakly connected digraph (resp. connected graph) and let $(G_1, \ldots, G_p)$, with $p \geq 1$, be its prime decomposition. If $(H_1, \ldots, H_k)$, with $k \geq 1$, is a decomposition of G such that all digraphs H_i's have at least two vertices, then $(G_1, \ldots, G_p)$* refines *$(H_1, \ldots, H_k)$.*

Cartesian product decomposition of graphs can equivalently been defined as colourings of their arcs or edges. Such colourings constitute the core of the approach of [8], and of our approach as well.

Definition 4 (Product colouring of arcs (resp. edges) [8]) *Let G be a digraph (resp. a graph) and $\mathcal{L}$ the Cartesian labelling of a decomposition of G, then the* colouring of arcs of G associated with $\mathcal{L}$ *is defined as follows: arc (resp. edge) xy is coloured with colour i iff x and y differ only on coordinate number i.*
A colouring of the arcs of a digraph G (resp. edges of graph) is called a product colouring *if it is the colouring associated to some Cartesian labelling of some decomposition of G.*

Note that the colouring associated with $\mathcal{L}$ is properly defined since each arc (resp. edge) is assigned a colour and only one. We will use the following definitions in order to express many key properties of colourings.

Definition 5 (monocoloured and bicoloured) *A set of edges or arcs is* monocoloured *iff all these edges or arcs have the same colour. Similarly, a set of vertices of a graph or a digraph is* monocoloured *iff all edges or all arcs between these vertices have the same colour.*
And a pair of edges is bicoloured *iff these two edges do not have the same colour.*

The next theorem restates Theorem 2 in terms of product colourings. It appears as Lemma 2.3 in [8] for graphs and in [5] for digraphs (stated in terms of partitions of the arcs).

Theorem 3 (Finest product colouring [8]) *Let G be a weakly connected digraph (resp. connected graph) and let C be the arc colouring associated with the prime decomposition of G, and let C' be a product colouring of G. Then, the partition of the arcs of G induced by C refines the one induced by C'. C is called the* finest product colouring *of G.*

Product colourings have strong structural properties which are characterised in the next two theorems. These are the key properties on which is based the correctness and the complexity of our algorithm. The first theorem deals with undirected graphs. It rephrases several lemmas and reasoning used by Imrich and Peterin to design and analyse their linear algorithm [8].

Theorem 4 (Square property of product colourings - undirected version [8]) *Let C be a colouring of the edges of some connected graph G. C is a product colouring iff the two following properties hold:*

1. *for any bicoloured pair $\{\{u,v\},\{v,w\}\}$ of edges of G, there exists a unique vertex v' such that $uvwv'$ is a cycle of G, and this vertex v' is such that the colour of $\{u,v'\}$ (resp. $\{v',w\}$) is the same as the one of $\{v,w\}$ (resp. $\{u,v\}$), and*

2. *for any vertices $x,y \in V(G)$, where $x \neq y$, if there exists a path from x to y using only a subset $C' \subseteq C$ of colours, then all paths from x to y must contain at least one edge whose colour belongs to C'.*

Proof. Let C be a product colouring, thus associated with a product decomposition $G = \prod_{1 \leq i \leq p} G_i$. We first show that Condition 1 holds. If edge $\{u,v\}$ has colour i and edge $\{v,w\}$ has colour j, with $j \neq i$, then it means that the coordinate labellings of vertices u and w differ on both coordinates i et j and are equal on all other coordinates. Moreover, from the definition of Cartesian product, necessarily, $\{v_i,u_i\}$ is an edge of G_i and $\{v_j,w_j\}$ is an edge of G_j. It follows that the only possibility for a vertex v' to be adjacent to both u and w is to have the same coordinates as u and w except on coordinates i and j, and on coordinates i and j, we must have $v'_i = u_i$ and $v'_j = w_j$. This implies that Condition 1 is satisfied.

Let us now examine Condition 2. If a path from x to y, with $y \neq x$, only uses edges of colours in C', it means that the labellings of the vertices of the path only differ on coordinates that belong to C'. Since $x \neq y$, their labellings are different on at least one coordinate $i \in C'$, thus any path between them must have an edge which changes this coordinate i, which gives Condition 2.

For the converse implication, one can use the *Isomorphism Lemma* of Imrich and Peterin (Lemma 2.1 in [8]) which states that if an edge colouring C satisfies the next two properties:

(i) every connected component induced by the edges of a fixed colour meets every connected component induced by the other edges in exactly one vertex and

(ii) the set of edges between two connected components induced by the edges of a fixed colour is either empty or a monocoloured perfect matching,

then C is a product colouring.

To check this, consider a colouring C satisfying both Condition 1 and 2. Let us first show that C satisfies (ii). We denote H_i the graph induced by edges of colour i in G. Let R_i and R'_i be two connected components of H_i and such that there exists an edge in G between some vertex $v_0 \in R_i$ and some vertex $v'_0 \in R'_i$, of colour j (necessarily we have $j \neq i$ since R_i and R'_i are distinct connected components for colour i). Let us prove that the set of edges in G between R_i and R'_i is a perfect matching of colour j:

- By induction on the size of a connected set $H_i[R_i]$, one can prove that any vertex $v \in R_i$ admits an incident edge $\{v, v'\}$ coloured by j where $v' \in R'_i$. Let $S \subsetneq R_i$ be a connected subset of vertices in $H_i[R_i]$ of size $k \geq 1$ such that every $x \in S$ has an incident edge coloured by j to some vertex of R'_i (consider $S = \{v_0\}$ for initialisation of the induction). As $H_i[R_i]$ is connected, there exists a vertex $u \in R_i \setminus S$ which is connected to some vertex $v \in S$ in $H_i[R_i]$. By induction hypothesis, there exists $w \in R'_i$ such that $\{v, w\}$ is an edge coloured j. Since C satisfies Condition 1, then there exists a vertex v' such that v' is linked to w by an edge of colour i (implying that $v' \in R'_i$) and v' is linked to u by an edge of colour j, which shows that the induction hypothesis holds for a connected component $S \cup \{u\}$ of $H_i[R_i]$ of size $k+1$. As a conclusion, every vertex of R_i is linked to some vertex of R'_i by an edge coloured j. And we have the symmetric property: every vertex of R'_i is linked to some vertex of R_i by an edge coloured j.

- Now consider the set of edges between R_i and R'_i of colour j. It is necessarily a perfect matching, otherwise there would exist w.l.o.g. three vertices $v \in R_i$, $v' \in R'_i$ and $v'' \in R'_i$, with $v' \neq v''$, such that $\{v, v'\}$ and $\{v, v''\}$ are edges coloured by j. But it would then create two monocoloured paths from v' to v'': $v'vv''$ coloured by j and the path coloured by i inside R'_i, thus violating Condition 2. As a consequence, the set of edges between R_i and R'_i of colour j is a matching incident to all vertices of R_i and R'_i, it is a perfect matching.

- To prove (ii), it remains to show that there is no other edge between R_i and R'_i. If there was such an edge $\{v, v'\}$, its colour k would be different from i and j, and it would contradict Condition 2 since, beside the monocoloured path from v to v' (reduced to an edge) coloured by k, we would have an alternative path without colour k, leaving v with the edge coloured by j of the perfect matching and then reaching v' thanks to a path coloured by i within R'_i.

Thus, colouring C satisfies (ii), let us now show that it also satisfies (i). Let us consider the edges of colour i, and the set $\mathcal{C}_i$ of connected components they induce in G. Now consider a connected component S induced by the edges of all colours different from i. We first show that S meets every element of $\mathcal{C}_i$ on at least one vertex. Assume for contradiction that S only meets the subset $M \subsetneq \mathcal{C}_i$ of elements of $\mathcal{C}_i$. As G is connected, there exists an edge between some element $R_i \in M$ and some element $R'_i \in \mathcal{C}_i \setminus M$. Clearly this edge is of colour $j \neq i$ and from property (ii), we know that all vertices of R_i have an incident edge of colour j to R'_i. And since R_i contains at least one vertex u of S, it follows that $u \in S$ is linked to some vertex $u' \in R'_i$ by an edge of colour $j \neq i$. Then, $u' \in S$ and S meets R'_i, which is a contradiction. Hence, S meets every connected component R_i induced by the edges of colour i in at least one vertex. But clearly, if S

intersect some R_i at two distinct vertices $u \neq v$ then there exists a path from u to v coloured by i (inside R_i) and another path from u to v without colour i (inside S), which is forbidden by Condition 2. Thus, S meets every connected component R_i induced by the edges of colour i in exactly one vertex, and colouring C satisfies property (i).

And finally, since (i) and (ii) are fulfilled, the *Isomorphism Lemma* of [8] ensures that C is indeed a product colouring. □

Remark 1 *Due to Condition 2, the cycle $uvwv'$ in Condition 1 necessarily has no chord: it is called a* square.

This characterisation can be extended to digraphs, up to adding a third minor condition (Condition 0) and carefully checking the arc orientations in Condition 1 of Theorem 4. For digraphs, a pair of adjacent vertices x, y is called *monocoloured* if all arcs between x and y (possibly two) have the same colour. We also define some types to enumerate the different cases of arc orientation between two vertices.

Definition 6 *Let G be a digraph, the type of a couple (x, y) of adjacent vertices, denoted $type(x, y)$, is:* dir *iff xy is an arc in G but not yx;* ind *iff yx is an arc in G but not xy; and* sym *iff both xy and yx are arcs in G.*

Theorem 5 (Square property of product colourings - directed version [5, 8]) *Let C be a colouring of the arcs of some weakly connected digraph G. C is a product colouring iff the three following properties hold:*

0. *all pairs of adjacent vertices are monocoloured, and*
1. *for any bicoloured pair $\{\{u, v\}, \{v, w\}\}$ of edges of $\tilde{G}$, there exists a unique vertex v' such that $uvwv'$ is a cycle of $\tilde{G}$, and this vertex v' is such that the type and the colour of (u, v') (resp. (v', w)) are the same as those of (v, w) (resp. (u, v)), and*
2. *for any vertices $x, y \in V(G)$, where $x \neq y$, if there exists a path of $\tilde{G}$ from x to y using only a subset $C' \subseteq C$ of colours, then all paths of $\tilde{G}$ from x to y must contain at least one edge whose colour belongs to C'.*

Proof. Let C be a product colouring of digraph G associated with the decomposition $G = \prod_{i=1}^{p} G_i$. By definition of the Cartesian product and the underlying undirected graph, we clearly have $\tilde{G} = \prod_{i=1}^{p} \tilde{G}_i$ (which is at the heart of all the reductions from the directed case to the undirected case). Thus C also corresponds to a product colouring of $\tilde{G}$. Thanks to Theorem 4, it yields Condition 2 and Condition 1 except the preservation of types which still has to be checked. Condition 0 forces something only for pairs of arcs where both arcs (x, y) and (y, x) exist. In that case, considering Cartesian labellings, if (x, y) has colour i, then only coordinate i changes between x and y, thus the arc (y, x) is also associated with a change only on coordinate i, it must be coloured by i. To check type preservation in squares of Condition 1, suppose that $\{u, v\}$ has colour i and $\{v, w\}$ has colour j, then the Cartesian labellings of u, v, w only differ on coordinates i and j. Moreover, by definition of the Cartesian product, $u_j = v_j$, $v_i = w_i$, $type(u, v)$ in G is the same as $type(u_i, v_i)$ in G_i, and $type(v, w)$ in G is the same as $type(v_j, w_j)$ in G_j. Since

the fourth vertex v' only differs by having coordinates $v'_i = u_i$ and $v'_j = w_j$, we have $type(u, v') = type(u_j, v'_j) = type(v_j, w_j) = type(v, w)$ and $type(v', w) = type(v'_i, w_i) = type(u_i, v_i) = type(u, v)$ which ends the proof that Condition 1 holds.

Conversely, consider an arc colouring C which satisfies all conditions and let us show that C is a product colouring. Thanks to Condition 0, it induces a non-ambiguous colouring of $\tilde{G}$. By applying Theorem 4, C is a product colouring of $\tilde{G}$ and thus associated with a factorisation $\tilde{G} = \prod_{i=1}^{p} H_i$ where all H_i's are undirected connected graphs (since $\tilde{G}$ is connected). For each $i \in [\![1, p]\!]$, we define the digraph G_i such that the undirected underlying graph of G_i is H_i and the arcs of G_i are as follows: for each pair $\{x_i, y_i\}$ of adjacent vertices in H_i, choose any arbitrary couple of vertices (x, y) of G such that $x(i) = x_i$ and $y(i) = y_i$ and all other coordinates of x and y are equal, and set the arcs between x_i and y_i in G_i such that $type(x_i, y_i) = type(x, y)$. We now show that $G = \prod_{i=1}^{p} G_i$. In other words, we simply aim at proving that all couples (x, y) of G such that $x(i) = x_i$ and $y(i) = y_i$ and all other coordinates of x and y are equal have the same type. Let $S = \{x \in V(G) \mid x(i) = x_i\}$ and let $x_0 \in S$. For any vertex $x \in S$, we define $p_y(x)$ as the vertex of V having the same coordinates as x except on coordinate i where $p_y(x)$ has coordinate y_i. Clearly $G[S]$ is weakly connected as $\tilde{G}[S]$ is isomorphic to $\prod_{j \neq i} H_j$, and the H_i's are connected. Consider the maximum weakly connected (in G) subset X of vertices of S containing x_0 and such that $\forall x \in X, type(x, p_y(x)) = type(x_0, p_y(x_0))$. Proving that $X = S$ will achieve our goal.

Assume for contradiction that $X \neq S$. Since S is weakly connected, there exists some vertex $x' \in S \setminus X$ such that x' has a neighbour $x \in X$. Pair $\{x, p_y(x)\}$ is coloured i while pair $\{x, x'\}$ is not coloured i, as both $x, x' \in S$ and have therefore the same i-coordinate. Then, Condition 1 applies and the unique vertex v whose existence is stated by this condition is $p_y(x')$, which gives $type(x', p_y(x')) = type(x, p_y(x))$. And since, by definition of X, we have $type(x, p_y(x)) = type(x_0, p_y(x_0))$, it follows that $x' \in X$, which is a contradiction. As a consequence, $X = S$ and all couples (x, y) of G such that $x(i) = x_i$ and $y(i) = y_i$ and all other coordinates of x and y are equal have indeed the same type. Thus, G is the product of the G_i's we defined, which shows that C is a product colouring and achieves the proof. □

We will make an intensive use of Theorem 5 to prove the correctness of our approach.

2 Our approach

Like [8], our approach consists in computing the finest product colouring $\mathcal{C}_G$ of the arcs of G (see Theorem 3), which is precisely the one corresponding to the prime decomposition of G. To that purpose, we first colour the arcs of G according to the finest product colouring $\mathcal{C}_{\tilde{G}}$ (which is taken as input by our algorithm) of the undirected underlying graph $\tilde{G}$ of G, then, we merge some classes of colours into one single colour in order to obtain $\mathcal{C}_G$. Our main result is to show that the classes of colours to be merged can be computed in linear time with regard to the size of G, and that the labels of the vertices can be updated in linear time as well during these merges. The fact that one can always proceed by merging some colours of the undirected colouring of $\tilde{G}$ is stated by Lemma 1 below. Before, in the following remark, we note that there is a natural correspondence

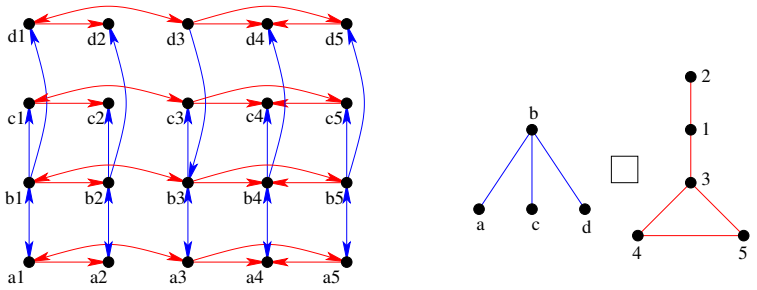

Figure 1: A directed graph G. Thanks to the layout, the vertex labels and the edge colouring, one should be convinced that the underlying undirected graph $\tilde{G}$ is the Cartesian product of the two graphs on the right (which are prime). Nonetheless G with its orientation of edges is prime: the directed edge $(d3, b3)$ does not fit the orientation of the related edges $(d1, b1)$, $(d2, b2)$, $(d4, b4)$.

between some colourings of a digraph G and the colourings of its underlying undirected graph $\tilde{G}$. We make extensive use of this correspondence in the end of this section.

Remark 2 *Any colouring $\tilde{\mathcal{C}}$ of the undirected underlying graph $\tilde{G}$ of some digraph G naturally induces a colouring $\mathcal{C}$ of the arcs of G by assigning to each arc (u, v) the colour of edge $\{u, v\}$ in $\tilde{\mathcal{C}}$. And conversely, a colouring of G such that two opposite arcs between vertices u, v are always assigned the same colour induces a univoquely defined colouring of $\tilde{G}$.*

Lemma 1 *Let G be a digraph, let $\mathcal{C}_G$ be the finest product colouring of G, and let $\mathcal{C}_{\tilde{G}}$ be the finest product colouring of $\tilde{G}$. We denote $\mathcal{C}_{dir}$ the colouring of the arcs of G induced by $\mathcal{C}_{\tilde{G}}$. Then, $\mathcal{C}_{dir}$ is finer than $\mathcal{C}_G$.*

Proof. From Condition 0 of Theorem 5, in a directed product colouring, two symmetric arcs are always assigned the same colour. Then, each directed product colouring $\mathcal{C}$ of G induces a colouring $\tilde{\mathcal{C}}$ of the edges of $\tilde{G}$. It turns out that this colouring $\tilde{\mathcal{C}}$ of $\tilde{G}$ is also a product colouring: indeed, if $\mathcal{C}$ satisfies the conditions of Theorem 5 to be a directed product colouring then $\tilde{\mathcal{C}}$ satisfies the conditions of Theorem 4 to be an undirected product colouring (which are weaker). Now consider the undirected product colouring $\mathcal{C}_{undir}$ of $\tilde{G}$ induced by $\mathcal{C}_G$. Since $\mathcal{C}_{undir}$ is a product colouring of $\tilde{G}$, from Theorem 3, $\mathcal{C}_{undir}$ is coarser than $\mathcal{C}_{\tilde{G}}$. And Since $\mathcal{C}_{dir}$ is the colouring of G induced by $\mathcal{C}_{\tilde{G}}$, it follows that, in G, $\mathcal{C}_G$ is coarser than $\mathcal{C}_{dir}$. □

In order to design an algorithm, we must be able to determine which classes of colours have to be merged in $\mathcal{C}_{dir}$ to obtain the finest product colouring $\mathcal{C}_G$ we aim at computing. We will show (Lemma 3 below) that $\mathcal{C}_G$ can be obtained by merging all the pairs of colours that are *conflicting* in $\mathcal{C}_{dir}$.

Definition 7 (Conflicting colours and colour-conflict graph) *Let G be a digraph and let $\mathcal{C}$ be a colouring of its arcs such that all pairs of vertices of G are monocoloured. Two colours c_1, c_2 of $\mathcal{C}$ are* conflicting *iff there exists some bicoloured pair*

$\{\{u,v\},\{v,w\}\}$ *of edges of* $\tilde{G}$ *such that* $\{u,v\}$ *is coloured by* c_1 *and* $\{v,w\}$ *is coloured by* c_2 *and* $\{\{u,v\},\{v,w\}\}$ *does not satisfy Condition 1 of Theorem 5.*
The colour-conflict graph *of a colouring* $\mathcal{C}$ *is the graph whose vertex set is the colours of* $\mathcal{C}$ *and whose edges are the pairs of colours conflicting in* $\mathcal{C}$*.*

Lemma 2 below shows that the bicoloured pairs of vertices $\{\{u,v\},\{v,w\}\}$ giving rise to a conflict on their colours are strongly structured: they necessarily belong to the set of properly coloured squares of $\mathcal{C}_{dir}$. In other words, the only reason why a conflict may appear between two colours of $\mathcal{C}_{dir}$ is because of the orientation of arcs. This is a key property which we use in the proof of Lemma 3 and in the description of our algorithm, allowing it to remain conceptually simple and to run in linear time.

Lemma 2 *Let* G *be a digraph, let* $\mathcal{C}_{\tilde{G}}$ *be the finest product colouring of* $\tilde{G}$*, and let* $\mathcal{C}_{dir}$ *be the colouring of the arcs of* G *induced by* $\mathcal{C}_{\tilde{G}}$*. If two colours are conflicting in* $\mathcal{C}_{dir}$ *on some bicoloured pair* $\{\{u,v\},\{v,w\}\}$*, then*

1. *there exists a unique vertex* v' *such that* $uvwv'$ *is a cycle of* $\tilde{G}$*, and this vertex* v' *is such that in* $\mathcal{C}_{dir}$*,* $colour(u,v') = colour(v,w)$ *and* $colour(v',w) = colour(u,v)$*, but*
2. $type(u,v') \neq type(v,w)$ *or* $type(v',w) \neq type(u,v)$*.*

Proof. Since pair $\{\{u,v\},\{v,w\}\}$ is bicoloured in $\mathcal{C}_{dir}$ and since $\mathcal{C}_{dir}$ is induced from $\mathcal{C}_{\tilde{G}}$, then pair $\{\{u,v\},\{v,w\}\}$ is bicoloured in $\mathcal{C}_{\tilde{G}}$. And since $\mathcal{C}_{\tilde{G}}$ is a product colouring, then, from Condition 1 of Theorem 4, Condition 1 of Lemma 2 is satisfied. But since bicoloured pair $\{\{u,v\},\{v,w\}\}$ is conflicting, by definition, it does not satisfy Condition 1 of Theorem 5. Thus, necessarily, we have $type(u,v') \neq type(v,w)$ or $type(v',w) \neq type(u,v)$. □

Lemma 3 below claims that the classes of colours of $\mathcal{C}_{dir}$ that have to be merged into a single colour in order to obtain the finest product colouring $\mathcal{C}_G$ of G are precisely the connected components of the colour-conflict graph G_{conf} of $\mathcal{C}_{dir}$.

Lemma 3 *Let* G *be a digraph and let* $\mathcal{C}_{dir}$ *be the colouring of* G *induced by the finest product colouring of* $\tilde{G}$*. Consider the colour-conflict graph* G_{conf} *of* $\mathcal{C}_{dir}$*. Then, the colouring* $\mathcal{C}_{conf}$ *of the arcs of* G *obtained by merging in* $\mathcal{C}_{dir}$ *each connected component of* G_{conf} *into one single colour is the finest product colouring* $\mathcal{C}_G$ *of* G*.*

Proof. First, we show that $\mathcal{C}_{conf}$ is finer than $\mathcal{C}_G$. To this purpose, we just have to show that if two arcs e_1, e_2 have the same colour in $\mathcal{C}_{conf}$ then, they have the same colour in $\mathcal{C}_G$.
If e_1 and e_2 have the same colour in $\mathcal{C}_{dir}$, since from Lemma 1, $\mathcal{C}_{dir}$ is finer than $\mathcal{C}_G$, then e_1 and e_2 have the same colour in $\mathcal{C}_G$ as well.
Otherwise, if e_1 and e_2 have the same colour in $\mathcal{C}_{conf}$ but are coloured with two distinct colours in $\mathcal{C}_{dir}$, denoted respectively a and b, then, necessarily, a and b belong to the same connected component of G_{conf}. Then, there exists a sequence $c_1, \ldots, c_k$, with $k \geq 2$, of pairwise distinct colours of $\mathcal{C}_{dir}$ such that $c_1 = a$ and $c_k = b$ and for any $i \in [\![1, k-1]\!]$, $c_i c_{i+1} \in E(G_{conf})$.
Consider some $i \in [\![1, k-1]\!]$. Since c_i and c_{i+1} are conflicting in $\mathcal{C}_{dir}$, from Definition 7, there exists some pair $\{\{u,v\},\{v,w\}\}$ of edges of $\tilde{G}$ such that $\{u,v\}$ is coloured c_i and

$\{v, w\}$ is coloured c_{i+1} and $\{\{u, v\}, \{v, w\}\}$ is conflicting. From Lemma 2, it follows that there exists a unique vertex v' such that u, v, w, v' is a cycle in $\tilde{G}$, and this vertex v' is such that $type(u, v') \neq type(v, w)$ or $type(v', w) \neq type(u, v)$. Then, in any colouring of the arcs of G where pair $\{\{u, v\}, \{v, w\}\}$ is bicoloured, the colours of $\{u, v\}$ and $\{v, w\}$ are conflicting. And consequently, since $\{\{u, v\}, \{v, w\}\}$ is not conflicting in $\mathcal{C}_G$, it follows that pair $\{\{u, v\}, \{v, w\}\}$ is not bicoloured in $\mathcal{C}_G$. Then, since from Lemma 1, $\mathcal{C}_G$ is coarser than $\mathcal{C}_{dir}$, necessarily all arcs coloured c_i or c_{i+1} in $\mathcal{C}_{dir}$ have the same colour in $\mathcal{C}_G$. As this holds for any $i \in [\![1, k-1]\!]$, it follows that all arcs coloured $c_1 = a$ and all arcs coloured $c_k = b$ in $\mathcal{C}_{dir}$ have the same colour in $\mathcal{C}_G$. In particular, e_1 and e_2 have the same colour in $\mathcal{C}_G$, which we wanted to prove. Thus, $\mathcal{C}_{conf}$ is finer than $\mathcal{C}_G$.

We now prove that $\mathcal{C}_{conf}$ is a product colouring. Remind that $\mathcal{C}_{conf}$ is obtained from $\mathcal{C}_{dir}$ by merging some subsets of colours into a single one. Note that since $\mathcal{C}_{dir}$ is induced from $\mathcal{C}_{\tilde{G}}$ it necessarily satisfies Conditions 0 and 2 of Theorem 5, which actually concerns $\tilde{G}$ rather than G itself. Also observe that merging colours preserves these conditions, and then $\mathcal{C}_{conf}$ satisfies Conditions 0 and 2 of Theorem 5 as well. As a consequence, in order to show that $\mathcal{C}_{conf}$ is a product colouring, we just need to show that it satisfies Condition 1 of Theorem 5. Consider an arbitrary bicoloured pair $\{\{u, v\}, \{v, w\}\}$ in $\mathcal{C}_{conf}$. It is necessarily bicoloured in $\mathcal{C}_{dir}$ (since $\mathcal{C}_{conf}$ is by definition coarser than $\mathcal{C}_{dir}$) and satisfies Condition 1, because otherwise it would generate a conflict between the two concerned colours and consequently these two colours would have been merged in $\mathcal{C}_{conf}$, that is $\{\{u, v\}, \{v, w\}\}$ would not be bicoloured in $\mathcal{C}_{conf}$. Moreover, observe that if a bicoloured pair satisfies Condition 1 of Theorem 5, it also satisfies this condition in any coarser colouring. As a consequence, since, by definition, $\mathcal{C}_{conf}$ is coarser than $\mathcal{C}_{dir}$ and since $\{\{u, v\}, \{v, w\}\}$ satisfies Condition 1 in $\mathcal{C}_{dir}$, then it satisfies this condition in $\mathcal{C}_{conf}$ as well. Finally, since all the three conditions of Theorem 5 are satisfied, it follows that $\mathcal{C}_{conf}$ is a product colouring. And since it is finer than $\mathcal{C}_G$, it is the finest product colouring of G, that is $\mathcal{C}_{conf}$ and $\mathcal{C}_G$ are the same, up to renaming colours. □

3 Algorithm

Our algorithm takes as input the adjacency lists of the digraph G as well as the prime decomposition of the underlying graph $\tilde{G}$ of G, given in the form described below (Section 3.1), in particular including the finest product colouring of the edges of $\tilde{G}$. It then outputs the finest product colouring of the arcs of G together with the corresponding Cartesian labelling of the vertices of G. It operates in two steps:

1. *Conflicting pairs of colours*: list all pairs of colours that are conflicting in $\mathcal{C}_{dir}$ (see Lemma 1) and build its colour-conflict graph G_{conf}.

2. *Merge*: merge each connected component of G_{conf} into one single colour and update the labels of the vertices of G accordingly.

In this section, we deals with implementation details and show how to perform the two steps above in linear time with regard to the size of G. We start by precisely describing the data structure we use (both for input and output of our algorithm), called *Cartesian*

representation, which is an implementation of adjacency lists respecting the product structure of the graph.

3.1 Cartesian representation

Given a product colouring $\mathcal{C}$, using colours $\{1, \ldots, p\}$, of a digraph G (or a product colouring of its underlying undirected graph $\tilde{G}$) and the corresponding Cartesian labelling $V = \prod_{i \in [\![1,p]\!]} V_i$ of its vertices, we encode the digraph G, its colouring $\mathcal{C}$ and the Cartesian labels of its vertices into a data-structure that we call the *Cartesian representation* of G. It is very similar to the classical adjacency lists, except that the lists of neighbours of the vertices are stored in a matrix M_{cart} instead of a one-dimensional array.

For all $i \in [\![1, p]\!]$, we denote $n_i = |V_i|$. The vertices in V_i are numbered from 1 to n_i so that the label $(x_1, \ldots, x_p)$ of a vertex x belongs to $[\![1, n_1]\!] \times \ldots \times [\![1, n_p]\!]$. M_{cart} is an $n_1 \times \ldots \times n_p$ matrix indexed by the labels of the vertices of G and such that the cell $M_{cart}(x_1, \ldots, x_p)$ contains two fields storing the information of the vertex x whose label is $(x_1, \ldots, x_p)$: the first field is simply the label $(x_1, \ldots, x_p)$ of x, and the second field is a one dimensional array denoted $N(x)$ and indexed by the p colours of colouring $\mathcal{C}$, from 1 to p. For any $i \in [\![1, p]\!]$, the cell indexed i of $N(x)$ contains the list $N_i(x)$ of neighbours y of x such that the arcs between x and y are coloured i. Each cell of the list $N_i(x)$ corresponding to a neighbour y of x again contains two fields: the first one is $type(x, y)$ (see Definition 6) and the second one is a pointer to the cell $M_{cart}(y_1, \ldots, y_p)$ of M_{cart}, where $(y_1, \ldots, y_p)$ is the label of y. Moreover, in the Cartesian representation, each list $N_i(x)$ is stored sorted by increasing value of the i-th component y_i of the neighbours $y = (y_1, \ldots, y_i, \ldots, y_p)$ of x it contains. Note that since only the component y_i changes in the list $N_i(x)$, the order defined on $N_i(x)$ by the i-th component of the label is exactly the Cartesian order on the label of vertices of $N_i(x)$.

Finally, let us mention that the reason why we use a pointer to $M_{cart}(y_1, \ldots, y_p)$ instead of the label $(y_1, \ldots, y_p)$ of y, in the cell of y in the list $N_i(x)$, is that reading one pointer takes constant time while reading a sequence of integer takes a time proportional to the length of the sequence ($\Omega(p)$ time in this case). Then, storing a pointer instead of the label allows to access in constant time to the cell $M_{cart}(y_1, \ldots, y_p)$ containing the information of y. In addition, if the label of y also needs to be accessed, it can be done in constant time as well, as it is stored in the first field of cell $M_{cart}(y_1, \ldots, y_p)$. These features are necessary in order to achieve linear time.

Now that we precisely described the data-structure we use, we can go through the description of the two steps of our algorithm, which takes as input the Cartesian representation of G according to the finest product colouring $\mathcal{C}_{\tilde{G}}$ of its underlying undirected graph and outputs the Cartesian representation of G according to its finest product colouring $\mathcal{C}_G$. To that purpose, in the rest of the paper, we use the following notations.

Notation 2 *Let G be a directed graph, let $\tilde{G}$ be its underlying undirected graph and let $\mathcal{C}_{\tilde{G}}$ be the finest product colouring of the edges of $\tilde{G}$. Then, we denote $\mathcal{C}_{dir}$ the colouring of the arcs of G induced by the colouring $\mathcal{C}_{\tilde{G}}$ of the edges of $\tilde{G}$. And we denote M_{dir} the matrix M_{cart} of the Cartesian representation of G according to $\mathcal{C}_{\tilde{G}}$.*

List-conflicts(G)

1. $conflicts \leftarrow [\,]$
2. **For** $x_{i,j}$ in Cartesian order, with $i \geq 2$ and $j \geq 2$ **Do**
3. Compute partition $\mathcal{P}(Y_{i,j})$
4. $cross\_type\_colours \leftarrow$ `Crossing-colours`$(M_{cart}(G[Y_{i,j}]), \mathcal{P}(Y_{i,j}))$
5. **For** $colour \in cross\_type\_colours$ **Do** $conflicts \leftarrow conflicts.[(i, colour)]$
6. $layer\_type\_conflicts \leftarrow$ `List-conflicts`$(G[Y_{i,j}])$
7. $conflicts \leftarrow conflicts.layer\_type\_conflicts$
8. **End of for**
9. **Return** $conflicts$

Figure 2: Routine `List-conflicts`: the outline of our algorithm for listing conflicting pairs of colours. Note the incremental scheme implemented by the loop at line 2, the recursive calls on the different layers encountered during the execution (line 6) and the call to Routine `Crossing-colours`, which we describe further in Section 3.2.2. Lists are denoted between brackets, $[\,]$ stands for the empty list and symbol . denotes the concatenation operation.

3.2 Conflict-detection step

This step of the algorithm outputs the list (eventually with repetitions) of all pairs of conflicting colours. This is the most challenging part of the algorithm as the conflicts may occur on any bicoloured square of $\mathcal{C}_{dir}$. And it turns out that the number of such squares may be superlinear[3] in the number of arcs of G, while we aim at achieving a linear complexity. The key point is that we can actually detect all the conflicts between colours without parsing all the squares of $\mathcal{C}_{dir}$, but only a certain subset of them, whose size is linear in the size of the graph. To that purpose, we take advantage of the product structure computed for $\tilde{G}$, which we parse incrementally layer by layer, each layer being processed in linear time with regard to the size of the layer plus the size of its connections to the previously treated layers.

Our algorithm has two passes: the first one computes all the pairs (c_1, c_2) of colours, with $c_2 > c_1$, conflicting on some square because of the orientations of arcs of colour c_2 (remind that from Lemma 2, the only possibility for conflicts to appear in $\mathcal{C}_{dir}$ is because of the orientation of arcs, not because of colours); then we reverse the order on colours and run the same algorithm, in order to list the conflicts occurring because of the orientation of arcs of the lower colour as well. Note that reversing the order of colours can be done easily in linear time on our data-structure as follows. First, for every vertex x, we reverse the sequence $(x_1, \ldots, x_p)$ constituting the label of x and we reverse the array $N(x)$ storing the coloured lists of neighbours of x. Doing so for all vertices takes $O(pn) = O(m)$ time. Then, we have to reorder the matrix M_{dir}, which is storing the Cartesian representation of G, according to the reverse order on the labels. Namely, for every label $(x_1, \ldots, x_p)$,

[3]Consider for example a graph G such that $\tilde{G} = K_k \square K_k$. The number of bicoloured squares of $\mathcal{C}_{dir}$ is then $\Omega(k^4)$, while the number of arcs in G is only $O(k^3)$.

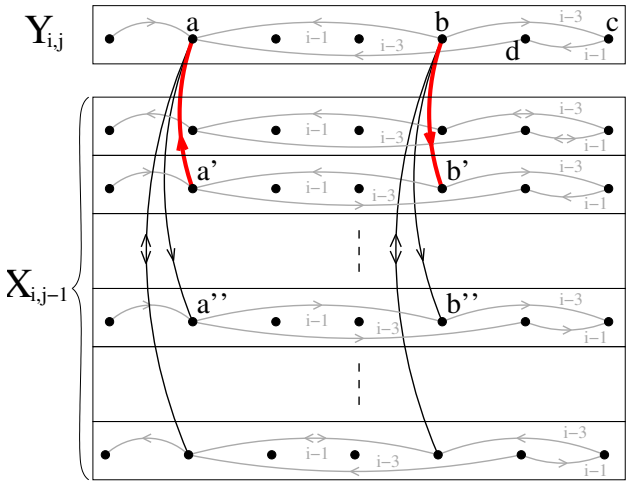

Figure 3: The incremental scheme of our algorithm. The arcs of colours less than i are depicted in grey, and the arcs of colour i in black (except aa' and bb' that are red and bold). Square $abcd$ is a layer-type conflicting square, because of the orientations of bc and ad. It will be detected during the recursive call to Routine `List-conflicts` on $G[Y_{i,j}]$. Square $abb'a'$ is a cross-type conflicting square, because of the orientations of aa' and bb'. This conflict will be detected by our algorithm since a and b are linked by an arc and will not be in the same part of partition $\mathcal{P}$. On the opposite, the conflict between colour $i-1$ and i on the square $abb''a''$ will not be detected in the first pass of the algorithm, as it occurs because of the orientation of arcs of the lower colour $i-1$; though, it will be detected in the second pass after reversing the order on colours.

we must swap the content of cells $M_{dir}(x_1, \ldots, x_p)$ and $M_{dir}(x_p, \ldots, x_1)$. This can be done by a simple parse of M_{dir}, which takes $O(pn) = O(m)$ time (as reading one label takes $O(p)$ time). Thus, the Cartesian representation of the graph where the order on the colours has been reversed can be computed in linear time. We now concentrate on one single pass of the algorithm, which is done by Routine `List-conflicts` of Figure 2.

As previously, we denote $V = \prod_{i \in [\![1,p]\!]} V_i$ the Cartesian labelling of the vertices of G, with $n_i = |V_i|$. For each $i \in [\![1,p]\!]$, we number the vertices of V_i from 1 to n_i. For $i \in [\![1,p]\!]$ and $j \in [\![1,n_i]\!]$, we denote $x_{i,j}$ the vertex of V_i numbered j and we denote $N_<(x_{i,j}) = \{x_{i,k} \mid k < j \text{ and } x_{i,j} \text{ and } x_{i,k} \text{ are adjacent in } G_i\}$, and $d_<(x_{i,j}) = |N_<(x_{i,j})|$. We also denote $Y_{i,j} = \{x = (x_1, \ldots, x_p) \mid x_i = j \text{ and } \forall k > i, x_k = 1\}$, and $X_{i,j} = \bigcup_{l \leq j} Y_{i,l}$ (see Figure 3). Finally, $n_{<i}$ and $m_{<i}$ respectively stand for the number of vertices and the number of adjacent pair of vertices in $G[Y_{i,j}]$. Note that, from the definition of the Cartesian product, these numbers do not depend on j. For example, for any $i \geq 2$, we have $n_{<i} = \prod_{j<i} n_j$.

Our algorithm is incremental in the sense that it starts with the set of vertices $\{x_{1,j}, j \in [\![1,n_1]\!]\} \cup \{x_{i,1}, i \in [\![1,p]\!]\}$, and considers the vertices $(x_{i,j})$ of the G_i's, with $i \geq 2$ and $j \geq 2$, one by one by increasing (i,j) for the Cartesian order (see the loop at line 2 of Routine `List-conflicts`). Before adding $x_{i,j}$, list $conflicts$ already contains all the pairs of colours conflicting on some square of $G[X_{i,j-1}]$, and when adding $x_{i,j}$, our algorithm extends this list with all the pairs (c_1, c_2) of colours, with $c_2 > c_1$, conflicting on some square of $G[X_{i,j}]$ (because of the orientation of c_2-coloured arcs) that involves some vertex

of $Y_{i,j}$ (line 3 to 7 of Routine `List-conflicts`). Lemma 4 below states that there are only two types of such squares (see Figure 3).

Lemma 4 *Let $a, b, b', a' \in X_{i,j}$ such that a, b, b', a' is a conflicting square and $\{a, b, b', a'\} \not\subseteq X_{i,j-1}$. Then, exactly one of the two following conditions holds:*

1. layer-type square condition: *$a, b, b', a' \in Y_{i,j}$, or*
2. cross-type square condition: *up to renaming vertices, $a, b \in Y_{i,j}$ and $a', b' \in X_{i,j-1}$ and ($\forall k \neq i, a'_k = a_k$ and $b'_k = b_k$) and ($a'_i = b'_i < j$).*

Proof. If the layer-type square condition is not satisfied for a, b, b', a', then, since $\{a, b, b', a'\} \not\subseteq X_{i,j-1}$, it follows that $\{a, b, b', a'\} \cap Y_{i,j} \neq \varnothing$ and $\{a, b, b', a'\} \cap X_{i,j-1} \neq \varnothing$. Since there is a conflict on square a, b, b', a', from Lemma 2, square a, b, b', a' is properly bicoloured in $C_{\tilde{G}}$. Then edges ab and aa' does not have the same colour. Up to renaming vertices, we can assume without loss of generality that vertex a is in $Y_{i,j}$ and edge aa' is coloured i. Note that, in the product colouring $\mathcal{C}_{\tilde{G}}$ of $\tilde{G}$, all the edges between $Y_{i,j}$ and $X_{i,j-1}$ are necessarily coloured i while the edges between vertices of $Y_{i,j}$ are not coloured i. As a consequence, vertex a' is in $X_{i,j-1}$ while vertex b is in $Y_{i,j}$. And since edge aa' is coloured i, from the definition of Cartesian product, we have $\forall k \neq i, a'_k = a_k$. Since square a, b, b', a is properly bicoloured in $\mathcal{C}_{\tilde{G}}$, then edge bb' is also coloured i, so we have $\forall k \neq i, b'_k = b_k$. Finally, since bb' is coloured i and $b \in Y_{i,j}$, as mentioned above, b' is necessarily in $X_{i,j-1}$. And since $a'b'$ is an edge and is not coloured i, there must exist $j' < j$ such that a' and b' are both in $Y_{i,j'}$, that is $a'_i = b'_i = j'$. And thus, the cross-type square condition is satisfied for a, b, b', a', which ends the proof of the lemma. □

Layer-type conflicts are computed by a recursive call to Routine `List-conflicts` on $G[Y_{i,j}]$, at line 6, and then added to the list of conflicts, line 7. Thus, in the next section, we focus on computing cross-type conflicts (line 3 to 5 of Routine `List-conflicts`), which constitutes the essential part of the incremental scheme of our algorithm.

3.2.1 Cross-type conflicts

From Lemma 4, if a, b, b', a' is a cross-type square, then pairs $\{a, a'\}$ and $\{b, b'\}$ are necessarily coloured i, while pairs $\{a, b\}$ and $\{a', b'\}$ have colour at most $i - 1$. Then, all cross-type squares raise a conflict between colour i and another colour i' less than i, and our goal is to list all such colours i', for which Lemma 5 gives a characterisation. Before stating this characterisation which will be at the core of our algorithm, we need to define a particular partition $\mathcal{P}$ of the vertices of $Y_{i,j}$. Lemma 5 will sate that the colours conflicting with colour i on some cross-type squares are exactly the colours of the arcs of $G[Y_{i,j}]$ crossing partition $\mathcal{P}$.

Definition 8 *Partition $\mathcal{P}$ of the vertices of $Y_{i,j}$ is made of the equivalence classes of the binary relation defined as follows: two vertices $y, z \in Y_{i,j}$, with $y = (y_1, \ldots, y_{i-1}, j, 1, \ldots, 1)$ and $z = (z_1, \ldots, z_{i-1}, j, 1, \ldots, 1)$ are equivalent iff for all $j' < j$ such that $x_{i,j'}$ and $x_{i,j}$ are adjacent in G_i, we have type$(y, y') =$ type(z, z'), where $y' = (y_1, \ldots, y_{i-1}, j', 1, \ldots, 1)$ and $z' = (z_1, \ldots, z_{i-1}, j', 1, \ldots, 1)$.*

Note that the previous definition is valid because for any vertex $y \in Y_{i,j}$, from the definition of Cartesian product, the set $X_{i,j-1} \cap N(y)$ of its neighbours in $X_{i,j-1}$ is exactly the subset of vertices $y' \in X_{i,j-1}$ such that $\forall l \neq i, y'(l) = y(l)$ and $y'(i) \in \{j' \mid j' < j$ and $x_{i,j'}$ is adjacent to $x_{i,j}$ in $G_i\}$. In particular, note that all the vertices of $Y_{i,j}$ have the same number of neighbours in $X_{i,j-1}$, which is $d_<(x_{i,j})$. The fact that the relation defined above is an equivalence relation simply follows from the fact that equality of types is so. We are now ready to give the key characterisation on which our algorithm leans.

Lemma 5 *Colour $i' < i$ conflicts with colour i on some cross-type square because of orientation of arcs coloured i iff there exists some arc coloured i' in $G[Y_{i,j}]$ having its two extremities in two distinct classes of partition $\mathcal{P}$.*

Proof. If there is a i'-coloured arc between two vertices a and b that are in different parts of partition $\mathcal{P}$, then, from definition of $\mathcal{P}$, there exists $j' < j$ and $a', b' \in Y_{i,j'}$ such that $abb'a'$ is a square and $type(a, a') \neq type(b, b')$. Then, $abb'a'$ is a square which is conflicting because of orientation of the arcs between pair $\{a, a'\}$ and between pair $\{b, b'\}$, which are coloured i.

Conversely, if there exists a cross-type square a, b, b', a' involving colour $i' < i$, then necessarily the arcs between a and b is coloured i'. Moreover, from the definition of cross-type square, a' and b' both belong to $Y_{i,j'}$ for some $j' < j$. Denoting $a = (a_1, \ldots, a_{i-1}, j, a_{i+1}, \ldots, a_p)$ and $b = (b_1, \ldots, b_{i-1}, j, b_{i+1}, \ldots, b_p)$, it follows that $a' = (a_1, \ldots, a_{i-1}, j', a_{i+1}, \ldots, a_p)$ and $b' = (b_1, \ldots, b_{i-1}, j', b_{i+1}, \ldots, b_p)$. And since a, b, b', a' is conflicting because of orientation of arcs of colour i, we also have $type(a, a') \neq type(b, b')$. As a consequence, a and b do not belong to the same part of $\mathcal{P}$. Since they are linked by an arc coloured i', this arc crosses partition $\mathcal{P}$, which ends the proof. □

As a consequence of Lemma 5, in order to detect conflicts occurring on cross-type squares, our algorithm stores for each vertex of $Y_{i,j}$ the identifier $P(x)$ of the class of partition $\mathcal{P}$ to which it belongs. A natural way to list all the colours of arcs crossing partition $\mathcal{P}$ would be to parse all the arcs of $G[Y_{i,j}]$ and to check whether their two end vertices have been assigned the same label. Unfortunately, for complexity reasons, we cannot check all arcs of $G[Y_{i,j}]$, as an arc xy may then be examined in p distinct recursive calls along the algorithm (once for each colour i such that $x(i) = y(i) \neq 1$), thus leading to a $O(pm)$ time complexity, which is not linear. To avoid this superlinear cost, we will show how to list all colours crossing partition $\mathcal{P}$ in $O(n_{<i})$ time, which will guarantee a linear complexity for the whole algorithm, as shown in the complexity analysis of Section 3.2.2 below. To that purpose, we need a characterisation of the colours crossing partition $\mathcal{P}$ that does not rely on edges but rather on vertices of graph $G[Y_{i,j}]$. This is the purpose of Lemma 6 below, for which we need to introduce the two following definitions.

Definition 9 *For any vertex $x \in V$ such that $x = (x_1, \ldots, x_p)$ and any colour i such that $x(i) \neq 1$ we denote $f_i(x)$ the unique vertex $y \in V$ such that $\forall j \in [\![1, p]\!] \setminus \{i\}, y(j) = x(j)$ and $y(i) = x(i) - 1$.*

Definition 10 *For any vertex $x \in V$, we denote $P(x)$ the part of $\mathcal{P}$ to which x belongs, and we say that vertex $x \in V$ is a* breaking point *for colour i iff $x(i) \neq 1$ and $P(f_i(x)) \neq P(x)$.*

Lemma 6 *Let $G = (V, E)$ be a graph together with a Cartesian labelling of its vertices and a product colouring of its edges. And let $\mathcal{P}$ be a partition of the vertices of G. A colour $i \in [\![1, c]\!]$ crosses partition $\mathcal{P}$ iff there exists $x \in V$ which is a breaking point for i.*

Proof. If there exists $x \in V$ which is a breaking point for i, then, from the definition of a breaking point, x and $f_i(x)$ do not belong to the same part of partition $\mathcal{P}$. Now consider the set $R_i(x) = \{y \in V \mid \forall j \neq i, y(j) = x(j)\}$. From the definition of Cartesian product, $G[R_i(x)]$ is isomorphic to G_i and is therefore connected. As x and $f_i(x)$ do not belong to the same part of partition $\mathcal{P}$, there necessarily exists an arc between two vertices of $G[R_i(x)]$ such that one extremity is in the same part of $\mathcal{P}$ as x and the other extremity is in a different part of partition $\mathcal{P}$. Since all arcs of $G[R_i(x)]$ are coloured i, it follows that colour i crosses partition $\mathcal{P}$.

Conversely, if i has no breaking point in V, then consider some vertex $x \in V$ such that $x(i) = 1$. Let y be the vertex of V such that $y(i) = n_i$ and $\forall j \neq i, y(j) = x(j)$. We have $R_i(x) = \{f_i^k(y)\}_{0 \leq k \leq n_i - 1}$, where the exponent on f_i stands for composition of functions. And consequently, since i has no breaking points, then all vertices in $R_i(x)$ are in the same part of $\mathcal{P}$ as y and none of the edges between them crosses partition $\mathcal{P}$. As this holds for any vertex x such that $x(i) = 1$, it follows that no edges of colour i cross partition $\mathcal{P}$. □

From Lemma 6, one can determine the colours crossing partition $\mathcal{P}$ without examining the edges of the graph. Nevertheless, we cannot check whether x is a breaking point for colour i for all colours and all vertices, as this would result in an $O(pn) = O(m)$ complexity, and we aim at achieving only $O(n)$ computation time for this step of the algorithm. Then one of the main challenge of our implementation is to check only $O(n)$ couples of vertex and colour, in such a way that if a colour i has a breaking point, we will check at least one couple x, i where x is a breaking point for i. We now show how to do so and give a more complete complexity analysis of our algorithm.

3.2.2 Complexity of Routine `List-conflicts`

Before getting into the details of the implementation to obtain a linear time complexity for this step of the algorithm, we first make a coarse grain analysis. There are two key points to obtain a linear complexity for Routine `List-conflicts`. The first one is to compute partition $\mathcal{P}$ (line 3) and assign its partition label to each vertex of $Y_{i,j}$ in $O(n_{<i} d_<(x_{i,j}))$ time. And the second one is to list all colours crossing partition $\mathcal{P}$ (line 4) in $O(n_{<i})$ time. If we can achieve these two tasks within these complexities, then we obtain a linear complexity for listing conflicting colours.

Indeed, note that in Routine `List-conflicts`, the sets of edges between $Y_{i,j}$ and $X_{i,j-1}$ involved in the different recursive calls to the Routine are all disjoint. Since computing partition $\mathcal{P}$ takes a time linear with regard to this number of edges, then the total time of computing all partitions $\mathcal{P}$ that have to be computed along the algorithm is $O(m)$.

Similarly, in the incremental scheme of Routine `List-conflicts(G)`, a vertex x of G can belong to at most one set $Y_{i,j}$ and will then be involved in at most one call to Routine `Crossing-colours` and at most one recursive call to `list-conflicts`. It follows that the total number of times when vertex x will be involved in some call to Routine `Crossing-colours` along the algorithm is bounded by the depth of the tree of recursive calls to `list-conflicts`. As the dimension of the product graph on which is

called Routine `list-conflicts` decreases by exactly one at each call, the depth of this tree is bounded by the number of colours p. Then, the total complexity of all calls to Routine `Crossing-colours` along the whole algorithm is $O(pn) = O(m)$ time.

Thus, provided that we can compute partition $\mathcal{P}$ and list all colours that crosses it within the desired complexity, we will obtain a linear time algorithm for computing all pairs of conflicting colours.

Computing partition $\mathcal{P}(Y_{i,j})$ To that purpose, we proceed as follows. We build an $n_1 \times \ldots \times n_{i-1}$ matrix T indexed by the vertices y of $Y_{i,j}$ and where each cell contains a one-dimensional array $T(y)$ indexed from 1 to $d_<(x_{i,j})$. Then, for each vertex $y \in Y_{i,j}$ we parse the vertices $z \in N(y) \cap X_{i,j-1}$ in increasing order of their Cartesian label (remind that this is the order in which they appear in M_{dir}) and we set the corresponding cell of $T(y)$ to $type(y, z)$. This takes $O(n_{<i}d_<(x_{i,j}))$ time. Then partition $\mathcal{P}$ is exactly the partition where the classes are the subsets of vertices having the same vector $T(y)$. We compute this partition $\mathcal{P}$ by bucket-sorting the vertices of $y \in Y_{i,j}$ according to the value of $T(y)$, with $T(y)(1)$ as primary key, $T(y)(2)$ as secondary key, and so on. One pass, for one key, takes $O(n_{<i})$ time since there are only 3 different values for the key: *dir*, *ind* and *sym*. Thus, the total cost of the bucket-sort is $O(n_{<i}d_<(x_{i,j}))$. At the end, we assign its partition label to each vertex x of $Y_{i,j}$, which is made of the identifier of the class to which vertex x belongs, that is an integer between 1 and $n_{<i}$. The total time needed to do so for all vertices of $Y_{i,j}$ is $O(n_{<i}d_<(x_{i,j}))$.

Listing colours crossing partition $\mathcal{P}(Y_{i,j})$: Routine `Crossing-colours` The second key point for obtaining a linear complexity is to be able to list all colours crossing partition $\mathcal{P}(Y_{i,j})$ (or simply $\mathcal{P}$ for short) in $O(n_{<i})$ time. This is the purpose of Routine `Crossing-colours` described on Figure 4 for which we prove validity and $O(n_{<i})$ time complexity. Routine `Crossing-colours` computes the list *non_broken* of colours that do not cross partition $\mathcal{P}$ and returns the complementary list $[1, c] \setminus non\_broken$ of colours that do cross partition $\mathcal{P}$. The validity of Routine `Crossing-colours` directly follows from the invariant stated by Lemma 8. Lemma 7 below states that during the algorithm, when x is not a breaking point for some colour i in *non_broken*, we do not need to check the colours of *non_broken* greater than i. This property plays a key role in the proof of validity (Lemma 8) and is also at the core of the complexity of Routine `Crossing-colours`: roughly speaking, this allows to check only one colour per vertex, resulting in an $O(n_{<i})$ time complexity.

Lemma 7 *Let $x \in V$ and let $NB(X^-)$ be the set of colours that have no breaking points in $X^- = \{y \in V \mid y <_{cart} x\}$, where $<_{cart}$ denotes the Cartesian order on the labels of vertices. If x is not a breaking point for some colour $i \in NB(X^-)$ such that $x(i) \neq 1$, then x is not a breaking point for any colour $j \in NB(X^-)$ such that $j > i$.*

Proof. Let $j \in NB(X^-)$ and $j > i$. By definition, if $x(j) = 1$ then x is not a breaking point for colour j. Let us consider the case where $x(j) \neq 1$. Note that both $f_i(x)$ and $f_j(x)$ are in X^-. Since i and j are in $NB(X^-)$, it follows that $f_i(x)$ and $f_j(x)$ are not breaking points for respectively colour j and i. So we have $P(f_j(f_i(x))) = P(f_i(x))$ and $P(f_i(f_j(x))) = P(f_j(x))$. Moreover, from the definition of f_i's (see Definition 9), we have

Crossing-colours$(M_{cart}(G), \mathcal{P}(V))$

1. $x \leftarrow (1, 1, \ldots, 1)$
2. $non\_broken \leftarrow [1, 2, \ldots, c]$
3. $i \leftarrow c$
4. **While** $non\_broken \neq \varnothing$ and $x \leq (n_1, n_2, \ldots, n_c)$ **Do**
5. **If** $x(i) = 1$
6. **Then** $x \leftarrow x + 1$
7. **Else**
8. **While** $i \neq \perp$ and $P(f_i(x)) \neq P(x)$ **Do**
9. $i \leftarrow succ(i, non\_broken)$
10. $remove(pred(i, non\_broken), non\_broken)$
11. **While** $i \neq \perp$ and $x(i) = 1$ **Do**
12. $i \leftarrow succ(i, non\_broken)$
13. **End of while**
14. **End of while**
15. **If** $i = \perp$ **Then** $i \leftarrow max(non\_broken)$
16. $x \leftarrow x + 1$
17. **If** $x(i) = 1$ **Then If** $pred(i, non\_broken) \neq \perp$ and $x(pred(i, non\_broken)) \neq 1$
18. **Then** $i \leftarrow pred(i, non\_broken)$
19. **Else** $i \leftarrow max(non\_broken)$
20. **End of while**
21. Return list $[1, c] \setminus non\_broken$

Figure 4: Parse of the matrix $M_{cart}(G)$ of the Cartesian representation of a graph G to list colours crossing partition $\mathcal{P}(V)$ of the set of vertices V of G. At each stage list *non_broken* contains the colours for which no breaking point has been discovered by the routine so far. Then, at the end of the routine, when all vertices have been considered, *non_broken* is exactly the list of colours having no breaking point, that is colours not crossing partition $\mathcal{P}$ according to Lemma 6. And the routine returns the complementary list $[1, c] \setminus non\_broken$ of colours that do cross $\mathcal{P}$. When e is an element of list l, we denote $succ(e, l)$ and $pred(e, l)$ respectively the successor and predecessor of e in l. When e is the last element of list l, denoted $max(l)$, by convention, $succ(e, l) = \perp$, and similarly, when e is the first element of l, $pred(e, l) = \perp$. And we also use the convention $pred(\perp, l) = max(l)$.

$f_j(f_i(x)) = f_i(f_j(x))$. As a consequence, we obtain that $P(f_i(x)) = P(f_j(x))$. Since x is not a breaking point for i, $P(f_i(x)) = P(x)$ and therefore $P(f_j(x)) = P(x)$, which ends the proof of the lemma. □

We are now ready to state the invariant which will prove the validity of Routine `Crossing-colours`. There are two parts in the invariant. The first one claims that the list *non_broken* computed at each step of the execution of the routine is indeed the list of colours that have no breaking points in the set X^- of the vertices visited so far. This implies that at the end of its execution, when $X^- = V$, the routine has computed the list *non_broken* of all colours that have no breaking points, that is the list of colours that do not cross partition $\mathcal{P}$ (see Lemma 6). The second part of the invariant is a technical condition that ensures that the colour i currently considered for breaking-point test by the routine is indeed the right colour to test, in the sense that if the test is successful (i.e. x is not a breaking point for this colour) then one does not need to test any other colour.

Lemma 8 *Every time the algorithm performs the test of line 4, we have the following invariant, which has two conditions:*

(a) list non_broken contains exactly the set of colours $NB(X^-)$ that have no breaking point in $X^- = \{y \in V \mid y < x\}$, and

(b) colour i is the smallest colour in list non_broken such that $x(i) \neq 1$, if there exist such a colour in non_broken, and i is the maximum colour of non_broken otherwise.

Proof. The initialising step (lines 1 to 3) guarantees that the invariant is satisfied at the beginning of the first iteration of the main loop. We now show that if the invariant is satisfied at the beginning of some iteration of the main loop (at line 4), then it is again satisfied at the end of this iteration (line 20), and so at the beginning of the next iteration.

If $x(i) = 1$ (*Then* member of line 6), since Condition (b) is satisfied at the beginning of the main loop, necessarily, all the colours j in *non_broken* are such that $x(j) = 1$. Then, by definition, no colour in *non_broken* can have x as breaking point, and the list *non_broken* do not need to be updated. Moreover, after the increment of x, the only colour j in *non_broken* which may be such that $x(j) \neq 1$ is the greatest colour of *non_broken*, which is current colour i from Condition (b). So colour i do not need to be updated after the increment of x.

In the case where $x(i) \neq 1$, we distinguish two parts in the code: 1) the secondary loop (from line 8 to 14) whose aim is to ensure that Condition (a) will be satisfied after the next increment of x and 2) lines 15 to 19 which increments x and updates i so that Condition (b) remains satisfied.

Let us start with Condition (a). So that this condition is still satisfied after the next increment of x, the secondary loop (lines 8 to 14) removes from list *non_broken* all the colours j such that the current vertex x is a breaking point for j. From Condition (b) of the invariant, before the first execution of the secondary loop, $\forall j \in$ *non_broken* such that $j < i$, we have $x(j) = 1$, and thus x is not a breaking point for j. This is the reason why the secondary loop parses only the colours $j \geq i$ in increasing order and executes two

kinds of operation: i) when a colour j such that x is a breaking point for j is encountered, colour j is removed from list *non_broken* (lines 9 and 10) and ii) when a colour j such that $x(j) = 1$ is encountered, colour j is simply skipped (inner loop of lines 11 to 13). When the secondary loop stops we either have $i = \perp$ or $x(i) \neq 1$ and x is not a breaking point for colour i. If $i = \perp$, then all the list *non_broken* has been parsed and if $x(i) \neq 1$ and x is not a breaking point for colour i, then, from Lemma 7, x is not a breaking point for any colour of *non_broken* greater than i. As all colours less than i such that x is a breaking point for them have been removed during the parse, it follows that after the execution of the secondary loop, all colours $j \in$ *non_broken* such that x is a breaking point for j have been removed from list *non_broken*. As a consequence, as list *non_broken* is not modified in the rest of the main loop (lines 15 to 19) and as x is incremented, Condition (a) of the invariant is still satisfied at the beginning of the next iteration of the main loop.

Now that we ensured that Condition (a) will be satisfied after the increment of x, the purpose of lines 15 to 19 is to increment x and update the value of i so that Condition (b) remains satisfied. As we mentioned previously, at the end of the secondary loop (between lines 14 and 15), we either have $i = \perp$ and all colours $j \in$ *non_broken* are such that $x(j) = 1$, or we have i is the least colour of *non_broken* such that $x(i) \neq 1$, and $P(f_i(x)) = P(x)$.

If $i = \perp$, then i is set to the maximum colour of *non_broken* (line 15). Afterwards, Condition (a) and (b) of the invariants are satisfied and $x(i) = 1$, exactly like when the test of line 5 succeeds. This is why at line 16, x is simply incremented, like at line 6, and the invariant is still satisfied after this increment (see case $x(i) = 1$ at the beginning of this proof).

If $i \neq \perp$ then $x(i) \neq 1$ and we also increment x at line 16. After the increment, we test whether $x(i) = 1$ (line 17). If the test fails, that is we still have $x(i) \neq 1$ after the increment, then for all the colours $j < i$, the value of $x(j)$ has not changed during the increment. In particular, all such colours j that are in *non_broken* are still such that $x(j) = 1$, as, from Condition (b), it was the case before the increment. Then, i still satisfy Condition (b) after the increment and does not need to be updated. That is why there is no *Else* member in the conditionnal of line 17.

On the other hand, if the test $x(i) = 1$ at line 17 succeeds, since before the increment $x(i)$ was different from 1 and became equal to one after, then for all colours $j \geq i$ in *non_broken*, after the increment, $x(j) = 1$. Moreover, before the increment, from Condition (b), all colours $j < i$ in *non_broken* were such that $x(j) = 1$. Then, if one of its colour do not satisfy this condition anymore after the increment, it is necessarily the colour preceding i in *non_broken*, and it is the only colour j of *non_broken* such that $x(j) \neq 1$. This is the reason why, if the second test of line 17 succeeds, then i is set to its predecessor in list *non_broken* (line 18). Otherwise, all colours j in *non_broken* are such that $x(j) = 1$, and in this case, i is set to the maximum colour of list *non_broken* (line 19). Thus, the invariant is still satisfied at the end of the iteration, with the new value of x, and at the beginning of the next iteration as well. □

The validity of Routine `Crossing-colours` directly follows from Lemma 8. From the test of the main loop at line 4, there are two reasons why the routine may stop. The first one is when *non_broken* $= \varnothing$. In this case, all colours have been removed from

list *non_broken*, which contains all colours at the beginning of the routine. As a colour is removed only when a breaking point is found for it, it follows that list *non_broken*, which is empty, indeed contains all colours that do not have any breaking point, that is colours that crosses partition $\mathcal{P}$ according to Lemma 6. As the routine outputs the complementary list $[1, c] \setminus non\_broken$, then its result is correct in this case.

The second reason for the routine to stop is when the label of x is greater than $(n_1, n_2, \ldots, n_c)$. In this case, as x is incremented only by one at each iteration of the main loop, we have $X^- = V$. Thus, Lemma 8 Condition (a) ensures that list *non_broken* contains exactly the colours that have no breaking point in V, that is those colours that do not cross partition $\mathcal{P}$ (see Lemma 6). And the routine also outputs the correct result in this case. Thus, Routine `Crossing-colours` is valid. Let us now analyse its time complexity and prove that it is $O(n)$, where n is the number of vertices of the graph G on which is called Routine `Crossing-colours`$(M_{cart}(G), \mathcal{P}(V))$.

Complexity analysis of Routine `Crossing-colours` Assume for now that we can access $f_i(x)$ in constant time, which we will show at the end of this analysis. Then, all atomic instructions of the algorithm are computed in constant time. Thus, the only concern about its complexity is the number of times each loop will be executed. There are two kinds of execution of the main loop: those that execute the secondary loop of lines 8 to 14 at least once, which we call *breaking executions*, and those that do not execute it at all, which we call *non-breaking executions*. Non-breaking executions contain a constant number of atomic instructions and at least one increment of x. It follows that the total cost of computation for non-breaking executions of the main loop during the algorithm is $O(n)$.

The number of breaking executions cannot exceed the number of colours p since, by definition, a breaking execution executes at least one iteration of the secondary loop and then removes at least one colour from list *non_broken* (line 10) which is non-augmenting during the algorithm. Then, we only need to bound the execution time of a breaking execution, and as the execution of lines 15 to 19 always takes constant time, our concern is to bound the number of executions of the secondary loop (lines 8 to 14) and inner loop (lines 11 to 13). To this purpose, note that each execution of the secondary loop and each execution of the inner loop contains an assignment of colour i to its successor in list *non_broken*. Moreover, the secondary and inner loop do not contain any other instruction assigning i and no instruction augmenting list *non_broken*. Therefore, the total number of executions of the secondary loop and of the inner loop (inside one given breaking execution of the main loop) is bounded by the number of colours p. And since the rest of the instructions contained in the secondary loop are atomic, it follows that the execution time of the secondary loop in one breaking execution is always bounded by $O(p)$. Since the number of breaking executions during all the routine is $O(p)$, we obtain that the total cost of all breaking executions is $O(p^2)$. And it follows that the cost of executing Routine `Crossing-colours` is $O(n + p^2) = O(n + \log^2 n) = O(n)$ time, provided that the computation of $f_i(x)$ can be implemented in constant time.

We now focus on this point and show how to compute, once and for all at the beginning of the conflict-detection step of the algorithm, a data-structure that given a colour i and a pointer toward $M_{dir}(x)$ provides a pointer to $M_{dir}(f_i(x))$. This data-structure is actually not independent of M_{dir}: we simply augment each cell $M_{dir}(x)$ with an array

$f(x)$ indexed from 1 to p (p being the number of colours), where cell indexed i contains a pointer to cell $M_{dir}(f_i(x))$ of M_{dir}. We show how to do so in $O(pn) = O(m)$ time, which keeps the complexity of our algorithm linear, as we compute this structure only once in the whole algorithm. We proceed as follows. We first compute for each $i \in [\![2, c]\!]$, the number $n_{<i} = \prod_{1 \leq j \leq i} n_j$ of vertices in graph $G_1 \times \ldots \times G_{i-1}$, and we use by convention $N_{<1} = 1$. Then, we initialise a one-dimensional array T indexed from 1 to n and we parse matrix M_{dir} in Cartesian order. For each vertex x encountered, having label $(x_1, \ldots, x_p)$, we store a pointer to $M_{dir}(x)$ in the cell indexed $index(x)$ of T, where $index(x) = 1 + \sum_{1 \leq i \leq p}(x_i - 1)n_{<i}$. And we also store $index(x)$ in cell $M_{dir}(x)$. Clearly, from the definition, function $index$ is a bijection from V onto $[\![1, n]\!]$. The interest of using function $index$ is that given a colour i such that $x(i) \neq 1$ and given the index $index(x)$ of a vertex x, the index of $f_i(x)$ is given by $index(f_i(x)) = index(x) - n_{<i}$, which can be computed in constant time. Then, after having filled array T, we parse again matrix M_{dir} and for each vertex x and each colour i we set the cell indexed i of array $f(x)$ to the value $T(index(x) - n_{<i})$, which is precisely the pointer toward cell $M_{dir}(f_i(x))$ of M_{dir}. The time needed to initialise array T with all indices is $O(pn)$, as each index can be computed in $O(p)$ operations, and the time needed to build the arrays $f(x)$ for all x is also $O(pn)$ as each array is of size p and the value contained in each of its cell is computed in constant time.

Then, the total complexity of computing this data structure is $O(pn) = O(m)$ time, and thanks to the array $f(x)$ stored in $M_{dir}(x)$, the cell $M_{dir}(f_i(x))$ of $f_i(x)$ can be accessed in constant time. As explained above, it follows that the time complexity of Routine `Crossing-colours` is $O(n)$.

3.2.3 Conclusion of the conflict-detection step

We have shown that Routine `List-conflicts` list all conflicts on pair of colours occurring because of orientation of arcs of the higher colour in $O(n + m)$ time. This Routine has to be ran twice: once with the normal order on colours and once with the reverse order. In this way, we obtain all conflicts and not only those occurring because of the higher colour. The result we obtain is the list of pairs of colours that are conflicting, eventually with repetitions. This list contains at most p^2 distinct values and its length is bounded by $O(n + m)$, as it is computed in $O(n + m)$ time. Therefore, we can obtain the list without repetitions by bucket sorting the list in $O(n + m + p^2) = O(n + m)$ time. We can then build the colour-conflict graph G_{conf} on the set of colours and parse it in order to obtain its connected components, which are the classes of colours we need to merge in $\mathcal{C}_{dir}$ in order to obtain $\mathcal{C}_G$ (see Lemma 3). This takes $O(p^2) = O(n)$ time. Then, as a conclusion, at the end of the conflict-detection step of our algorithm (Step 2), we obtain the list of connected components of G_{conf}, each of which is itself given by the list of colours it contains sorted by increasing order.

3.3 Merging step

In the previous step, we computed the classes of colours $C_l, l \in [\![1, q]\!]$ that have to be merged in order to obtain the finest product colouring of G. For any $l \in [\![1, q]\!]$ we denote $C_l = \{l_1, \ldots, l_{|C_l|}\} \subseteq [\![1, p]\!]$, with $l_1 < l_2 < \ldots < l_{|C_l|}$. We also denote $G = \prod_{i \in [\![1,q]\!]} G'_l$ the

prime decomposition of G, and we denote $n'_l = n_{l_1} n_{l_2} \dots n_{l_{|C_l|}}$ the number of vertices of G'_l. In order to obtain the Cartesian representation of G according to the finest colouring $\mathcal{C}_G$ of its arcs, we need to achieve three tasks: i) update the labelling of vertices of G and rearrange the matrix storing the adjacency lists accordingly, ii) for each vertex x merge its lists of neighbours that now belong to the same class of colours C_l and iii) sort the obtained lists of neighbours according to the Cartesian order on the labels of their vertices.

To achieve Task ii), we first build an array $Newcolour$ of size p that contains, for each $i \in [\![1, p]\!]$, the index l of the class of colours C_l to which colour i belongs. Then, we parse the array $N(x)$ of the Cartesian representation and we merge the lists $N_i(x)$ that now belong to the same class of colours C_l, into a new array $N(x)$ indexed from 1 to q. To achieve Task iii), we use the classical technique to sort adjacency lists of a graph G w.r.t. a given order σ on the vertices of G (here, σ is the Cartesian order on the labels) in linear time: 1) initialise a new copy of the adjacency lists of G with all lists empty and 2) for each vertex x of G considered in increasing order w.r.t. σ, parse its list of neighbours (the order of parsing does not matter) and for each vertex y encountered, append x at the end of the list of y in the new copy of the adjacency lists. Here, in addition to this classical technique, we simply have to respect the splitting of the lists with respect to the colours of the arcs, that is place x in the list $N_i(y)$, where i is the colour of pair $\{x, y\}$. Therefore, both Tasks ii) and iii) need only linear computation time. We now focus on Task i).

We aim at computing the matrix of the new Cartesian representation, following the new labels defined by the colouring $\mathcal{C}_G$ of the arcs of G. To that purpose, we first need to compute some matrices and arrays making the corres-pondences between ancient and new labels of the vertices and between ancient and new names of colours. First, we number the vertices of the graphs G'_l of the prime decomposition as follows. For each $l \in [\![1, q]\!]$, we build a $n_{l_1} \times \dots \times n_{l_{|C_l|}}$ matrix $NewName_l$ where cell $NewName_l(a_1, \dots, a_{|C_l|})$ contains the rank of $(a_1, \dots, a_{|C_l|})$ in the Cartesian order on $[\![1, l_1]\!] \times \dots \times [\![1, l_{|C_l|}]\!]$. From matrix $NewName_l$, we also compute the converse association array $AncName_l$ of size n'_l where cell $AncName_l(k)$ contains the $|C_l|$-tuple $(a_1, \dots, a_{|C_l|}) \in [\![1, l_1]\!] \times \dots \times [\![1, l_{|C_l|}]\!]$ such that $NewName_l(a_1, \dots, a_{|C_l|}) = k$. The t-th component a_t of $AncName_l(k)$ is denoted $AncName_l(k)(t)$. For the correspondence between new and ancient colours, we will make use of the array $Newcolour$ defined above plus another array $Newrank$ that contains, for each colour $i \in [\![1, p]\!]$, the rank of i in the ordered list C_l of its new class of colours (i.e. $Newcolour(i)$). All matrices $NewName_l$ and arrays $AncName_l$, for all l, can be computed in $O(n+q+np) = O(m)$ time, while arrays $Newcolour$ and $Newrank$ requires only $O(p)$ computation time. Thus, the total time complexity of building all matrices and arrays we need in the following is $O(m)$.

To achieve Task i), we build a $n'_1 \times \dots \times n'_q$ matrix M_{new} to store the adjacency lists of G organised according to its prime decomposition, that is the colouring $\mathcal{C}_G$ of its arcs. Then, for each $x' = (x'_1, \dots, x'_q) \in [\![1, n'_1]\!] \times \dots \times [\![1, n'_q]\!]$ we store in cell $M_{new}(x'_1, \dots, x'_q)$ the new label $(x'_1, \dots, x'_q)$ of vertex x' and the array $M_{dir}(x_1, \dots, x_p)$ containing the neighbours of vertex x', where $(x_1, \dots, x_p)$ is the former label of vertex x' in the Cartesian representation M_{dir} (see Section 3.1). To that purpose, we only need to compute the x_i's, for $i \in [\![1, p]\!]$, from the x'_j, $j \in [\![1, q]\!]$. This can be done thanks to arrays $Newcolour, NewRank$ and arrays $AncName_l$ as follows: $x_i = AncName_s(x'_s)(t)$ with $s = Newcolour(i)$ and $t =$

$NewRank(i)$. For each vertex x', writing its new label takes $O(q)$ time, and computing its former label takes $O(p)$ time. Then, the total time needed to achieve Task i), including the construction of matrix M_{new}, is $O(n + (p + q)n) = O(n + m)$, which is also the total complexity of the merging step of our algorithm.

3.4 Conclusion

As a general conclusion of our algorithm, each of its two steps runs in $O(n+m)$ time, which is then the total complexity of our algorithm for computing the prime decomposition of a directed graph G from the prime decomposition of its underlying undirected graph $\tilde{G}$. Within this complexity, our algorithm outputs the corresponding Cartesian labelling of vertices of G, the finest product colouring of the arcs of G, as well as the Cartesian representation of G according to this colouring, which is a very useful data-structure for all algorithms willing to exploit the product structure of G.

4 Computing the prime decomposition of a directed graph

As we mentionned in the introduction, combined with the algorithm of [8], the algorithm we present here gives the first linear-time algorithm to compute the prime decomposition of a directed graph G, taking as input only its adjacency lists. To that purpose, one simply need to first run [8]'s algorithm on $\tilde{G}$ and then gives the decomposition of $\tilde{G}$ as input to our algorithm. The only thing that needs to be carefully checked (this is the purpose of this section) is that, from the output of [8]'s algorithm, one can obtain the suitable format for the input of our algorithm, namely the Cartesian representation of G with regard to the product colouring $\mathcal{C}_{\tilde{G}}$ of $\tilde{G}$, in linear time.

Before this, in order to apply [8]'s algorithm, we need to compute the undirected adjacency lists of $\tilde{G}$, from the directed lists of G, which require only linear time. In the same time, we can also compute the types of all adjacent couples of vertices, which is needed for the Cartesian representation of G. We store these types into the cells of the adjacency lists, as done for the Cartesian representation.

The adjacency lists of $\tilde{G}$ are then given as input to [8]'s algorithm, which computes the prime decomposition $\tilde{G} = \prod_{i \in [\![1,p]\!]} \tilde{G}_i$ (as usual we denote $n_i = |V(\tilde{G}_i)|$). The algorithm gives the corresponding Cartesian labelling of the vertices of G and the colouring $\mathcal{C}_{\tilde{G}}$ of the edges of $\tilde{G}$. More explicitly, it produces a data-structure that, given a vertex, provides its label in constant time and, given an edge, provides its colour in constant time. Using this data-structure, we build the Cartesian representation of G with regard to $\mathcal{C}_{\tilde{G}}$ (described in Section 3.1). First, we build a matrix M_{dir} indexed by the labels of the vertices and for each x, we place its label $(x_1, \ldots, x_p)$ in the first field of $M_{dir}(x_1, \ldots, x_p)$, while in the second field, we build a one dimensional array $N(x)$ of size p and initialise each of its cells with an empty list. This takes $O(np) = O(m)$ time. Then, we parse the adjacency lists of $\tilde{G}$ and for each neighbour y of x, we determine the colour i of pair $\{x, y\}$ and place y at the beginning of list $N_i(x)$. Remember that the cell of y in the lists of $\tilde{G}$ also contains the type of (x, y), which we copy in the Cartesian representation. Since the colours are given in constant time by [8]'s data structure, this can be done in linear time.

Then, in order to complete the Cartesian representation we have to sort all the coloured adjacency lists $N_i(x)$, for every i and x, according to the Cartesian labels of the vertices they contain. This can again be done in linear time using the classical technique described in Section 3.3 to achieve Task iii). Thus, the total time needed to obtain the Cartesian representation of G according to colouring $\mathcal{C}_{\tilde{G}}$ from the output of [8]'s algorithm is linear. And finally, as our algorithm and the one of [8] both run in linear time, the prime decomposition of a directed graph with regard to the Cartesian product can be computed from its adjacency lists in linear time.

Acknowledgements

We warmly thank Thomas Lambert for his participation to the preliminary version of this work and Aniela Popescu for her useful comments and suggestions and for proofreading a draft of this article.

References

[1] F. Aurenhammer, J. Hagauer, and W. Imrich. Cartesian graph factorization at logarithmic cost per edge. *Computational Complexity*, 2:331–349, 1992.

[2] Maria Chudnovsky, Neil Robertson, Paul Seymour, and Robin Thomas. The strong perfect graph theorem. *Annals of Mathematics*, 164(1):51229, 2006.

[3] Christophe Crespelle, Eric Thierry, and Thomas Lambert. A linear-time algorithm for computing the prime decomposition of a directed graph with regard to the cartesian product. In *COCOON 2013*, volume 7936 of *LNCS*, pages 469–480, 2013.

[4] T. Feder. Product graph representations. *Journal of Graph Theory*, 16:467–488, 1992.

[5] J. Feigenbaum. Directed cartesian-product graphs have unique factorizations that can be computed in polynomial time. *Discr. Appl. Math.*, 15:105–110, 1986.

[6] J Feigenbaum, J. Hershberger, and A.A. Schäffer. A polynomial time algorithm for finding the prime factors of cartesian-product graphs. *Discrete Applied Mathematics*, 12:123–138, 1985.

[7] R. Hammack, W. Imrich, and S. Klavzar. *Handbook of Product Graphs.* CRC Press, 2011.

[8] W. Imrich and I. Peterin. Recognizing cartesian products in linear time. *Discrete Mathematics*, 307(3-5):472–483, 2007.

[9] M. Krebs and J. Schmid. Ordering the order of a distributive lattice by itself. *Journal of Logic and Algebraic Programming*, 76:198–208, 2008.

[10] G. Sabidussi. Graph multiplication. *Mathematische Zeitschrift*, 72(1):446–457, 1960.

[11] J. P. Spinrad. *Efficient graph representations*, volume 19 of *Fields Institute Monographs*. American Mathematical Society, 2003.

[12] V. G. Vizing. The cartesian product of graphs. *Vyisl. Sistemy*, 9:30–43, 1963.

[13] J. W. Walker. Strict refinement for graphs and digraphs. *Journal of Combinatorial Theory Series B*, 43(2):140–150, 1987.

[14] P. M. Winkler. Factoring a graph in polynomial time. *European Journal on Combinatorics*, 8:209–212, 1987.