A linear-time algorithm for computing the prime decomposition of a directed graph with regard to the cartesian product*

Christophe Crespelle† Eric Thierry‡ Thomas Lambert§

Abstract

In this paper, we design the first linear-time algorithm for computing the prime decomposition of a digraph G with regard to the cartesian product. A remarkable feature of our solution is that it computes the decomposition of G from the decomposition of its underlying undirected graph, for which there exists a linear-time algorithm. First, this allows our algorithm to remain conceptually very simple and in addition, it provides new insight into the connexions between the directed and undirected versions of cartesian product of graphs.

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 [10]). 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 $\square$, 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.

Cartesian product has been used from the early times of graph theory, but the first intensive studies were provided by Sabidussi [9] and Vizing [11].

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

†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

§ENS de Lyon, Université de Lyon, thomas.lambert@ens-lyon.fr

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. The unicity of the decomposition into prime factors was later extended to weakly connected digraphs by Feigenbaum [4] and Walker [12] (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 graphs, after the first polynomial algorithm by Feigenbaum et al [5] in $O(n^{4.5})$ time, where n is the number of vertices, the complexity was improved by Winkler [13], Feder [3] and Aurenhammer et al [1], until Imrich and Peterin [7] 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 [4] 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 [12] which also starts by factorising the graph once arcs are made undirected. Its complexity is not analysed precisely in [12], 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 [8], they obtain an $O(n^7)$ algorithm for solving this problem.

Our results. We present the first linear algorithm to compute the prime decomposition of digraphs with regard to the cartesian product, therefore improving the complexities of [4, 12, 8]. Unlike [12, 8], we solve the problem not only for posets but for arbitrary digraphs G. An interesting feature of our solution is that we use the algorithm of [7] as a black-box, once, at the beginning of our algorithm. In other words, we first compute the decomposition of the underlying undirected graph $\tilde{G}$ of G and afterwards only, we deal with the orientation of arcs, in linear time. This allows our algorithm to remain conceptually very simple. Compared to [4, 12], which proceed in the same way, the improvement of the complexity relies on 1) the incremental structure of our algorithm that fully takes advantage of the product

structure computed for $\tilde{G}$ by parsing it layer by layer, and 2) a careful implementation and choice of the data-structure.

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

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 [9, 11] and then extended to digraphs [4, 12]. 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.

Definition 1 (Cartesian product of digraphs) *The cartesian product* $G = \prod_{1 \le i \le p} G_i$ *of* p *directed graphs* $(G_i)_{1 \le i \le p}$ *is the directed graph* $G = (V(G), A(G))$ *whose vertex set is* $V(G) = \prod_{1 \le i \le p} V(G_i)$ *and such that for all* $x, y \in V(G)$, *with* $x = (x_1, \dots, x_p)$ *and* $y = (y_1, \dots, y_p)$, *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* $(x_1, \dots, x_p)$ *associated with each vertex* x *is called the* cartesian labelling *associated with this decomposition.*

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.*

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 [4, 12] and graphs [9, 11]) *For any weakly connected directed graph (resp. connected graph)* G, *there exists a unique* $p \ge 1$ *and a unique tuple* $(G_1, \dots, G_p)$ *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* $G = \prod_{i \in [\![1,p]\!]} G_i$. $(G_1, \dots, G_p)$ *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 [6]) *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 [7], and of our approach as well.

Definition 4 (Product colouring of arcs (resp. edges) [7]) *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. The next theorem restates Theorem 2 in terms of product colourings. It appears as Lemma 2.3 in [7] for graphs and in [4] for digraphs (stated in terms of partitions of the arcs).

Theorem 3 (Finest product colouring [7]) *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 [7].

Theorem 4 (Square property of product colourings (undirected version) [7]) *Let C be a colouring of the edges of some connected graph G. C is a product colouring iff the three following properties hold:*

1. *all triplets of vertices inducing a triangle in G are monocoloured, and*

2. *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. $\{u,v\}$).*

3. *for any colour $i \in [\![1,|C|]\!]$, if there exists a path of G from x to y, where $x \neq y$, made only of edges coloured i, then there does not exist any path from x to y having no edge coloured i.*

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

This characterisation can be extended to digraphs, up to adding a fourth (minor) condition and carefully checking the arc orientations in Condition 2 of Theorem 4. For digraphs, a pair of 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 5 *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) [7, 4]) *Let C be a colouring of the arcs of some weakly connected digraph G. C is a product colouring iff the four following properties hold:*

1. *all pairs of vertices are monocoloured, and*

2. *all triplets of vertices inducing a triangle in $\tilde{G}$ are monocoloured, and*

3. *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)).*

4. *for any colour $i \in [\![1,|C|]\!]$, if there exists a path of $\tilde{G}$ from x to y, where $x \neq y$, made only of edges coloured i, then there does not exist any path from x to y having no edge coloured i.*

Though not stated in a single theorem in literature, this theorem combines Theorem 4 and the forbidden oriented patterns identified by Feigenbaum in products of digraphs [4]. We take them into account by forcing the types of adjacency around the squares of the digraph in Condition 3 of Theorem 5. The preservation of types on opposite edges of squares is due to

the fact that both edges originate from the same pair of vertices in a factor of G, thus their types exactly reflect the type of this pair. Like for Theorem 4, we have rephrased the characterisation for our needs. We will make an intensive use of Theorem 5 to prove the correctness of our approach.

2 Our approach

Like [7], our approach consists in computing the finest product colouring $\mathcal{C}_G$ of the arcs of G, which is precisely the one corresponding to the prime decomposition of G. We proceed in two steps: i) first, we compute the finest product colouring $\mathcal{C}_{\tilde{G}}$ of the undirected underlying graph $\tilde{G}$ of G and we colour the arcs of G accordingly ii) 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.

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 $\mathcal{C}_G$.*

Sketch of proof. In a directed product colouring, two symmetric arcs are always assigned the same colour. Then, each directed product colouring of G induces a colouring of the edges of $\tilde{G}$. It turns out that this colouring of $\tilde{G}$ is also a product colouring, since the conditions to be an undirected product colouring are weaker than the conditions to be a directed product colouring. Now consider the undirected product colouring induced by $\mathcal{C}_G$, since it is a product colouring, from Theorem 3, it is coarser than $\mathcal{C}_{\tilde{G}}$. It follows that $\mathcal{C}_G$ 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 $\mathcal{C}_{dir}$ in order to obtain the finest product colouring $\mathcal{C}_G$ we aim at computing. We will show that $\mathcal{C}_G$ can be obtained by merging all the pairs of colours that are *conflicting* in $\mathcal{C}_{dir}$.

Definition 6 *Let G be a digraph and let $\mathcal{C}$ be a colouring of its arcs such that all pairs of vertices of G are mono-coloured. 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 3 of Theorem 5.*

The list of conflicting pairs of colours in $\mathcal{C}_{dir}$ defines a graph G_{conf}, which we call the *colour-conflict graph*, whose vertex set is the colours of

$\mathcal{C}_{dir}$. Lemma 2 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 G_{conf}.

Lemma 2 *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} whose vertices are the colours 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.*

Sketch of proof. From Lemma 1, we know that $\mathcal{C}_G$ can be obtained from $\mathcal{C}_{dir}$ by only merging some colours. If there is a conflict between colours c_1 and c_2 in $\mathcal{C}_{dir}$ on some pair $\{\{u,v\},\{v,w\}\}$, then, clearly, the only possibility so that the conflict disappear in $\mathcal{C}_G$ is to merge colours c_1 and c_2. Thus, all pairs of conflicting colours have to be merged, which results in the merge of all the colours in a connected component of G_{conf}. Thus, $\mathcal{C}_{conf}$ is a colouring of the arcs of G finer than $\mathcal{C}_G$. On the other hand, when all these merges have been performed, there is no remaining conflicts between colours, that is Condition 3 of Theorem 5 is satisfied. Consequently, since the other conditions of Theorem 5 are satisfied in $\mathcal{C}_{dir}$ and since these conditions are preserved by merging colours, it follows that $\mathcal{C}_{conf}$ satisfies all the conditions of Theorem 5 and is therefore a product colouring of G. As $\mathcal{C}_{conf}$ is finer than $\mathcal{C}_G$, we have $\mathcal{C}_{conf} = \mathcal{C}_G$.

A key property which we will use in the description of our algorithm, and which allows it to remain conceptually simple and to run in linear time, is 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 what is stated by Lemma 3 below.

Lemma 3 *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 2 of Theorem 4, Condition 1

of Lemma 3 is satisfied. But since bicoloured pair $\{\{u,v\},\{v,w\}\}$ is conflicting, it does not satisfy Condition 3 of Theorem 5. Thus, necessarily, we have $type(u,v') \neq type(v,w)$ or $type(v',w) \neq type(u,v)$.

3 Algorithm

Our algorithm takes as input the adjacency lists of the digraph G and outputs the finest product colouring of the arcs of G together with the corresponding cartesian labelling of the vertices of G. It operates in three steps:

1. *Undirected prime decomposition*: apply Imrich and Peterin's algorithm [7] on $\tilde{G}$ and obtain the induced colouring $\mathcal{C}_{dir}$ of the arcs of G.

2. *Conflicting pairs of colours*: list all pairs of colours that are conflicting in $\mathcal{C}_{dir}$ and build the colour-conflict graph G_{conf}.

3. *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 all of the three steps above in linear time with regard to the size of G.

Cartesian representation

Given a product colouring $\mathcal{C}$ 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 list $N_i(x)$ corresponding to a neighbour y of x again contains two fields: the first one is $type(x,y)$ 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 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 simply the label $(y_1, \ldots, y_p)$ of y is that reading one pointer takes constant time while reading a sequence of integer takes a time proportional to the length of the sequence. This feature is necessary in order to achieve linear time.

Undirected prime decomposition

In this step, from the adjacency lists of G, we compute the cartesian representation of G with regard to the product colouring $\mathcal{C}_{\tilde{G}}$ of $\tilde{G}$ given by the algorithm from [7]. First of all, in order to apply the algorithm from [7], we need to compute the undirected adjacency lists of $\tilde{G}$, from the directed lists of G. This can be done in linear time, and we can determine in the same time the types of all adjacent couples of vertices, which we write into the cells of the adjacency lists.

The adjacency lists of $\tilde{G}$ are then given as input to [7]'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 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, one can build very easily the cartesian representation of G described above. The only difficulty is to sort all the coloured adjacency lists according to the cartesian labels of the vertices they contain. To that purpose, 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 in linear time: 1) initialise a new copy of the adjacency lists 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 here) 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, we have to take care in addition of the colours of the arcs and of the type of the couples of vertices, which can be done without penalising the complexity. Then, the first step of our algorithm takes linear time.

Conflicting pairs of colours

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 properly coloured square of

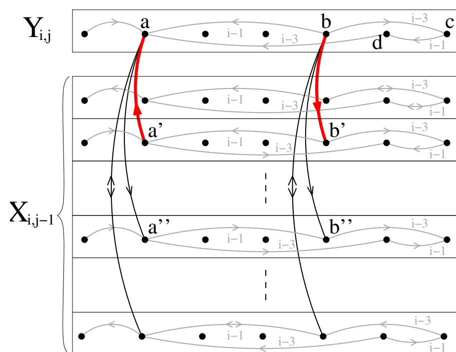

Figure 1: The incremental scheme of our algorithm. The arcs of colours less than i are depicted in grey, and the arcs of colour i in black (except aa' and bb' that are red and bold). Square $abcd$ is a layer-type conflicting square, because of the orientations of bc and ad. It will be detected during the recursive call of our algorithm 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 he second pass after reversing the order on colours.

$\mathcal{C}_{dir}$. And it turns out that the number of such squares may be quadratic, 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. 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.

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 3, the only possibility for conflicts to appear in $\mathcal{C}_{dir}$ is because of the orientation of arcs); then we reverse the order on colours (which can be done easily in linear time on our data-structure) and run the same algorithm. Here, we only describe the first pass.

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 $N_<(x_{i,j}) = \{x_{i,k} \mid k < j$ and $x_{i,j}$ and $x_{i,k}$ are adjacent in $G_i\}$, and $d_<(x_{i,j}) = |N_<(x_{i,j})|$. We also denote $Y_{i,j} = \{x = (x_1, \ldots, x_p) \mid x_i = j$ and $\forall k > i, x_k = 1\}$, and $X_{i,j} = \bigcup_{k \leq j} Y_{i,k}$. While $n_{<i}$

and $m_{<i}$ respectively stand for the number of vertices and the number of adjacent pair of vertices in $G[Y_{i,j}]$ (these numbers do not depend on 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 one by one in increasing (i, j) for the cartesian order. Before adding $x_{i,j}$, it has already computed 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 involve some vertex of $Y_{i,j}$. There are two types of such squares (see Figure 1):

1. *layer type*: those involving only vertices of $Y_{i,j}$.
2. *cross type*: those involving two vertices $a, b \in Y_{i,j}$ and two vertices $a', b' \in X_{i,j-1}$, which are necessarily, since a, b, b', a' is a square, such that $(\forall k \neq i, a'_k = a_k$ and $b'_k = b_k)$ and $(a'_i = b'_i = j')$, with $j' < j$.

In order to detect the layer-type conflicts, we simply recursively apply the algorithm on $G[Y_{i,j}]$. Since the cartesian representation of $G[Y_{i,j}]$ can be extracted from the one of G in linear time with regard to the size of $G[Y_{i,j}]$, then, in order to show that our algorithm performs in linear time, it is sufficient to show that the cross-type conflicts can be computed in time proportional to the number of arcs incident to vertices of $Y_{i,j}$, that is $O(n_{<i}d_<(x_{i,j}) + m_{<i})$ time. Note that for the cross-type conflicts, since we are interested only in the conflicts occurring because of the orientation of arcs of the higher colour, we need only to check the orientations of i-coloured arcs incident to vertices of $Y_{i,j}$. Indeed, in a cross-type square a, b, b', a', the arcs between a and a' and those between b and b' are coloured i, while the arcs linking a and b and linking a' and b' have colour at most $i - 1$. Note that, from the undirected product structure of $\mathcal{C}_{dir}$, for any vertex of $Y_{i,j}$, the number of its neighbours in $X_{i,j-1}$ is $d_<(x_{i,j})$.

In order to list cross-type conflicts, 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 and we set the corresponding cell of $T(y)$ to $type(y, z)$. This takes $O(n_{<i}d_<(x_{i,j}))$ time. Next, we partition $Y_{i,j}$ into the classes of vertices y having the same vector $T(y)$, and we label each vertex of $Y_{i,j}$ with the identifier of the class to which it belongs, that is an integer between 1 and $n_{<i}$. 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 $O(n_{<i}d_<(x_{i,j}))$.

The key property on which lean our algorithm is 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 $\mathcal{P}$, that is having their extremities in two distinct classes of the partition (see Figure 1 and its caption). Indeed, if a j-coloured arc between a and b crosses the partition, the neighbours a' and b' of respectively a and b that distinguished the type vectors $T(a)$ and $T(b)$ form a conflicting square with a and b. And conversely, if there exist a cross-type square a, b, b', a' involving colour $j < i$, then necessarily a and b are not in the same part of $\mathcal{P}$ and the arc between a and b is coloured j.

As a consequence, in order to compute the list of colours conflicting with colour i on some cross-type square, we simply parse the arcs of $G[Y_{i,j}]$ and check whether their two extremities have been assigned the same class identifier of $\mathcal{P}$. This takes $O(m_{<i})$ time and the total time needed to compute the colours conflicting with colour i on cross-type squares is then $O(n_{<i}d_{<}(x_{i,j}) + m_{<i})$. Thus, the running time of our algorithm for listing pairs of colours conflicting in G is $O(n+m)$. Note that the output we get is a list of length $O(n+m)$ with repetitions and containing at most p^2 distinct pairs of colours. We can thus obtain the list without repetitions by bucket sorting the list according to colours, 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 2). This takes $O(p^2) = O(m)$ time.

Merge

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 $C_l = \{l_1, \ldots, l_{|C_l|}\}$, 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 $n'_l = n_{l_1} n_{l_2} \ldots n_{l_{|C_l|}}$ 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. Task ii) is very easy to achieve by simply parsing the arrays $N(x)$ of the cartesian representation and merge the appropriate lists. For Task iii) we can again use the classical technique that allows to sort adjacency lists in linear time. Therefore, Tasks ii) and iii) need only linear computation time. Let us now focus on Task i).

In order to rearrange the matrix of the cartesian representation according to the new label, we first need to compute some matrices and arrays making the correspondences between ancient and new labels of the vertices and

between ancient and new names of colours. First, we number the vertices of the new G'_l's as follows. For each $l \in [\![1,q]\!]$, we build a $n_{l_1} \times \ldots \times n_{l_{|C_l|}}$ matrix $NewName_l$ where cell $NewName_l(a_1,\ldots,a_{|C_l|})$ contains the rank of $(a_1,\ldots,a_{|C_l|})$ in the cartesian order on $[\![1,l_1]\!] \times \ldots \times [\![1,l_{|C_l|}]\!]$. 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 $(a_1,\ldots,a_{|C_l|}) \in [\![1,l_1]\!] \times \ldots \times [\![1,l_{|C_l|}]\!]$ such that $NewName_l(a_1,\ldots,\ a_{|C_l|}) = k$. The t-th component a_t of $AncName_l(k)$ is denoted $AncName_l(k)(t)$. Finally, for each $i \in [\![1,p]\!]$ we compute the values $Newcolour(i) = l$, which is the number l of the class of colours to which colour i belongs, and $Newrank(i)$ which is the rank of i in the ordered list C_l of this class of colours. All matrices $NewName_l$ and arrays $AncName_l$, for all l, as well as arrays $Newcolour$ and $Newrank$ can be computed in $O(np) = O(m)$ time.

Then, we achieve Task i) by building a $n'_1 \times \ldots \times n'_q$ 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 $x' = (x'_1,\ldots,x'_q) \in [\![1,n'_1]\!] \times \ldots \times [\![1,n'_q]\!]$ we store in cell $M_{new}(x'_1,\ldots,x'_q)$ the new label $(x'_1,\ldots,x'_q)$ of vertex x' and the array $M_{dir}(x_1,\ldots,x_p)$ containing the neighbours of vertex x', where $(x_1,\ldots,x_p)$ is the former label of vertex x' in the cartesian representation M_{dir} (see the *Undirected prime decomposition* step of the algorithm). 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.

As a conclusion, each of the three steps of our algorithm runs in $O(n+m)$ time, which is then the total complexity of our algorithm for computing the prime decomposition of 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, which is a natural data-structure for all algorithms willing to exploit the product structure of G.

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):51–229, 2006.

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

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

[5] 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.

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

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

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

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

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

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

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

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