Recognizing Cartesian products in linear time

Wilfried Imrich[a], Iztok Peterin[b]

[a]*Chair of Applied Mathematics, Montanuniversität Leoben, A-8700 Leoben, Austria*
[b]*Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova ulica 17, 2000 Maribor, Slovenia*

Abstract

We present an algorithm that determines the prime factors of connected graphs with respect to the Cartesian product in linear time and space. This improves a result of Aurenhammer et al. [Cartesian graph factorization at logarithmic cost per edge, Comput. Complexity 2 (1992) 331–349], who compute the prime factors in $O(m \log n)$ time, where m denotes the number of vertices of G and n the number of edges. Our algorithm is conceptually simpler. It gains its efficiency by the introduction of edge-labellings.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Cartesian product graphs; Linear algorithm; Decomposition

1. Introduction

Cartesian products are common in graph theory. Typical examples are hypercubes, Hamming graphs, and grid graphs. Hypercubes are powers of K_2, Hamming graphs products of complete graphs, and grid graphs products of paths. These products and their isometric subgraphs have numerous applications in diverse areas, such as Computer Science, Mathematical Chemistry and Biology.

The Cartesian product also has unique algebraic, structural and metric properties. They were investigated in the 1960s by Sabidussi [7] and Vizing [8]. One of these properties is the representation of graphs as the Cartesian product of prime graphs, where a graph is called prime if it cannot be presented as the product of two nontrivial graphs, that is, as the product of two graphs with at least two vertices. Independently Sabidussi and Vizing showed that every connected finite graph has a prime factor decomposition with respect to the Cartesian product that is unique up to the order and isomorphisms of the factors. (For disconnected graphs the factorization is not unique.)

Sabidussi also studied Cartesian products of finite or infinite graphs with infinitely many factors and Vizing the domination number. A conjecture of Vizing [9] from that time about the domination number of Cartesian products is still open.[1]

With the advent of complexity theory in the 1970s the question arose whether one could find the prime factorization of connected graphs in polynomial time. The first positive answer was given in 1985 by Feigenbaum et al. [4], who presented an algorithm of complexity $\mathcal{O}(n^{4.5})$, where n denotes the number of vertices of the investigated graph. Their work extends a method of Sabidussi [7]. Winkler [10] independently found an entirely different algorithm of complexity

---

[1] Vizing conjectures that the domination number of the Cartesian product of two graphs is bounded from below by the product of the domination numbers of the factors.

$\mathcal{O}(n^4)$ and Feder [3] continued with an algorithm that requires $\mathcal{O}(mn)$ time and $\mathcal{O}(m)$ space. This was further improved to $\mathcal{O}(m \log n)$ time and $\mathcal{O}(m)$ space by Aurenhammer et al. [2].

The algorithm presented here is linear in time and space and conceptually simpler. For a given graph G it computes the partition of the edge set of G that is associated with the prime factor decomposition of G. This edge partition is an intrinsic, metric property of G. We only have to find it, we do not impose any additional structure on G by decomposing it into prime factors. No coordinatization of the vertices is needed. In order to prove its correctness, we rely on result about the structure of Cartesian products as presented in [6].

The idea behind our algorithm is simple: given a connected graph G, select a vertex v_0 of minimum degree, list the edges in BFS order with respect to v_0. Assume that every edge incident with v_0 is in a different factor. Scan the edges in BFS order and use this information to determine to which factor the other edges belong. This will be called the *coloring and labelling algorithm.* If it fails for an edge, then there are too many factors. *Merge* them as needed. Then check whether the conditions for Cartesian products are satisfied. This is the *consistency check.* If it fails, then there are too many factors. *Merge* as required. The algorithm ends when the last edge has been successfully processed.[2]

The algorithm is linear,[3] that is, linear in the size of the input. For a connected graph this is equivalent to saying it is linear in the number m of edges if the graph is given by its adjacency list. Since every edge has to be processed, every coloring and labelling operation as well as every consistency check must be effected in constant time. For the merge operations we observe that there cannot be more factors than the minimum degree d_0, hence at most d_0 merge operations are necessary. For each one of them we can use $\mathcal{O}(n)$ time, where n is the number of vertices, because $nd_0 \leqslant 2m$.

This is only possible with a carefully chosen data structure, which also has to satisfy the additional restriction of $\mathcal{O}(m)$ space.

The paper begins with three fundamental lemmas about the Cartesian product. Each of these lemmas is the basis of one of the main parts of the algorithm. The *Square Lemma* is essential for the *coloring and labelling algorithms*, the *Isomorphism Lemma* for the *consistency check* and the *Refinement Lemma* for the correctness of the *merge operations*.

2. Preliminaries

The *Cartesian product* $G \Box H$ of the graphs $G = (V(G), E(G))$ and $H = (V(H), E(H))$ is a graph with vertex set $V(G) \times V(H)$, where the vertices (a, x) and (b, y) are adjacent if $ab \in E(G)$ and $x = y$, or if $a = b$ and $xy \in E(H)$.

The Cartesian product is associative, commutative, and has the one vertex graph K_1 as a unit.

By the associativity we can write $G_1 \Box \cdots \Box G_k$ for a product G of graphs $G_1, \ldots, G_k$ and can label the vertices of G by the set of all k-tuples $(v_1, v_2, \ldots, v_k)$, where $v_i \in G_i$ for $1 \leqslant i \leqslant k$. If v is labelled by $(v_1, v_2, \ldots, v_k)$ we set $p_i v = v_i$ and call v_i the ith *coordinate* of v. The mapping p_i projects G onto G_i. If we restrict p_i to the subgraph induced by all vertices that differ from a given vertex w only in the ith coordinate, it clearly becomes an isomorphism. This subgraph is known as the G_i-*layer* through w and denoted by G_i^w.

As an example, consider the graph G of Fig. 1. Considered as $C_5 \Box C_4$ the layers are five- and four-cycles, considered as $C_5 \Box K_2 \Box K_2$ the layers are five-cycles and sets of parallel edges.

For the distance $d_G(u, v)$ between two vertices $u, v \in V(G)$ we have

$$d_G(u, v) = \sum_{i=1}^{k} d_{G_i}(p_i u, p_i v). \tag{1}$$

This immediately implies that the layers of a product are convex subgraphs, that is, every shortest path between two vertices of one and the same G_i^w is already contained in G_i^w.

In the sequel it will be convenient to select a root $v_0 \in G$ and to identify every G_i with $G_i^{v_0}$. The layers through v_0 are then called *unit layers*. With this terminology every vertex $v \in G_i$ is equal to $p_i v$. Moreover, v_0 is contained in all factors G_i, but otherwise the sets $V(G_i) \backslash \{v_0\}$ are mutually disjoint. The reader might like to visualize the unit-layers of the graph G in Fig. 1 both as a product $C_5 \Box C_4$ and $C_5 \Box K_2 \Box K_2$.

---

[2] See also the short summary of the algorithm at the end of the paper.

[3] In Section 3 we also describe a technically much simpler algorithm of complexity $\mathcal{O}(m^2)$.
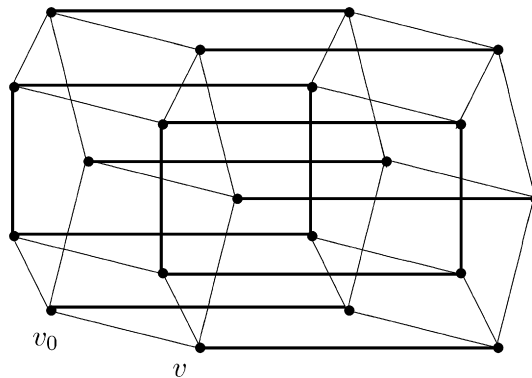
Fig. 1. $G = C_5 \square K_2 \square K_2 = C_5 \square C_4$.

The *preimage* $p_i^{-1}(v)$ of any vertex $v \in G_i$ is a set of vertices in G that induces a subgraph isomorphic to

$$G_i^* = \prod_{k \neq i} G_k.$$

Any such graph is a G_i^*-layer of $G_i \square G_i^*$.

Furthermore the subgraph induced by $p_i^{-1}(u) \cup p_i^{-1}(v)$, where $uv \in E(G_i)$, is isomorphic to $K_2 \square G_i^*$. We call it the *tower* with *base* uv. The K_2-layers of this product are a matching between $p_i^{-1}(u)$ and $p_i^{-1}(v)$. These matching edges induce an isomorphism between the subgraphs of G induced by $p_i^{-1}(u)$ and $p_i^{-1}(v)$. We call the set of these edges the *preimage* of $uv \in E(G_i)$. (The tower with base of $v_0 v$ in Fig. 1 is easily visualized as $K_2 \square C_5$ for the representation of G as $C_5 \square C_4$. For the representation of G as $C_5 \square K_2 \square K_2$ the tower is $K_2 \square (C_5 \square K_2)$, that is, the graph G itself.)

We now define an edge-coloring that reflects the layer structure of G. We first observe that the coordinate vectors[4] of two adjacent vertices u, v differ in exactly one place. Let this place be i. Then $uv \in G_i^u$ and we say that uv has color i, in symbols $c(uv) = i$. We call this a *proper product coloring* of G. (The graph G of Fig. 1 admits five proper product colorings. One with three colors for the representations $C_5 \square A \square B$, where $A \cong B \cong K_2$, three with two colors for the decompositions $C_5 \square (A \square B)$, $(C_5 \square A) \square B$, $(C_5 \square B) \square A$, and one with one and the same color for all edges.)

Every edge is contained in exactly one layer; the edge sets of the layers of G partition the edge-set of G. The edges of color i in the product $G = G_1 \square \cdots \square G_k$ form a spanning subgraph of G and every connected component of this subgraph is a G_i-layer.

The projections p_i and the isomorphism properties of the layers of G allow the characterization of Cartesian products, as we shall illustrate below.

It is easy to see that the restriction of p_i to G_i^w is an isomorphism from G_i^w onto G_i. If there is an edge between two G_i-layers G_i^u and G_i^v we say they are *adjacent*. Clearly the edges between any two adjacent G_i-layers G_i^u and G_i^v induce an isomorphism (and a matching) between them. It is $(p_i^v)^{-1} p_i^u$, where p_j^x denotes the restriction of p_i to G_i^x.

Suppose $e_1 e_2 \ldots e_\ell$ is a path P that connects the G_i-layer G_i^u with G_i^w, and that no e_i has color i. Then concatenation of the isomorphisms induced by the e_j yields an isomorphism of G_i^u onto G_i^w. (Note that $p_i(P)$ is a single vertex.) This isomorphism is uniquely determined and is, analogously to the above, just $(p_i^w)^{-1} p_i^u$. This is a consequence of the fact that every maximal connected subgraph of G that contains no edges of color i, where i is arbitrarily chosen, meets every G_i layer in exactly one vertex. Now the lemma.

Lemma 2.1 (*Isomorphism Lemma*). *Let $G = (V, E)$ be a connected graph and $E_1, E_2, \ldots, E_k$ a partition of E. Suppose that every connected component of $(V, \cup_{j \neq i} E_j)$ meets every connected component of (V, E_i) in exactly*

---

[4] Recall that the coordinate vectors are just labels of length k that satisfy several properties, see above.
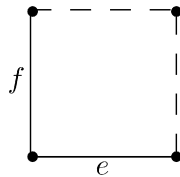
Fig. 2. The square property.

one vertex and that the edge set between any two components of (V, E_i) induces an isomorphism between them if it is nonempty. Then

$$G = \prod G_i,$$

where the G_i are arbitrary components of the graphs (V, E_i).

A graph satisfies the *isomorphism property* if it satisfies the assumptions of the Isomorphism Lemma.

Let two components C and D of (V, E_i) be adjacent. To verify that the edges between adjacent components C and D of (V, E_i) induce an isomorphism between them it clearly suffices to check that every edge of C and D is contained in exactly one square consisting of one edge in C, one in D and two edges between C and D. This gives rise to the Square Lemma.

Lemma 2.2 (*Square Lemma*). *Let G be a properly colored Cartesian product. If e and f are incident edges of different colors, then there exists exactly one square without diagonals that contains e and f.*

It is clear what we mean by the *square property*, see Fig. 2.

Suppose the edges of a graph G are colored in such a way that the square property is satisfied. Then we can infer rather strong assertions about the coloring of certain induced subgraphs. For example, *all edges in a triangle have the same color* and *all edges in a square with at least one diagonal have the same color*.

Also, *all edges in a $K_{2,3}$ are of the same color*, because any two edges incident in a vertex of degree two are contained in two squares without diagonals, and not just one.

Furthermore *any two opposite edges of a square have the same color* and, most importantly, *any two incident edges uv, vw have the same color if v is the only common neighbor of u and w*.

We conclude this section with a formulation of the unique prime factorization property of connected graphs as we shall use it in this paper. Before stating it we wish to remark that we do not always distinguish between a partition of a set and the associated edge-coloring or the equivalence relation whose equivalence classes are just the sets in the given partition. The equivalence relations inherited from a product decomposition are also called *product relations*.

Lemma 2.3 (*Refinement Lemma*). *All product relations with respect to the prime factor decompositions of a connected graph G are identical or finer than any other product relation of G.*

For a proof we refer to [6]. Note that the product relation, resp. product coloring, that corresponds to the decomposition $C_5 \square K_2 \square K_2$ of the graph G of Fig. 1 is finer than all the other product relations of G. Thus $C_5 \square K_2 \square K_2$ is the prime factorization of G. Of course it is not difficult to see directly.

Clearly this holds in general—the finest coloring that satisfies the isomorphism property gives rise to the unique finest product relation and therefore, to the prime factorization of connected graphs. It is our aim to find this coloring in linear time and space.

3. A direct algorithm for the finest product relation

The aim of this section is to demonstrate that the finest product relation is an intrinsic relation of connected graphs that depends only on the metric. It also shows that there is a straightforward $\mathcal{O}(m^2)$ algorithm to compute the finest

product relation and thus the prime factorization of a connected graph. The results shed some light on the underlying ideas of the paper but are not needed in the sequel, so this section can be skipped.

We begin with the definition of two relations Θ and τ on the edge set $E(G)$ of a connected graph G. Two edges $e = xy$ and $f = uv$ are in the relation Θ if

$$d(x, u) + d(y, v) \neq d(x, v) + d(y, u), \tag{2}$$

and they are in the relation τ if $y = u$ is the only common neighbor of x and v. Let σ denote the transitive closure of the union of Θ with τ,

$$\sigma = (\Theta \cup \tau)^*.$$

Theorem 3.1 (*Feder [3]*). *Let G be a finite connected graph. Then $(\Theta \cup \tau)^*$ is the finest product relation of G.*

For a proof see [3] or [6]. To see how the theorem works, consider the graph G of Fig. 1. Any two adjacent edges in one of the pentagons are in the relation τ, and thus any two nonadjacent edges of a pentagon in the relation τ^*, that is, in the transitive closure of τ. Moreover, any two parallel edges (in the figure) are in the relation Θ. Thus, any two edges in the union of the pentagons are in the relation σ. Any two horizontal edges are in the relation Θ as well as any two of the remaining edges. This gives a partition of $E(G)$. It is readily checked that no two edges in different sets of this partition are in the relation Θ or τ. We have thus determined the relation σ_G. Clearly it gives rise to the factorization $C_5 \square K_2 \square K_2$.

Lemma 3.2. *The prime factorization of a connected graph on m edges can be determined within time and space complexity $\mathcal{O}(m^2)$.*

Proof. It suffices to show that σ can be determined within time and space complexity $\mathcal{O}(m^2)$. To see this, we begin with Θ and observe that there are m^2 pairs of edges in G. For every such pair we have to find four distances, and check whether Eq. (2) is satisfied. This can be done in constant time with the aid of the distance matrix, which can be determined in $\mathcal{O}(mn)$ time.

To find τ it suffices to consider all vertices u and check for any other vertex v whether it is adjacent to u or any of its neighbors. Thus, there are $n(d(u) + 1)$ adjacences to be checked for every vertex u. Since

$$\sum_u n(d(u) + 1) = n \sum_u (d(u) + 1) = \mathcal{O}(nm),$$

the time complexity for determining τ is $\mathcal{O}(nm)$.

Thus, the at most m^2 pairs of edges in that are in the relations Θ or τ can be found within time complexity $\mathcal{O}(m^2)$. Hence the transitive closure of $\Theta \cup \tau$ can also be found within $\mathcal{O}(m^2)$ time.[5] Clearly this straightforward algorithm also requires $\mathcal{O}(m^2)$ space. $\square$

Theorem 3.1 is a very special result, because it tells us two things: it makes a deep assertion about the structure of Cartesian products and simultaneously yields a straightforward polynomial algorithm to find the prime factorization of connected graphs. Interestingly, this theorem also holds for infinite connected graphs, see [5].

It should be noted that Feder [3] improved the complexity of this approach by showing that Θ^* can be determined in $\mathcal{O}(mn)$ time and space. As $(\Theta \cup \tau)^* = (\Theta^* \cup \tau)^*$ it is not necessary to determine Θ. This also implies that Θ^* cannot contain more than $\mathcal{O}(mn)$ pairs of edges. Since this also holds for τ both the space and time complexity of the prime factorization reduce to $\mathcal{O}(mn)$.

In the next section we develop a different approach for finding the product coloring of a connected graph.

---

[5] Consider a graph H with $V(H) = E(G)$, where any two edges e, f are adjacent in H if they are in at least one of the relations Θ or τ. Then the equivalence classes of σ correspond to the connected components of H and can be found within time complexity $\mathcal{O}(|E(H)|) = \mathcal{O}(m^2)$.

4. Factorization with additional information

Clearly every graph is a Cartesian product—albeit a trivial one in most cases. Our task is to find the decomposition. Here we find it with some additional information. We show how to find the product coloring of a graph, provided that the colors of the edges incident with a vertex v_0 are known. The only other data of G that we require is the adjacency list.

The importance of the following proposition is the method of proving it: one begins with the coloring of the edges incident with a root vertex v_0 and extends it to all edges of the graph in BFS order. The complexity of this approach is $\mathbb{O}(mn)$.

Proposition 4.1. *Let* $G = G_1 \square G_2 \square \cdots \square G_k$ *be a connected graph. Suppose the colors of the product coloring of* G *with respect to the given decomposition are known for the edges incident with a root vertex* v_0. *Then the product coloring can be recovered in BFS order with respect to the root* v_0 *in* $\mathbb{O}(mn)$ *time and* $\mathbb{O}(n^2)$ *space.*

Proof. We begin by arranging the vertices of G in BFS order with respect to the root v_0 and denote the *distance levels* with respect to this ordering by L_i, $i = 1, \ldots, r$. In other words, $L_i = \{v \mid d_G(v_0, v) = i\}$. Furthermore we partition the set of edges incident with every vertex u into three sets: the set of *down-edges, cross-edges* and *up-edges*. Note that an up-edge uv of u is a down-edge of v. It is thus convenient to consider our edges as ordered pairs of vertices,[6] such that the statement *xy is a down-edge* means that *xy is a down-edge with respect to x*. For cross edges the statement *the cross edge xy* will mean *the cross edge xy with respect to the vertex x*.

We color the cross edges of L_1 first. This is easy, because every triangle is monochromatic. For every cross-edge uv in L_1 we only have to set $c(uv) = c(uv_0)$.

Then we proceed by induction. We assume that we have colored every edge up to the cross-edges of L_i and continue with the down-edges and then the cross-edges of L_{i+1}. Here we do not have to consider the up-edges of L_i separately, because they are also down-edges of L_{i+1}. To color the down-edges of L_{i+1} we scan the vertices $u \in L_{i+1}$ in BFS order and fix a down-edge uv.

1. Suppose this is the only down-edge of u. Since $v \in L_i$, $i \geqslant 1$, there exists a down-edge vx. Clearly v is the only common neighbor of u and x. Thus $c(uv) = c(vx)$.
2. The other possibility is that there are other down-edges. For every such down-edge uw we look for a vertex x that is adjacent to v and w. If such a vertex exists, then $c(uv) = c(wx)$ and $c(uw) = c(vx)$, no matter whether ux and wx have the same or different colors. If no such x exists, then we consider any down-neighbor x of v. As before, v is the only common neighbor of u and x and so $c(uv) = c(vx)$. Analogously we color uw in this case.

For the cross-edges of L_{i+1} we scan the vertices of L_{i+1} again. For every vertex u that is an endpoint of a cross-edge we select an arbitrary cross-edge uv and a down-edge uw. As before we look for common neighbors x of v and w. If such an x exists, then $c(uv) = c(wx)$, otherwise $c(uv) = c(uw)$.

To find the G_i we observe that it suffices to find the $G_i^{v_0}$. Since they are convex, a vertex v is in $V(G_i^{v_0})$ if and only if all of its down-neighbors have color i. A scan of $V(G)$ readily produces these vertices, and the adjacency lists of the $G_i^{v_0}$ are just the i-chromatic sublists of the adjacency list of G, restricted to $V(G_i^{v_0})$.

In Section 2 we called the $G_i^{v_0}$ *unit-layers*. We will call their vertices *unit-layer vertices* henceforth. Clearly all vertices of L_1 are unit-layer vertices.

All down-neighbors and cross-neighbors of a unit-layer vertex $u \in G_i^{v_0}$ are also in $G_i^{v_0}$. This is a consequence of the convexity of layers in a product. (See also Lemma 7.29 on p. 235 in [6].)

For an estimate of the complexity of our procedure we note that we fix a down-neighbor v of every vertex $u \in L_i$, $i > 0$, and then scan all cross- and down-edges of u, so we have a total of at most $2m$ steps to perform, where m denotes the number of edges of G. For every pair of the already determined edges uv and uw we then search for common neighbors of v and w. We have fewer than $n = |V(G)|$ possible choices for x. If we work with the adjacency matrix of G the checks whether x is adjacent to u or w can be performed in constant time. Thus, we arrive at an overall *complexity of* $\mathbb{O}(mn)$ *time and* $\mathbb{O}(n^2)$ *space* for this unrefined approach. $\square$

---

[6] If we consider every edge $[u, v]$ as a set consisting of the two arcs (u, v) and (v, u) we obtain a partition of the set of arcs of G.

5. Coordinatizing a product

In our algorithm we do not need the coordinatization of the vertices. We wish to show here that one can obtain the coordinates of a Cartesian product in linear time and space once the colors of the edges are known.

Theorem 5.1. *Let $G = G_1 \square G_2 \square \cdots \square G_k$ be a connected graph. Suppose the product coloring of G with respect to the given decomposition is known. Then the product can be coordinatized in $\mathcal{O}(m)$ time and space.*

Proof. Again we begin with a BFS ordering of the vertices. For the coordinatization then need a coordinate vector of length k for every vertex, where k is the number of factors. Clearly k is bounded by the minimum degree d_0 of G. Since $n d_0 \leqslant 2m$ the total space needed for these vectors is $\mathcal{O}(m)$.

By definition the ith coordinate of a vertex u is $p_i(u)$. If we identify every G_i with the unit-layer $G_i^{v_0}$ and the vertices of G with their BFS-numbers, then the ith coordinate of u is the BFS-number of $p_i(u)$. As shown in [6] we can then coordinatize the vertices of G as follows:

Begin the BFS-numbering with 0 and let the coordinate vector of v_0 consist of k zeros. Then scan all other vertices u of G in BFS order.

If u is a unit-layer vertex in $G_i^{v_0}$ set the ith place of u equal to the BFS-number of u and all other places to zero.

If u is not a unit-layer vertex there must be down-edges uv and uw of different colors. Set $u_i = \max(v_i, w_i)$ for $1 \leqslant i \leqslant k$.

Clearly the time complexity for this procedure is $\mathcal{O}(m)$. $\quad\square$

6. Labelling the edges of a product

In this section we refine the coloring of the edges and call it *labelling*. All edges in the preimage of a unit layer edge will receive the same *label*. We show that products can be labelled, and thus also colored, in linear time. This labelling allows to tell the position of every edge with respect to the given product decomposition in constant time. In the next one and a half pages we describe the data structure used for storing the labels. Actually, the label will turn out to be the position of an edge in an array of edges with the same endpoint and color.

Note that we used the coordinate vectors for the characterization of the position of vertices in the product, the total length of these vectors being $\mathcal{O}(m)$. For the position of an edge uv in the product we use the initial vertex u, the color i of uv and the projection $p_i(uv)$, that is, $p_i u \, p_i v$. This edge is the *base* of uv. It is in the ith unit layer $G_i^{v_0}$, has the same color as uv, and, because of Eq. (1), is a down-, resp., cross-, or up-edge, if and only if uv is a down-, resp., cross-, or up-edge. Below we describe how to store information about the base efficiently.

The BFS-ordering of Section 4 partitions the set of edges incident with every vertex into arrays of down-, cross-, and up-edges. Every such array is further partitioned into subarrays of edges of the same color. We use the position j of $p_i(uv)$ in its sublist to be able to locate $p_i(uv)$ fast and call $n(uv) = j$ the *name* of uv. In general $n(uv)$ will be different from $n(vu)$. The pair $\langle c(uv), n(uv) \rangle$ is then the *label* of uv, in symbols $\ell(uv)$. Together with $p_{c(uv)}u$ it describes the position of uv in the product.

The position of the initial edge of every color in such an array of down-, cross-, and up-edges is stored in a vector of length $d(v_0)$, thus the jth element of any such subarray of edges of the same color can be accessed in constant time.

Clearly the labelling is well defined. It depends on the ordering of the down- and cross-edges of every unit-layer vertex (all down- and cross-edges of a unit-layer vertex have the same color) and on the ordering of the up-edges of color i in the lists of up-edges of color i for the vertices of $G_i^{v_0}$. (Up-edges of other colors of the vertices in $G_i^{v_0}$ different from v_0 are not unit-layer edges.) The ordering of the other monochromatic sublists of the lists of down-, cross-, and up-edges does not effect the labelling. We shall reorder the monochromatic sublists of edges that are not unit-layer edges according to their names.

It is convenient to generate an edge-list that tells of every edge uv its origin u, its terminus v, whether it is a down-cross- or up-edge as well as its color and name. This way we can find the position of the edge in its monochromatic subarray in constant time once its number is known.

We also modify the adjacency matrix by inserting the name of the edge uv in position (u, v) in order to obtain the labelling of uv in constant time if the adjacency of u and v is checked.

Our data structure thus consists of original adjacency list of the given graph, the modified edge list, and the modified adjacency matrix. Moreover the vertices are now arranged in BFS order, every vertex has a BFS-number and a BFS-level, an optional coordinate vector of length at most $d(v_0)$, and is associated with three arrays: the arrays of down-, cross-, and up-edges.

Every array is subdivided into subarrays of monochromatic edges, where the positions of the initial edge in the subarray are stored in a vector of length $d(v_0)$. Clearly the time needed to build the data structure without the subdivision of these arrays is linear in m. We have to show that the subdivision can also be effected in linear time.

With the exception of the adjacency matrix, the space needed is linear in m too. It is the topic of the next section to show that we do not need the full adjacency matrix and that we can make due with constructing single lines of the adjacency matrix a constant number of times.

Theorem 6.1. *Let $G = G_1 \square G_2 \square \cdots \square G_k$ be a connected graph. Suppose the colors of the product coloring of G with respect to the given decomposition are known for the edges incident with a vertex v_0 of minimum degree. Then the edges of G can be labelled in $\mathcal{O}(m)$ time.*

Proof. We have to describe a linear labelling algorithm. The originally colored edges can easily be labelled. Furthermore, since every edge in $L_0 \cup L_1$ is a unit-layer edge the arrays of down- and cross-edges of L_1 need not be partitioned. The coloring of the cross-edges of L_1 poses no difficulty, every such edge can be colored in constant time, the names are assigned as the positions in the respective sub arrays of up-, down-, or cross-edges.

Suppose we have already labelled all edges up to level L_i such that we can access edges incident with a given vertex and label in constant time. We scan all vertices of L_{i+1} and treat the down edges first. Let $u \in L_{i+1}$.

1. Vertices of down-degree one. If u has down-degree one, then the coloring procedure of Section 4 for the only down-edge of u can be executed in constant time. Its name is 1.
2. Finding a pivot square for vertices of down-degree larger than one.
   2.1. First run. Let uv be the first down-edge of u and vx a down-edge of v. We now check for common vertices of u and x. This can be done by scanning the down-neighbors of u and checking via the adjacency matrix whether they are also neighbors of x. (*We use the line of x.*) The time complexity for finding such a joint neighbor (or to find out that none exists) is $\mathcal{O}(d(u))$.

   If no such neighbor exists, then all down-edges of u and vx have the same color. It is trivial to label all down-edges of u in this case since u is a unit-layer vertex and no partition of the array of down-edges is necessary.

   Suppose now that there exists a common neighbor w of u and x.

   If vx and wx have different colors, then we set $\ell(uv) = \ell(wx)$ and $\ell(uw) = \ell(vx)$. We can also easily find the positions of the initial edges of every color in the subarrays of the array of down-edges of u. We only have to observe that the length of a subarray of any color $\neq c(uv)$ has the same length in arrays of down-edges of u and v and that the length of the subarray of color $c(uv)$ is the same for the down-edges of u and w. The labelling is effected in constant time, the initial edges of every subarray can be found in $\mathcal{O}(d(v_0))$ time.

   We call $uvxw$ a *pivot square* and use it to label the other down-edges of u, but before that we have to treat the case $c(vx) = c(wx)$.
   2.2. Second run. If vx and wx have the same color, then uv and uw also have that color. If all other down-edges of v are of the same color, then u is a unit-layer vertex, a case that has been treated already. So, let vx' be a down-edge with $c(uv) \neq c(vx')$. There must be a common neighbor w' of u and x', we can find it by scanning the down-neighbors of u and checking the adjacency via the adjacency matrix. (*We use the line of x'.*) Now set $\ell(uv) = \ell(w'x')$ and $\ell(uw') = \ell(vx')$. The time complexity for finding w' and x' is $\mathcal{O}(d(u))$. Again the initial edges of the subarrays of the array of down-edges of u can be found in $\mathcal{O}(d(v_0))$ time. In this case $uvx'w'$ is a pivot square.
3. Labelling all down edges of u with a pivot square. To label the other down-edges of u we use the pivot squares $uvxw$, resp., $uvx'w'$. For simplicity we rename x' and w' into x and w for that purpose. Thus, let us scan all down edges of u and let uy be such a down-edge. Consider the down edge yz with $\ell(yz) = \ell(uv)$. Via the adjacency

matrix we check whether z and v are adjacent. (*We use the line of v.*) If they are, then $\ell(uy) = \ell(vz)$. Otherwise $c(uy) = c(uv)$, but then $c(uw) \neq c(uy)$ and we can label uy by means of uw. (*Using the line of w.*) Clearly the time complexity for every vertex y is constant, thus the total complexity for this step is $\mathcal{O}(d(u))$.

Hence the complexity of labelling the down-edges of the vertices in L_{i+1} is

$$\mathcal{O}\left(\sum_{u \in L_{i+1}} d(u)\right).$$

Similarly one can show that the cross-edges of L_{i+1} and the up-edges of L_i can be labelled within the same time complexity. $\square$

7. Labelling in linear time and space

Up to now we have indiscriminately used the adjacency matrix whenever we wished to check in constant time whether two given vertices were adjacent. Unfortunately it requires n^2 space, where n denotes the number of vertices of the graph. It exceeds our aspired space complexity and, even worse, initializing it would destroy the time complexity. Luckily we need only one line of the adjacency matrix at a time, which saves the space complexity. Concerning initialization we note that it is possible to effectively obtain the adjacency matrix for a graph G in $\mathcal{O}(m)$ time, despite the fact that $\mathcal{O}(n^2)$ storage is required. This can be accomplished by a trick first published by Aho et al. [1].

Applying their method to the lines of the adjacency matrix we first observe that the time needed to generate the line for a vertex x is proportional to $d(x)$. Since $\sum_{x \in V(G)} d(x) = 2m$ we stay within the aspired time limit, unless we repeatedly generated lines of vertices with high degree. Care has to be taken to avoid this, we will show below how this is achieved for the labelling of the down-edges.

Theorem 7.1. *Let $G = G_1 \square G_2 \square \cdots \square G_k$ be a connected graph. Suppose the colors of the product coloring of G with respect to the given decomposition are known for the edges incident with a vertex v_0 of minimum degree. Then G can be labelled in linear time and space.*

Proof. By Theorem 6.1 it suffices to show that the total space requirement of the labelling algorithm can be reduced to $\mathcal{O}(m)$ without increasing the time complexity.

Step 1 in the labelling algorithm for the down-edges of level L_{i+1} for the vertices of down-degree one poses no problem.

Step 2 is the search for a pivot square. Two runs may be necessary. In the first run an arbitrary down neighbor x of an arbitrary down-neighbor v of the vertex u of L_{i+1} that is being considered is needed for neighborhood-checking. Let us denote x by x_u. The down-neighbor $x_{u'}$ of distance two of another vertex u' of L_{i+1} may be identical with x_u.

The idea is to determine all such x_u before the line of x_u in the adjacency matrix is generated and to execute the first run for all u' with $x_u = x_{u'}$. The second run is treated just the same.

In *Step* 3 two such procedures will be necessary, this time for vertices in level L_i.

For the cross-edges and the up-edges similar procedures apply, but now things are slightly easier since the down-edges are already labelled and since the colors of the up-edges are already known. In either case two such procedures may be necessary, yielding a total of eight. Thus, no line of the adjacency matrix will have to be generated more than eight times. $\square$

Since the labelling is a refinement of the coloring, G can also be colored in linear time and space under the assumptions of the theorem. Moreover, by Theorem 5.1, it can also be coordinatized within the same time and space complexity.

Note that the main lemma needed here was the Square Lemma.

8. Consistency check

The preceding operations worked under the assumption that the given graph was a product and that we knew the colors of the edges incident with a given vertex. If we are given a coloring and asked to check whether it is a product

coloring it might be tempting to run the labelling algorithm with the information at one vertex and to check whether the computed coloring coincides with the given one. Of course, if it does not, or if the labelling algorithm breaks down, for example in Section 6, Step 2.1, where it says "*there must be. . .*" then the given coloring is not a product coloring. However, it is quite possible that the labelling algorithm goes through, even if these assumptions are not satisfied, for example in a cube with a missing edge or a missing vertex.

So we need a more effective way of testing the validity of the assumptions. We turn to the Isomorphism Lemma for this purpose.

Isomorphisms of graphs are bijections of the vertex-sets that preserve incidence. Using the BFS-ordering of our graphs we check inductively whether the restrictions of these mappings up to BFS level L_i satisfy the all the assumptions and continue with the next level thereafter.

Proposition 8.1. *Suppose that the isomorphism properties hold up to level L_i. Then one can check in time proportional to the sum of numbers of down- and cross-edges of L_{i+1} and the up-edges of L_i whether these properties also hold up to L_{i+1}.*

Proof. We proceed in two steps.

1. For every vertex $u \in L_{i+1}$ that is not a unit-layer vertex we pick two down-edges uv and uw of different color (we could use the pivot square for this purpose) and check whether the down- and cross-edges of u and v that have a color different from $c(uv)$ are in one–one correspondence. This means that for every down- or cross-edge uz of color $\neq c(uv)$ there exists exactly one edge vz' such that $\ell(uz) = \ell(vz')$ and vice versa. Furthermore, zz' must be an edge with $\ell(uv) = \ell(zz')$.
2. For the up-edges of vertices in L_i we proceed similarly. We scan all vertices u of L_i. If u is not a unit-layer vertex, we choose a pivot square $uvxw$ and test the up-edges of u against those of v and w.
   If u is a unit-layer vertex, say $u \in G_i^{v_0}$, we select an arbitrary down-neighbor v of u and check the up-edges of u and v against each other that have a color different from i.

Since every single check can be performed in constant time because of our edge-labelling the total time needed for every vertex u is proportional to the degree of u. Thus, the time complexity for this procedure remains linear.

This way we check the isomorphisms between the layers G_i^u and G_i^v, resp., G_i^w. The others need not be checked. To see this, consider an arbitrary neighbor a of u. Suppose it is a down-neighbor and that $c(ua) \neq c(uv)$. We wish to check the isomorphism between G_i^u and G_i^a induced by the edges between them, where $i \neq c(uv)$.

We have to show that for any down- or cross-edge ab of color $\neq c(ua)$ there is exactly one edge uc with the same label as ab such that b and c are adjacent and $\ell(ua) = \ell(cb)$. Vice versa, to every down- or cross-edge uc of color $\neq c(ua)$ there is exactly one edge ab with the same label as uc such that b and c are adjacent and $\ell(ua) = \ell(cb)$.

We treat the first case. We have already checked the existence of an edge aa' with the same label as uv. By the induction hypothesis all isomorphisms up to level L_i have been checked. Hence, we can use the Square Lemma. We complete $\{a, a', b\}$ to a square $abb'a'$, then $\{v, a', b'\}$ to a square $va'b'c'$ and finally $\{b, b', c'\}$ to a square $bb'c'c$. The edge vc' has the same label as ab and cc' the same label as uv. By the isomorphism check between u and v the vertices u and c are adjacent and uc has the same label as vc', which is the same as that of ab.

The other cases are similarly verified. $\square$

This procedure is called the *consistency check*. We have thus shown:

Theorem 8.2. *The total time and space complexity for the consistency check is linear.*

9. Factorizing by merging colors

Now we have prepared all the tools we need for the prime factorization of a given connected graph G. By the refinement Lemma it suffices to compute the finest product relation of the given graph G. The coloring it induces is the product coloring of the unique prime factor decomposition. It is the finest edge coloring satisfying the Isomorphism Property.

Theorem 9.1. *The prime factorization of connected graphs can be found in linear time and space.*

Proof. We have to present an algorithm. The idea of the algorithm is to start with a coloring that is not coarser than the finest product coloring and to merge color classes when necessary. Since we can color a product if we know the colors of the edges incident with a given vertex v_0 we start with the relation that assigns different colors to the edges incident with v_0. Since every vertex meets edges of all colors no colors can be missed. It is practical to choose v_0 among the vertices of minimal degree, there cannot be more than $d(v_0)$ color. We call these colors *initial colors* henceforth.

We choose v_0 as the basis of the BFS ordering and run the labelling algorithm and the consistency check. It is possible that both go through. Then the result is the coloring with respect to the prime factor decomposition of G.

Most likely though the labelling algorithm will run into difficulties or the consistency check. For example, if there are cross-edges in L_1 the labelling algorithm breaks down. So, what should we do? The answer is easy, every cross edge in L_1 has to have the same color as its down-edges. Since this is not so, our coloring is too fine, we have too many colors and merge them whenever necessary.

It will be useful not to recolor any edges though, we simply note the initial colors that have been merged. Since $d(v_0)^2 \leqslant d(v_0)n \leqslant 2m$ we need no sophisticated data structure to do this from the point of view of space requirements. Also, since we cannot have less than one color, we have at most $d(v_0)$ merge operations in which two sets (of initial colors) of total size less than $d(v_0)$ are merged. If we order the sets, then this can be done in $d(v_0)$ steps. So the total time needed is linear in m too. The initial color of smallest index in a set of merged colors will be called its *principal color*, it is the representative of the set of merged colors. These sets are the new colors. An array of colors as described in the labelling algorithm will thus consist of subarrays of initial colors and the labels will be given with respect to the initial colors, of course always keeping in mind to which (principal) color they belong.

In general we will have checked the consistency of the coloring up to level L_i and wish to continue. The first step is the labelling algorithm. When it fails we have to merge colors. A short analysis of the reasons why it may fail shows that there may be missing vertices, missing edges, too many vertices or too may edges or an array into which the algorithm tries to put an edge is too short. These cases have been analyzed in [6, p. 237], the answer is always the same: if something goes wrong the vertex just being considered must be classified as a unit-layer vertex to allow the labelling algorithm to continue. This means that all initial colors of the down- and cross-edges of this vertex have to be merged.

It is clear that we do not have to recolor any edges. Even more importantly, we do not have to rerun the consistency check. The reason is that fewer colors mean fewer and larger layers. Hence, fewer consistency checks are necessary and all the ones performed retain their validity.

After the labelling algorithm has been completed for L_{i+1} we run the consistency check. Again, if something goes wrong, the reasons can only be the same as before and we merge colors. The result is, that the consistency checks that did not work just do not have to be performed any more! □

We conclude the paper with a short summary of the main steps of our algorithm.

Algorithm 9.2.

1. *Initialization.*
    1.1. *Choose a vertex v_0 of minimum degree and execute the BFS algorithm with base v_0.*
    1.2. *Label the up-edges of v_0 (with name 1 and distinct colors).*
2. *For every BFS level L_i, $i = 1, 2, \ldots, r - 1$ do*
    2.1. *Label (down-, cross-, and up-) edges of L_i.*
    2.2. *Merge colors if necessary.*
    2.3. *Check for the consistency.*
    2.4. *Merge colors if necessary.*

Acknowledgment

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

[2] F. Aurenhammer, J. Hagauer, W. Imrich, Cartesian graph factorization at logarithmic cost per edge, Comput. Complexity 2 (1992) 331–349.

[3] T. Feder, Product graph representations, J. Graph Theory 16 (1992) 467–488.

[4] J. Feigenbaum, J. Hershberger, A.A. Schäffer, A polynomial time algorithm for finding the prime factors of Cartesian-product graphs, Discrete Appl. Math. 12 (1985) 123–138.

[5] W. Imrich, Embedding graphs into Cartesian products, in: Graph Theory and its Applications: East and West (Jinan, 1986), Annals of the New York Academy of Sciences, vol. 576, New York Academy of Sciences, New York, 1989, pp. 266–274.

[6] W. Imrich, S. Klavžar, Product Graphs: Structure and Recognition, Wiley, New York, 2000.

[7] G. Sabidussi, Graph multiplication, Math. Z. 72 (1960) 446–457.

[8] V.G. Vizing, The Cartesian product of graphs (Russian), Vyčisl. Systemy 9 (1963) 30–43, English translation in Comput. Electron Syst. 2 (1966) 352–365.

[9] V.G. Vizing, Some unsolved problems in graph theory, Russian Math. Surveys 23 (1968) 125–141.

[10] P.M. Winkler, Factoring a graph in polynomial time, European J. Combin. 8 (1987) 209–212.