ARS MATHEMATICA CONTEMPORANEA

Also available at http://amc-journal.eu
ISSN 1855-3966 (printed edn.), ISSN 1855-3974 (electronic edn.)
ARS MATHEMATICA CONTEMPORANEA 9 (2015) 233–252

Fast recognition of partial star products and quasi cartesian products*

Marc Hellmuth †
Center for Bioinformatics, Saarland University,
D - 66041 Saarbrücken, Germany

Wilfried Imrich
Chair of Applied Mathematics, Montanuniversität,
A-8700 Leoben, Austria

Tomas Kupka
Department of Applied Mathematics, VSB-Technical University of Ostrava,
Ostrava, 70833, Czech Republic

Received 12 August 2013, accepted 16 December 2013, published online 8 December 2014

Abstract

This paper is concerned with the fast computation of a relation $\mathfrak{d}$ on the edge set of connected graphs that plays a decisive role in the recognition of approximate Cartesian products, the weak reconstruction of Cartesian products, and the recognition of Cartesian graph bundles with a triangle free basis.

A special case of $\mathfrak{d}$ is the relation δ^*, whose convex closure yields the product relation σ that induces the prime factor decomposition of connected graphs with respect to the Cartesian product. For the construction of $\mathfrak{d}$ so-called Partial Star Products are of particular interest. Several special data structures are used that allow to compute Partial Star Products in constant time. These computations are tuned to the recognition of approximate graph products, but also lead to a linear time algorithm for the computation of δ^* for graphs with maximum bounded degree.

Furthermore, we define *quasi Cartesian products* as graphs with non-trivial δ^*. We provide several examples, and show that quasi Cartesian products can be recognized in linear time for graphs with bounded maximum degree. Finally, we note that quasi products can be recognized in sublinear time with a parallelized algorithm.

*We thank Lydia Ostermeier for her insightful comments on graph bundles, as well as for the suggestion of the term "quasi product". This work was supported in part by ARRS Slovenia and the Deutsche Forschungsgemeinschaft (DFG) Project STA850/11-1 within the EUROCORES Program EuroGIGA (project GReGAS) of the European Science Foundation. This paper is based on part of the dissertation of the third author.

†Corresponding Author

Keywords: Cartesian product, quasi product, graph bundle, approximate product, partial star product, product relation.

Math. Subj. Class.: 05C15, 05C10

1 Introduction

Cartesian products of graphs derive their popularity from their simplicity, and their importance from the fact that many classes of graphs, such as hypercubes, Hamming graphs, median graphs, benzenoid graphs, or Cartesian graph bundles, are either Cartesian products or closely related to them [5]. As even slight disturbances of a product, such as the addition or deletion of an edge, can destroy the product structure completely [2], the question arises whether it is possible to restore the original product structure after such a disturbance. In other words, given a graph, the question is, how close it is to a Cartesian product, and whether one can find this product algorithmically. Unfortunately, in general this problem can only be solved by heuristic algorithms, as discussed in detail in [8]. That paper also presents several heuristic algorithms for the solution of this problem.

One of the main steps towards such algorithms is the computation of an equivalence relation $\mathfrak{d}_{|S_v}(W)^*$ on the edge-set of a graph. The complexity of the computation of $\mathfrak{d}_{|S_v}(W)^*$ in [8] is $O(n\Delta^4)$, where n is the number of vertices, and Δ the maximum degree of G. Here we improve the recognition complexity of $\mathfrak{d}_{|S_v}(W)^*$ to $O(m\Delta)$, where m is the number of edges of G, and thereby improve the complexity of the just mentioned heuristic algorithms.

A special case is the computation of the relation $\delta^* = \mathfrak{d}_{|S_v}(V(G))^*$. This relation defines the so-called quasi Cartesian product, see Section 3. Hence, quasi products can be recognized in $O(m\Delta)$ time. As the algorithm can easily be parallelized, it leads to sublinear recognition of quasi Cartesian products.

When the given graph G is a Cartesian product from which just one vertex was deleted, things are easier. In that case, the product is uniquely defined and can be reconstructed in polynomial time from G, see [1] and [3]. In other words, if G is given, and if one knows that there is a Cartesian product graph H such that $G = H \setminus x$, then H is uniquely defined. Hagauer and Žerovnik showed that the complexity of finding H is $O(mn(\Delta^2 + m))$. The methods of the present paper will lead to a new algorithm of complexity $O(m\Delta^2 + \Delta^4)$ for the solution of this problem. This is part of the dissertation [13] of the third author, and will be the topic of a subsequent publication.

Another class of graphs that is closely related to Cartesian products are Cartesian graph bundles, see Section 3. In [11] it was proved that Cartesian graph bundles over a triangle-free base can be effectively recognized, and in [14] it was shown that this can be done in $O(mn^2)$ time. With the methods of this paper, we suppose that one can improve it to $O(m\Delta)$ time. This too will be published separately.

2 Preliminaries

We consider finite, connected undirected graphs $G = (V, E)$ without loops and multiple edges. The *Cartesian product* $G_1 \Box G_2$ of graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$

E-mail addresses: marc.hellmuth@bioinf.uni-sb.de (Marc Hellmuth), imrich@unileoben.ac.at (Wilfried Imrich), tomas.kupka@teradata.com (Tomas Kupka)

is a graph with vertex set $V_1 \times V_2$, where the vertices (u_1, v_1) and (u_2, v_2) are adjacent if $u_1u_2 \in E_1$ and $v_1 = v_2$, or if $v_1v_2 \in E_2$ and $u_1 = u_2$. The Cartesian product is associative, commutative, and has the one vertex graph K_1 as a unit [5]. By associativity we can write $G_1 \Box G_2 \Box \cdots \Box G_k$ for a product G of graphs $G_1, G_2, \ldots, G_k$ 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 $1 \leq i \leq k$. If v is labeled $(v_1, v_2, \ldots, v_k)$, then we call v_i its ith coordinate. One says two edges have the same *Cartesian color* if their endpoints differ in the same coordinate.

A graph G is *prime* if it is non-trivial, and if the identity $G = G_1 \Box G_2$ implies that G_1 or G_2 is the one-vertex graph K_1. A representation of a graph G as a product $G_1 \Box G_2 \Box \cdots \Box G_k$ of prime graphs is called a *prime factorization* of G. It is well known that every connected graph G has a prime factor decomposition with respect to the Cartesian product, and that this factorization is unique up to isomorphisms and the order of the factors, see Sabidussi [15]. Furthermore, the prime factor decomposition can be computed in linear time, see [10].

Following the notation in [8], an induced cycle on four vertices is called *chordless square*. Let the edges $e = vu$ and $f = vw$ span a chordless square $vuxw$. Then f is the *opposite* edge of the edge xu. The vertex x is called *top vertex* (w.r.t. the square spanned by e and f). A top vertex x is *unique* if $|N(x) \cap N(v)| = 2$, where $N(u)$ denotes the (open) 1-neighborhood of vertex u. In other words, a top vertex x is not unique if there are further squares with top vertex x spanned by the edges e or f together with a third distinct edge g. Note that the existence of a unique top vertex x does not imply that e and f span a unique square, as there might be another square $vuyw$ with a possible unique top vertex y. Thus, e and f span a unique square $vuxw$ only if $|N(u) \cap N(w)| = 2$. The *degree* $\deg(u) := |N(u)|$ of a vertex u is the number of edges that contain u. The maximum degree of a graph is denoted by Δ and a path on n vertices by P_n.

We now recall the Breadth-First Search (BFS) ordering of the vertices $v_0, v_1, \ldots, v_{n-1}$ of a graph: select an arbitrary, but fixed vertex $v_0 \in V(G)$, called the *root*, and create a sorted list of vertices. Begin with v_0; append all neighbors $v_1, \ldots, v_{\deg(v_0)}$ of v_0 to the list; then append all neighbors of v_1 that are not already in the list; and continue recursively with $v_2, v_3, \ldots$ until all vertices of G are processed.

2.1 The Relations δ, σ and the Square Property.

There are two basic relations δ and σ, among other relations that are defined on the edge set of a given graph, that play an important role in the field of Cartesian product recognition. In the sequel we shall also use the notation R^* for the *transitive closure* of a relation R, that is, R^* is the smallest transitive relation containing R.

Definition 2.1. Two edges $e, f \in E(G)$ are in the *relation* δ_G, if one of the following conditions in G is satisfied:

(i) e and f are adjacent and it is not the case that there is a unique square spanned by e and f, and that this square is chordless.

(ii) e and f are opposite edges of a chordless square.

(iii) $e = f$.

Clearly, this relation is reflexive and symmetric but not necessarily transitive. The transitive closure δ_G^* is an equivalence relation.

If adjacent edges e and f are not in relation δ, that is, if Condition (i) of Definition 2.1 is not fulfilled, then they span a unique square, and this unique square spanned by e and f is chordless. We call such a square the *unique chordless square (spanned by e and f)*.

Two edges e and f are in the *product relation* σ_G if they have the same Cartesian colors with respect to the prime factorization of G. The product relation σ_G is a uniquely defined equivalence relation on $E(G)$ that contains all information about the prime factorization[1]. Furthermore, δ_G and δ_G^* are contained in σ_G. If there is no risk of confusion we write δ or σ for δ_G or σ_G, respectively.

We say an equivalence relation ρ defined on the edge set of a graph G has the *square property* if the following three conditions hold:

(a) For any two edges $e = uv$ and $f = uw$ that belong to different equivalence classes of ρ there exists a unique vertex $x \neq u$ of G that is adjacent to v and w.

(b) The square $uvxw$ is chordless.

(c) The opposite edges of any chordless square belong to the same equivalence class of ρ.

From the definition of δ it easily follows that δ is a refinement of any such ρ. It also implies that δ^*, and thus also σ, have the square property. This property is of fundamental importance, both for the Cartesian and the quasi Cartesian product. We note in passing that σ is the convex hull of δ^*, see [12].

2.2 The Partial Star Product

This section is concerned with the *partial star product*, which plays a decisive role in the local approach. As it was introduced in [8], we will only define it here, list some of its most basic properties, and refer to [8] for details.

Let $G = (V, E)$ be a given graph and E_v the set of all edges incident to some vertex $v \in V$. We define the local relation $\mathfrak{d}_v$ as follows:

$$\mathfrak{d}_v = ((E_v \times E) \cup (E \times E_v)) \cap \delta_G \subseteq \delta_{\langle N_2^G[v]\rangle},$$

where $\langle N_2^G[v]\rangle$ denotes the induced closed 2-neigborhood of v in G. In other words, $\mathfrak{d}_v$ is the subset of δ_G that contains all pairs $(e, f) \in \delta_G$, where at least one of the edges e and f is incident to v. Clearly $\mathfrak{d}_v^*$, which is not necessarily a subset of δ, is contained in δ^*, see [8].

Let S_v be a subgraph of G that contains all edges incident to v and all squares spanned by edges $e, e' \in E_v$ where e and e' are not in relation $\mathfrak{d}_v^*$. Then S_v is called *partial star product* (PSP for short). To be more precise:

Definition 2.2 (Partial Star Product (PSP))**.** Let $F_v \subseteq E \setminus E_v$ be the set of edges which are opposite edges of (chordless) squares spanned by $e, e' \in E_v$ that are in different $\mathfrak{d}_v^*$ classes, that is, $(e, e') \notin \mathfrak{d}_v^*$.

Then the *partial star product* is the subgraph $S_v \subseteq G$ with edge set $E' = E_v \cup F_v$ and vertex set $\cup_{e\in E'} e$, which consists of the end vertices of the edges in E'. We call v the *center* of S_v, E_v the set of *primal edges*, F_v the set of *non-primal edges*, and the vertices adjacent to v *primal vertices* of S_v.

[1]For the properties of σ that we will cite or use, we refer the reader to [5] or [9].

As shown in [8], a partial star product S_v is always an isometric subgraph or even isomorphic to a Cartesian product graph H, where the factors of H are so-called stars $K_{1,n}$. These stars can directly be determined by the respective $\mathfrak{d}_v^*$ classes, see [8].

Now we define a *local coloring* of S_v as the restriction of the relation $\mathfrak{d}_v^*$ to S_v:

$$\mathfrak{d}_{|S_v} := \mathfrak{d}^*_{v|S_v} = \{(e,f) \in \mathfrak{d}_v^* \mid e,f \in E(S_v)\}.$$

In other words, $\mathfrak{d}_{|S_v}$ is the subset of $\mathfrak{d}_v^*$ that contains all pairs of edges $(e,f) \in \mathfrak{d}_v^*$ where both e and f are in S_v and edges obtain the same local color whenever they are in the same equivalence class of $\mathfrak{d}_{|S_v}$. As an example consider the PSP S_v in Figure 1(d). The relation $\mathfrak{d}_{|S_v}$ has three equivalence classes (highlighted by thick, dashed and double-lined edges). Note, δ^* just contains one equivalence class. Hence, $\mathfrak{d}_{|S_v} \neq \delta^*_{S_v}$.

For a given subset $W \subseteq V$ we set

$$\mathfrak{d}_{|S_v}(W) = \cup_{v \in W} \mathfrak{d}_{|S_v} .$$

The transitive closure of $\mathfrak{d}_{|S_v}(W)$ is then called the *global coloring* with respect to W. As shown in [8], we have the following theorem.

Theorem 2.3. *Let $G = (V,E)$ be a given graph and $\mathfrak{d}_{|S_v}(V) = \cup_{v \in V} \mathfrak{d}_{|S_v}$. Then*

$$\mathfrak{d}_{|S_v}(V)^* = \delta^*_G.$$

For later reference and for the design of the recognition algorithm we list the following three lemmas about relevant properties of the PSP.

Lemma 2.4 ([8])**.** *Let G=(V,E) be a given graph and S_v be a PSP of an arbitrary vertex $v \in V$. If $e,f \in E_v$ are primal edges that are not in relation $\mathfrak{d}_v^*$, then e and f span a unique chordless square with a unique top vertex in G.*

Conversely, suppose that x is a non-primal vertex of S_v. Then there is a unique chordless square in S_v that contains x, and that is spanned by edges $e,f \in E_v$ with $(e,f) \notin \mathfrak{d}_v^*$.

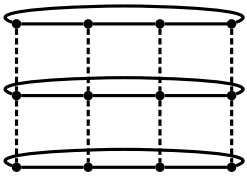
Lemma 2.5 ([8])**.** *Let G=(V,E) be a given graph and $f \in F_v$ be a non-primal edge of a PSP S_v of an arbitrary vertex $v \in V$. Then f is opposite to exactly one primal edge $e \in E_v$ in S_v, and $(e,f) \in \mathfrak{d}_{|S_v}$.*

Lemma 2.6 ([8])**.** *Let G=(V,E) be a given graph and $W \subseteq V$ such that $\langle W \rangle$ is connected. Then each vertex $x \in W$ meets every equivalence class of $\mathfrak{d}_{|S_v}(W)^*$ in $\cup_{v \in W} S_v$.*

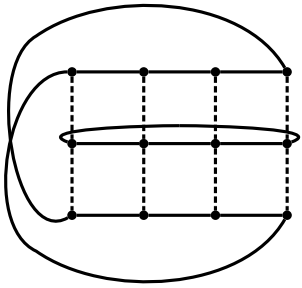
3 Quasi Cartesian Products

Given a Cartesian product $G = A \square B$ of two connected, prime graphs A and B, one can recover the factors A and B as follows: the product relation σ has two equivalence classes, say E_1 and E_2, and the connected components of the graph $(V(G), E_1)$ are all isomorphic copies of the factor A, or of the factor B, see Figure 1(a). This property naturally extends to products of more than two prime factors.

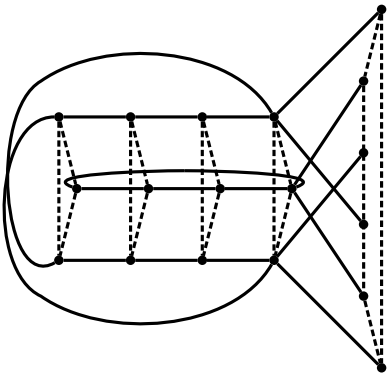
We already observed that δ is finer than any equivalence relation ρ that satisfies the square property. Hence the equivalence classes of ρ are unions of δ^*-classes. This also



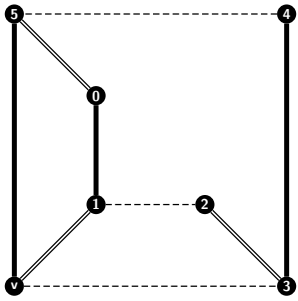
(a) The Cartesian product $G = P_3 \square C_4$.



(b) A quasi Cartesian product, which is also a graph bundle.



(c) A quasi Cartesian product, which is not a graph bundle.



(d) The approximate product and PSP S_v, which is neither a quasi product nor a graph bundle.

Figure 1: Shown are several quasi Cartesian products, graph bundles and approximate products.

holds for σ. It is important to keep in mind that σ can be trivial, that is, it consists of a single equivalence class even when δ^* has more than one equivalence class.

We call all graphs G with a non-trivial equivalence relation ρ that is defined on $E(G)$ and satisfies the square property *quasi (Cartesian) products*. Since $\delta^* \subseteq \rho$ for every such relation ρ, it follows that δ^* must have at least two equivalence classes for any quasi product. By Theorem 2.3 we have $\mathfrak{d}_{|S_v}(V(G))^* = \delta^*$. In other words, quasi products can be defined as graphs where the PSP's of all vertices are non-trivial, that is, none of the PSP's is a star $K_{1,n}$, and in addition, where the union over all $\mathfrak{d}_{|S_v}$ yields a non-trivial δ^*.

Consider the equivalence classes of the relation δ^* of the graph G of Figure 1(b). It has two equivalence classes, and locally looks like a Cartesian product, but is actually reminiscent of a Möbius band. Notice that the graph G in Figure 1(b) is prime with respect to Cartesian multiplication, although δ^* has two equivalence classes: all components of the first class are paths of length 2, and there are two components of the other δ^*-class, which do not have the same size. Locally this graph looks either like $P_3 \square P_3$ or $P_2 \square P_3$.

In fact, the graph in Figure 1(b) is a so-called Cartesian graph bundle [11], where *Cartesian graph bundles* are defined as follows: Let B and F be graphs. A graph G is a (Carte-

sian) graph bundle with fiber F over the base B if there exists a weak homomorphism[2] $p : G \to B$ such that

(i) for any $u \in V(B)$, the subgraph (induced by) $p^{-1}(u)$ is isomorphic to F, and

(ii) for any $e \in E(B)$, the subgraph $p^{-1}(e)$ is isomorphic to $K_2 \square F$.

The graph of Figure 1(c) shows that not all quasi Cartesian products are graph bundles. On the other hand, not every graph bundle has to be a quasi product. The standard example is the complete bipartite graph $K_{3,3}$. It is a graph bundle with base K_3 and fiber K_2, but has only one δ^*-class.

Note, in [8] we considered "approximate products" which were first introduced in [7, 6]. As approximate products are the graphs that have a (small) edit distance to a non-trivial product graph, it is clear that every bundle and quasi product can be considered as an approximate product, while the converse is not true. For example, consider the graph in Figure 1(d). Here, δ^* has only one equivalence class. However, the relation $\mathfrak{d}_{|S_v}$ has, in this case, three equivalence classes (highlighted by thick, dashed and double-lined edges).

Because of the local product-like structure of quasi Cartesian products we are led to the following conjecture:

Conjecture 3.1. *Quasi Cartesian products can be reconstructed in essentially the same time from vertex-deleted subgraphs as Cartesian products.*

4 Recognition Algorithms

4.1 Computing the Local and Global Coloring

For a given graph G, let $W \subseteq V(G)$ be an arbitrary subset of the vertex set of G such that the induced subgraph $\langle W \rangle$ is connected. Our approach for the computation is based on the recognition of all PSP's S_v with $v \in W$, and subsequent merging of their local colorings. The subroutine computing local colorings calls the vertices in BFS-order with respect to an arbitrarily chosen root $v_0 \in W$.

Let us now briefly introduce several additional notions used in the PSP recognition algorithm. At the start of every iteration we assign pairwise different *temporary local colors* to the primal edges of every PSP. These colors are then merged in subroutine processes to compute *local colors* associated with every PSP. Analogously, we use *temporary global colors* that are initially assigned to every edge incident with the root v_0.

For any vertex v of distance two from a PSP center c we store attributes called *first and second primal neighbor*, that is, references to adjacent primal vertices from which v was "visited" (in pseudo-code attributes are accessed by $v.FirstPrimalNeighbor$ and $v.SecondPrimalNeighbor$). When v is found to have at least two primal neighbors we add v to $\mathbb{T}_c$, which is a stack of candidates for non-primal vertices of S_c. Finally, we use *incidence* and *absence lists* to store recognized squares spanned by primal edges. Whenever we recognize that two primal edges span a square we put them into the incidence list. If we find out that a pair of primal edges cannot span a unique chordless square with unique top vertex, then we move it into the absence list. Note that the above structures are local and are always associated with a certain PSP recognition subroutine (Algorithm 4.1). Finally, we will "map" local colors to temporary global colors via temporary vectors which helps us to merge local with global colors.

[2] A weak homomorphism maps edges into edges or single vertices.

Algorithm 4.1 computes a local coloring for a given PSP and merges it with the global coloring $\mathfrak{d}_{|S_v}(W)^*$ where $W \subseteq V(G)$ is the set of treated centers. Algorithm 4.2 summarizes the main control structure of the local approach.

Algorithm 4.1 (*PSP recognition*)

Input: *Connected graph $G = (V, E)$, PSP center $c \in V$, global coloring $\mathfrak{d}_{|S}(W)^*$, where $W \subseteq V$ is the set of treated centers and where the subgraph induced by $W \cup c$ is connected.*

Output: *New temporary global coloring $\mathfrak{d}_{|S}(W \cup c)^*$.*

1. *Initialization.*
2. `FOR` *every neighbor u of c* `DO:`
   (a) `FOR` *every neighbor w of u (except c)* `DO:`
      i. `IF` *w is primal w.r.t. c* `THEN` *add pair of primal edges (cu, cw) to absence list.*
      ii. `ELSEIF` *w was not visited* `THEN` *set $w.FirstPrimalNeighbor = u$.*
      iii. `ELSE` *(w is not primal and was already visited)* `DO:`
         A. `IF` *only one primal neighbor v ($v \neq u$) of w was recognized so far, then* `DO:`
            - *Set $w.SecondPrimalNeighbor = u$.*
            - `IF` *(cu, cv) is not in incidence list, then add w to the stack $\mathbb{T}_c$ and add the pair (cu, cv) to incidence list.*
            - `ELSE` *(cu and cv span more squares) add pair (cu, cv) to absence list.*
         B. `ELSE:`
            - *Add all pairs formed by primal edges cv_1, cv_2, cu to absence list, where v_1, v_2 are first and second primal neighbors of w.*
3. *Assign pairwise different temporary local colors to primal edges.*
4. `FOR` *any pair (cu, cv) of primal edges cu and cv* `DO:`
   (a) `IF` *(cu, cv) is contained in absence list* `THEN` *merge temporary local colors of cu and cv.*
   (b) `IF` *(cu, cv) is not contained in incidence list* `THEN` *merge temporary local colors of cu and cv.*

   (Resulting merged temporary local colors determine local colors of primal edges in S_c. We will reference them in the following steps.).
5. `FOR` *any primal edge cu* `DO:`
   (a) `IF` *cu was already assigned some temporary global color d_1* `THEN`
      i. `IF` *local color b of cu was already mapped to some temporary global color d_2, where $d_2 \neq d_1$,* `THEN` *merge d_1 and d_2.*
      ii. `ELSE` *map local color b to d_1.*
6. `FOR` *any vertex v from the stack $\mathbb{T}_c$* `DO:`
   (a) *Check local colors of primal edges cw_1 and cw_2 (where w_1, w_2 are first and second primal neighbor of v, respectively).*
   (b) `IF` *they differ in local colors* `THEN`
      i. `IF` *there was defined temporary global color d_1 for vw_1* `THEN`

A. `IF` local color b of cw_2 was already mapped to some temporary global color d_2, where $d_2 \neq d_1$ `THEN` merge d_1 and d_2.
B. `ELSE` map local color b to d_1.

ii. `IF` there was already defined temporary global color d_1 for vw_2 `THEN`:

A. `IF` local color b of cw_1 was already mapped to some temporary global color d_2, where $d_2 \neq d_1$ `THEN` merge d_1 and d_2.
B. `ELSE` map local color b to d_1.

7. Take every edge e of the PSP S_c that was not colored by any temporary global color up to now and assign it d, where d is the temporary global color to which the local color of e or the local color of its opposite primal edge e' was mapped.
(If there is a local color b that was not mapped to any temporary global color, then we create a new temporary global color and assign it to all edges of color b.)

Algorithm 4.2 (*Computation of* $\mathfrak{d}_{|S_v}(W)^*$)

Input: A connected graph G, $W \subseteq V(G)$ s.t. the induced subgraph $\langle W \rangle$ is connected, and an arbitrary vertex $v_0 \in W$.
Output: Relation $\mathfrak{d}_{|S_v}(W)^*$.

1. Initialization.
2. Set sequence Q of vertices $v_0, v_1, \ldots, v_n$ that form W in BFS-order with respect to v_0.
3. Set $W' := \emptyset$.
4. Assign pairwise different temporary global colors to edges incident to v_0.
5. `FOR` any vertex v_i from sequence Q `DO`:
   (a) Use Algorithm 4.1 to compute $\mathfrak{d}_{|S_v}(W' \cup v_i)^*$.
   (b) Add v_i to W'.

In order to show that Algorithm 4.1 correctly recognizes the local coloring, we define the (temporary) relations α_c and β_c for a chosen vertex c: Two *primal* edges of S_c are

- in *relation* α_c if they are contained in the incidence list and
- in *relation* β_c if they are contained in the absence list

after Algorithm 4.1 is executed for c. Note, we denote by $\overline{\alpha}_c$ the complement of α_c, which contains all pairs of primal edges of PSP S_c that are not listed in the incidence list.

Lemma 4.1. *Let e and f be two primal edges of the PSP S_c. If e and f span a square with some non-primal vertex w as unique top-vertex, then $(e, f) \in \alpha_c$.*

Proof. Let $e = cu_1$ and $f = cu_2$ be primal edges in S_c that span a square cu_1wu_2 with unique top-vertex w, where w is non-primal. Note, since w is the unique top vertex, the vertices u_1 and u_2 are its only primal neighbors. W.l.o.g. assume that for vertex w no first primal neighbor was assigned and let first u_1 and then u_2 be visited. In Step 2a vertex w is recognized and the first primal neighbor u_1 is determined in Step 2(a)ii. Take the next vertex u_2. Since w is not primal and was already visited, we are in Step 2(a)iii. Since only one primal neighbor of w was recognized so far, we go to Step 2(a)iiiA. If (cu_1, cu_2) is not already contained in the incidence list, it will be added now and thus, $(cu_1, cu_2) \in \alpha_c$. □

Corollary 4.2. *Let e and f be two adjacent distinct primal edges of the PSP S_c. If $(e, f) \in \overline{\alpha}_c$, then e and f do not span a square or span a square with non-unique or primal top vertex. In particular, $\overline{\alpha}_c$ contains all pairs (e, f) that do not span any square.*

Proof. The first statement is just the contrapositive of the statement in Lemma 4.1. For the second statement observe that if $e = cx$ and $f = cy$ are two distinct primal edges of S_c that do not span a square, then the vertices x and y do not have a common non-primal neighbor w. It is now easy to verify that in none of the substeps of Step 2 the pair (e, f) is added to the incidence list, and thus, $(e, f) \in \overline{\alpha}_c$. □

Lemma 4.3. *Let e and f be two primal edges of the PSP S_c that are in relation β_c. Then e and f do not span a unique chordless square with unique top vertex.*

Proof. Let $e = cu_1$ and $f = cu_2$ be primal edges of S_c. Then pair (e, f) is added to the absence list in:

a) Step 2(a)i, when u_1 and u_2 are adjacent. Then no square spanned by e and f can be chordless.

b) Step 2(a)iiiA (`ELSE`-condition), when (e, f) is already listed in the incidence list and another square spanned by e and f is recognized. Thus, e and f do not span a unique square.

c) Step 2(a)iiiB, when e and f span a square with top vertex w that has more than two primal neighbors and at least one of the primal vertices u_1 and u_2 are recognized as first or second primal neighbor of w. Thus e and f span a square with non-unique top vertex.

□

Lemma 4.4. *Relation β_c^* contains all pairs of primal edges (e, f) of S_c that satisfy at least one of the following conditions:*

a) e and f span a square with a chord.

b) e and f span a square with non-unique top vertex.

c) e and f span more than one square.

Proof. Let $e = cu_1$ and $f = cu_2$ be primal edges of the PSP S_c.

a) If e and f span a square with a chord, then u_1 and u_2 are adjacent or the top vertex w of the spanned square is primal and thus, there is a primal edge $g = cw$. In the first case, we can conclude analogously as in the proof of Lemma 4.3 that $(e, f) \in \beta_c$. In the second case, we analogously obtain $(e, g), (f, g) \in \beta_c$ and therefore, $(e, f) \in \beta_c^*$.

b) Let e and f span a square with non-unique top vertex w. If at least one of the primal vertices u_1, u_2 is a first or second neighbor of w then e and f are listed in the absence list, as shown in the proof of Lemma 4.3. If u_1 and u_2 are neither first nor second primal neighbors of w, then both edges e and f will be added to the absence list in Step 2(a)iiiB, together with the primal edge $g = cu_3$, where u_3 is the first recognized primal neighbor of w. In other words, $(e, g), (f, g) \in \beta_c$ and hence, $(e, f) \in \beta_c^*$.

c) Let e and f span two squares with top vertices w and w', respectively and assume w.l.o.g. that first vertex w is visited and then w'. If both vertices u_1 and u_2 are recognized as first and second primal neighbors of w and w', then (cu_1, cu_2) is added to the incidence list when visiting w in Step 2(a)iiiA. However, when we visit w', then we insert (cu_1, cu_2) to the absence list in Step 2(a)iiiA, because this pair is already included in the incidence list. Thus, $(e, f) \in \beta_c$. If at least one of the vertices w, w' does not have u_1 and u_2 as first or second primal neighbor, then e and f must span a square with non-unique top vertex. Item b) implies that $(e, f) \in \beta_c^*$.

□

Lemma 4.5. *Let f be a non-primal edge and e_1, e_2 be two distinct primal edges of S_c. Let $(e_1, f), (e_2, f) \in \mathfrak{d}_c$. Then $(e_1, e_2) \in \beta_c^*$.*

Proof. Since the edge f is non-primal, f is not incident with the center c. Recall, by the definition of $\mathfrak{d}_c$, two distinct edges can be in relation $\mathfrak{d}_c$ only if they have a common vertex or are opposite edges in a square. To prove our lemma we need to investigate the three following cases, which are also illustrated in Figure 2:

a) Suppose both edges e_1 and e_2 are incident with f. Then e_1 and e_2 span a triangle and consequently (e_1, e_2) will be added to the absence list in Step 2(a)i.

b) Let e_1 and e_2 be opposite to f in some squares. There are two possible cases (see Figure 2 b)). In the first case e_1 and e_2 span a square with non-unique top vertex. By Lemma 4.4, $(e_1, e_2) \in \beta_c^*$. In the second case e_1 and e_2 span triangles with other primal edges e_3 and e_4. As in Case a) of this proof, we have $(e_1, e_3) \in \beta_c$, $(e_3, e_4) \in \beta_c$, $(e_4, e_2) \in \beta_c$ and consequently, $(e_1, e_2) \in \beta_c^*$

c) Suppose only e_1 has a common vertex with f and e_2 is opposite to f in a square. Again we need to consider two cases (see Figure 2 c)). Since e_1 and f are adjacent and $(e_1, f) \in \mathfrak{d}_c$, we can conclude that either no square is spanned by e_1 and f, or that the square spanned by e_1 and f is not chordless or not unique. It is easy to see that in the first case the edges e_1 and e_2 are contained in a common triangle and thus will be added to the absence list in Step 2(a)i. In the second case e_1, e_2 span a square which has a chord or has a non-unique top vertex. In both cases Lemma 4.4 implies that e_1 and e_2 are in relation β_c^*.

□

Lemma 4.6. *Let e and f be distinct primal edges of the PSP S_c. Then $(e, f) \in (\overline{\alpha}_c \cup \beta_c)^*$ if and only if $(e, f) \in \mathfrak{d}_c^*$.*

Proof. Assume first that $(e, f) \in \overline{\alpha}_c \cup \beta_c$. By Corollary 4.2, if $(e, f) \in \overline{\alpha}_c$, then e and f do not span a common square, or span a square with non-unique or primal top vertex. In the first case, e and f are in relation δ_G and consequently also in relation $\mathfrak{d}_c$. On the other hand, if e and f span square with non-unique top vertex then, by Lemma 2.4, e and f are in relation $\mathfrak{d}_c^*$ as well. Finally, if e and f span a square with primal top vertex w, then this square has a chord cw and $(e, f) \in \mathfrak{d}_c^*$. If $(e, f) \in \beta_c$, then Lemma 4.3 implies that e and f do not span a unique chordless square with unique top vertex. Again, by Lemma 2.4, we infer that $(e, f) \in \mathfrak{d}_c^*$. Hence, $\overline{\alpha}_c \cup \beta_c \subseteq \mathfrak{d}_c^*$, and consequently, $(\overline{\alpha}_c \cup \beta_c)^* \subseteq \mathfrak{d}_c^*$.

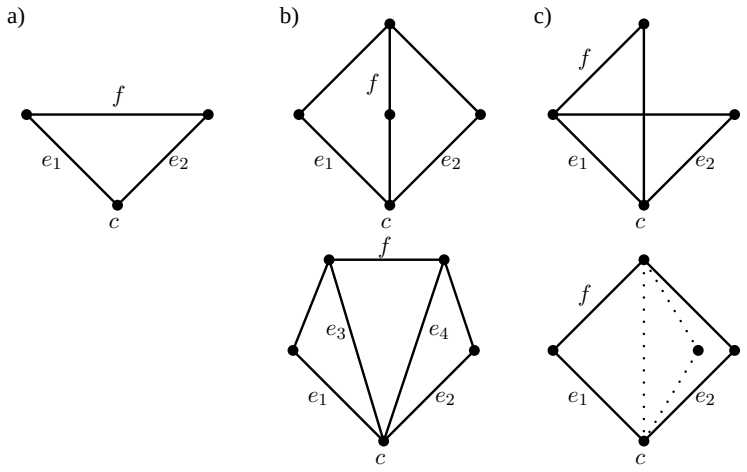


Figure 2: The three possible cases a), b), and c) that are investigated in the proof of Lemma 4.5.

Now, let $(e, f) \in \mathfrak{d}_c^*$. Then there is a sequence $U = (e = e_1, e_2, \ldots, e_k = f)$, $k \geq 2$, with $(e_i, e_{i+1}) \in \mathfrak{d}_c$ for $i \in \{1, 2, \ldots, k-1\}$. By definition of $\mathfrak{d}_c$, two primal edges are in relation $\mathfrak{d}_c$ if and only if they do not span a unique and chordless square. Corollary 4.2 and Lemma 4.4 imply that all these pairs are contained in $(\overline{\alpha}_c \cup \beta_c)^*$. Hence, any two consecutive primal edges e_i and e_{i+1} contained in the sequence U are in relation $(\overline{\alpha}_c \cup \beta_c)^*$. Assume that there is an edge $e_i \in U$ that is not incident to the center c and thus, non-primal. By the definition of $\mathfrak{d}_c$, and since $(e_{i-1}, e_i), (e_i, e_{i+1}) \in \mathfrak{d}_c$, we can conclude that the edges e_{i-1} and e_{i+1} must be primal in S_c. Lemma 4.5 implies that e_{i-1} and e_{i+1} must be in relation β_c^*. Thus, if we remove the edge e_i from U, we still can claim that all consecutive primal edges in $U \setminus \{e_i\}$ are in relation $(\overline{\alpha}_c \cup \beta_c)^*$. By removing all non-primal edges from U we therefore obtain a sequence $U' = e = e_1, e_2', \ldots, e_j' = f$ of primal edges. By analogous arguments as before, all pairs (e_i', e_{i+1}') of U' must be contained in $(\overline{\alpha}_c \cup \beta_c)^*$. By transitivity, e and f are also in $(\overline{\alpha}_c \cup \beta_c)^*$. □

Corollary 4.7. *Let e and f be primal edges of the PSP S_c. Then $(e, f) \in (\overline{\alpha}_c \cup \beta_c)^*$ if and only if e and f have the same local color in S_c.*

Proof. This is an immediate consequence of Lemma 4.6, the local color assignment, and the merging procedure (Step 3 and 4) in Algorithm 4.1. □

Lemma 4.8. *Let $\mathfrak{d}_{|S_v}(W)^*$ be a global coloring associated with a set of treated centers W and assume that the induced subgraph $\langle W \rangle$ is connected. Let c be a vertex that is not contained in W but adjacent to a vertex in W. Then Algorithm 4.1 computes the global coloring $\mathfrak{d}_{|S_v}(W \cup c)^*$ by taking W and c as input.*

Proof. Let $W \subseteq V(G)$ be a set of PSP centers and let $c \in V(G)$ be a given center of PSP S_c where $c \notin W$ and $\langle W \cup c \rangle$ is connected. In Step 2 of Algorithm 4.1 we compute the absence and incidence lists. In Step 3, we assign pairwise different temporary local colors to any primal edge adjacent to c. Two temporary local colors b_1 and b_2 are then merged in Step 4 if and only if there exists some pair of primal edges $(e_1, e_2) \in (\overline{\alpha}_c \cup \beta_c)$ where

e_1 is colored with b_1 and e_2 with b_2. Therefore, merged temporary local colors reflect equivalence classes of $(\overline{\alpha}_c \cup \beta_c)^*$ containing the primal edges incident to c. By Corollary 4.7, $(\overline{\alpha}_c \cup \beta_c)^*$ classes indeed determine the local colors of primal edges in S_c.

Note, if one knows the colors of primal edges incident to c, then it is very easy to determine the set of non-primal edges of S_c, as any two primal edges of different equivalence classes span a unique and chordless square. In Step 6, we investigate each vertex v from stack $\mathbb{T}_c$ and check the local colors of primal edges cw_1 and cw_2, where w_1 and w_2 are the first and second recognized primal neighbors of v, respectively. If cw_1 and cw_2 differ in their local colors, then vw_1 and vw_2 are non-primal edges of S_c, as follows from the PSP construction. Recall that the stack contains all vertices that are at distance two from center c and which are adjacent to at least two primal vertices. In other words, the stack contains all non-primal top vertices of all squares spanned by primal edges. Consequently, we claim that all non-primal edges of the PSP S_c are treated in Step 6. Note that non-primal edges have the same local color as their opposite primal edge, which is unique by Lemma 2.5.

As we already argued, after Step 4 is performed we know, or can at least easily determine all edges of S_c and their local colors. Recall that local colors define the local coloring $\mathfrak{d}_{|S_c}$. Suppose, temporary global colors that correspond to the global coloring $\mathfrak{d}_{|S_v}(W)^*$ are assigned. Our goal is to modify and identify temporary global colors such that they will correspond to the global coloring $\mathfrak{d}_{|S_v}(W \cup c)^*$. Let $B_1, B_2, \ldots, B_k$ be the classes of $\mathfrak{d}_{|S_c}$ (*local classes*) and $D_1, D_2, \ldots, D_l$ be the classes of $\mathfrak{d}_{|S_v}(W)^*$ (*global classes*). When a local class B_i and a global class D_j have a nonempty intersection, then we can infer that all their edges must be contained in a common class of $\mathfrak{d}_{|S_v}(W \cup c)^*$. Note, by means of Lemma 2.6, we can conclude that for each local class B_i there is a global class D_j such that $B_i \cap D_j \neq \emptyset$, see also [8]. In that case we need to guarantee that edges of B_i and D_j will be colored by the same temporary global color. Note, in the beginning of the iteration two edges have the same temporary global color if and only if they lie in a common global class.

In Step 5 and Step 6, we investigate all primal and non-primal edges of S_c. When we treat first edge e that is colored by some local color b_i, that is $e \in B_i$, and has already been assigned some temporary global color d_j, and therefore $e \in D_j$, then we map b_i to d_j. Thus, we keep the information that $e \in B_i \cap D_j$. In Step 7, we then assign temporary global color d_j to any edge of S_c that is colored by the local color b_i. If the local color b_i is already mapped to some temporary global color d_j, and if we find another edge of S_c that is colored by b_i and simultaneously has been assigned some different temporary global color $d_{j'}$, then we merge d_j and $d_{j'}$ in Step 5(a)i. Obviously this is correct, since $B_i \cap D_j \neq \emptyset$ and $B_i \cap D_{j'} \neq \emptyset$, and hence $D_j, D_{j'}$ and B_i must be contained in a common equivalence class of $\mathfrak{d}_{|S_v}(W \cup c)^*$. Recall, for each local class B_i there is a global class D_j such that $B_i \cap D_j \neq \emptyset$. This means that every local color is mapped to some global color, and consequently there is no need to create a new temporary global color in Step 7.

Therefore, whenever local and global classes share an edge, then all their edges will have the same temporary global color at the end of Step 7. On the other hand, when edges of two different global classes are colored by the same temporary global color, then both global classes must be contained in a common class of $\mathfrak{d}_{|S_v}(W \cup c)^*$.

Hence, after the performance of Step 7, the merged temporary global colors determine the equivalence classes of $\mathfrak{d}_{|S_v}(W \cup c)^*$. □

Lemma 4.9. *Let G be a connected graph, $W \subseteq V(G)$ s.t. $\langle W \rangle$ is connected, and v_0*

an arbitrary vertex of G. Then Algorithm 4.2 computes the global coloring $\mathfrak{d}_{|S_v}(W)^*$ by taking G, W, and v_0 as input.

Proof. In Step 2 we define the BFS-order in which the vertices will be processed and store this sequence in Q. In Step 4 we assign pairwise different temporary global colors to all edges that are incident with v_0. In Step 5 we iterate over all vertices of the given induced connected subgraph $\langle W\rangle$ of G. For every vertex we execute Algorithm 4.1. Lemma 4.8 implies that in the first iteration we correctly compute the local colors for S_{v_0}, and consequently also $\mathfrak{d}_{|S_v}(\{v_0\})^*$. Obviously, whenever we merge two temporary local colors of two primal edges in the first iteration, then we also merge their temporary global colors. Consequently, the resulting temporary global colors correspond to the global coloring $\mathfrak{d}_{|S_v}(\{v_0\})^*$ after the first iteration. Lemma 4.8 implies that after all iterations are performed, that is, all vertices in Q are processed, the resulting temporary global colors correspond to $\mathfrak{d}_{|S_v}(W)^*$ for the given input set $W \subseteq V(G)$. □

For the global coloring, Theorem 2.3 implies that $\mathfrak{d}_{|S_v}(V(G))^* = \delta^*_G$. This leads immediately to the following theorem.

Theorem 4.10. *Let G be a connected graph and v_0 an arbitrary vertex of G. Then Algorithm 4.2 computes the global coloring δ^*_G by taking G, $V(G)$, and v_0 as input.*

4.2 Time Complexity

We begin with the complexity of merging colors. We have global and local colors, and will define *local* and *global color graphs*. Both graphs are acyclic temporary structures. Their vertex sets are the sets of temporary colors in the initial state. In this state the color graphs have no edges. Every component is a single vertex and corresponds to an initial temporary color. Recall that we color edges of graphs, for example the edges of G or S_v. The color of an edge is indicated by a pointer to a vertex of the color graph. These pointers are not changed, but the colors will correspond to the components of the color graph. When two colors are merged, then this will be reflected by adding an edge between their respective components.

The color graph is represented by an adjacency list as described in [5, Chapter 17.2] or [9, pp. 34 -37]. Thus, working with the color graph needs $O(k)$ space when k colors are used. Furthermore, for every vertex of the color graph we keep an index of the connected component in which the vertex is contained. We also store the actual size of every component, that is, the number of vertices of this component.

Suppose we wish to merge temporary colors of edges e and f that are identified with vertices a, respectively b, in the color graph. We first check whether a and b are contained in the same connected component by comparing component indices. If the component indices are the same, then e and f already have the same color, and no action is necessary. Otherwise we insert an edge between a and b in the color graph. As this merges the components of a and b we have to update component indices and the size. The size is updated in constant time. For the component index we use the index of the larger component. Thus, no index change is necessary for the larger component, but we have to assign the new index to all vertices of the smaller component.

Notice that the color graph remains acyclic, as we only add edges between different components.

Lemma 4.11. *Let $G_0 = (V, E)$ be a graph with $V = \{v_1, \ldots, v_k\}$ and $E = \emptyset$. The components of G_0 consist of single vertices. We assign component index j to every component $\{v_j\}$. For $i \in \{1, \ldots, k-1\}$ let G_{i+1} denote the graph that results from G_i by adding an edge between two distinct connected components, say C and C'. If $|C| \leq |C'|$, we use the component index of C' for the new component and assign it to every vertex of C.*

Then every G_i is acyclic, and the total cost of merging colors is $O(k \log_2 k)$.

Proof. Acyclicity is true by construction.

A vertex is assigned a new component index when its component is merged with a larger one. Thus, the size of the component at least doubles at every such step. Because the maximum size of a component is bounded by k, there can be at most $\log_2 k$ reassignments of the component index for every vertex. As there are k vertices, this means that the total cost of merging colors is $O(k \log_2 k)$. □

The color graph is used to identify temporary local, resp., global colors. Based on this, we now define the *local* and *global color graph*.

Assigned labels of the vertices of the global color graph are stored in the edge list, where any edge is identified with at most one such label. Notice that the original graph is represented by an extended adjacency list, where for any vertex and its neighbor a reference to the edge (in the edge list) that connects them is stored. This reference allows to access a global temporary color from adjacency list in constant time.

In every iteration of Algorithm 4.2, we recognize the PSP for one vertex by calling Algorithm 4.1. In the following paragraph we introduce several temporary attributes and matrices that are used in the algorithm.

Suppose we execute an iteration that recognizes some PSP S_c. To indicate whether a vertex was treated in this iteration we introduce the attribute $visited$, that is, when vertex v is visited in this iteration we set $v.visited = c$. Any value different from c means that vertex v was not yet treated in this iteration. Analogously, we introduce the attribute $primal$ to indicate that a vertex is adjacent to the current center c. The attribute $tempLabel$ maps primal vertices to the indices of rows and columns of the matrices $incidenceList$ and $absenceList$. For any vertex v that is at distance two from the center c we store its first and second primal neighbor w_1 and w_2 in the attributes $FirstPrimalNeighbor$ and $SecondPrimalNeighbor$. Furthermore, we need to keep the position of vw_1 and vw_2 in the edge list to get their temporary global colors. For this purpose, we use attributes $firstEdge$ and $secondEdge$. Attribute $mapLocalColor$ helps us to map temporary local colors to the vertices of the global color graph. Any vertex that is at distance two from the center and has a least two primal neighbors is a candidate for a non-primal vertex. We insert them to the $stack$. The temporary structures help to access the required information in constant time:

- $v.visited = c$
  vertex v has been already visited in the current iteration.
- $v.primal = c$
  vertex v is adjacent to center c.
- $incidenceList[v.tempLabel, u.tempLabel] = 0$
  pair of primal edges (cv, cu) is missing in the incidence list.
- $absenceList[v.tempLabel, u.tempLabel] = 1$
  pair of primal edges (cv, cu) was inserted to the absence list.

- $v.firstPrimalNeighbor = u$
  u is the first recognized primal neighbor of the non-primal vertex v.
- $v.firstEdge = e$
  edge e joins the non-primal vertex v with its first recognized primal neighbor (it is used to get the temporary global color from the edge list).
- $b.mapLocalColor = d$
  local color b is mapped to temporary global color d (i.e. there exists an edge that is colored by both colors).

Note that the temporary matrices $incidenceList$ and $absenceList$ have dimension $\deg(c) \times \deg(c)$ and that all their entries are set to zero in the beginning of every iteration.

Theorem 4.12. *For a given connected graph $G = (V, E)$ with maximum degree Δ and $W \subseteq V$, Algorithm 4.2 runs in $O(|E|\Delta)$ time and $O(|E| + \Delta^2)$ space.*

Proof. Let G be a given graph with m edges and n vertices. In Step 1 of Algorithm 4.2 we initialize all temporary attributes and matrices. This consumes $O(m+n) = O(m)$ time and space, since G is connected, and hence, $m \geq n - 1$. Moreover, we set all temporary colors of edges in the edge list to zero, which does not increase the time and space complexity of the initial step. Recall that we use an extended adjacency list, where every vertex and its neighbors keep the reference to the edge in the edge list that connects them. To create an extended adjacency list we iterate over all edges in the edge list, and for every edge $uv = e \in E(G)$ we set a new entry for the neighbor v for u and, simultaneously, we add a reference $v.edge = e$. The same is done for vertex v. It can be done in $O(m)$ time and space.

In Step 2 of Algorithm 4.2, we build a sequence of vertices in BFS-order starting with v_0, which is done in $O(m + n)$ time in general. Since G is connected, the BFS-ordering can be computed in $O(m)$ time. Step 3 takes constant time. In Step 4 we initialize the global color graph that has $\deg(v_0)$ vertices (bounded by Δ in general). As we already showed, all operations on the global color graph take $O(\Delta \log_2 \Delta)$ time and $O(\Delta)$ space. We proceed to traverse all neighbors $u_1, u_2, \ldots, u_{\deg(v_0)}$ of the root $v_0 \in V(G)$ (via the adjacency list) and assign them unique labels $1, 2, \ldots, \deg(v_0)$ in edge list, that is, every edge v_0u_i gets the label i. In this way, we initialize pairwise different temporary global colors of edges incident with v_0 , that is, to vertices of the global color graph. Using the extended adjacency list, we set the label to an edge in the edge list in constant time. In Step 5 we run Algorithm 4.1 for any vertex from the defined BFS-sequence.

In the remainder of this proof, we will focus on the complexity of Algorithm 4.1. Suppose we perform Algorithm 4.1 for vertex c to recognize the PSP S_c. The recognition process is based on temporary structures. We do not need to reset any of these structures, for any execution of Algorithm 4.1 for a new center c, except $absenceList$ and $incidenceList$. This is done in Step 1. Further, we set here the attribute $tempLabel$ for every primal vertex v, such that every vertex has assigned a unique number from $\{1, 2, \ldots, \deg(c)\}$. Finally, we traverse all neighbors of the center c and for each of them we set $primal$ to c. Hence, the initial step of Algorithm 4.1 is done in $O(\deg(c)^2)$ time.

Step 2a is performed for every neighbor of every primal vertex. The number of all such neighbors is at most $\deg(c)\Delta$. For every treated vertex, we set attribute $visited$ to c.

This allows us to verify in constant time that a vertex was already visited in the recognition subroutine Algorithm 4.1.

If the condition in Step 2(a)i is satisfied, then we add primal edges cu and cw to the absence list. By the previous arguments, this can be done in constant time by usage of $tempLabel$ and $absenceList$.

If the condition in Step 2(a)ii is satisfied, we set vertex u as first primal neighbor of vertex w. For this purpose, we use the attribute $firstPrimalNeighbor$. We also set $w.firstEdge = e$, where e is a reference to the edge in the edge list that connects u and w. This reference is obtained from the extended adjacency list in constant time. Recall, the edge list is used to store the labels of vertices of the global color graph for the edges of a given graph, that is, the assignment of temporary global colors to the edges. Using $w.firstEdge$, we are able to directly access the temporary global color of edge uw in constant time.

Step 2(a)iii is performed when we try to visit a vertex w from some vertex u where w has been already visited before from some vertex v. If v is the only recognized primal neighbor of w, then we perform analogous operations as in the previous step. Moreover. if (cu, cv) is not contained in the incidence list, then we set u as second primal neighbor of w, add (cu, cv) to the incidence list and add w to the stack. Otherwise we add (cu, cv) to the absence list. The number of operations in this step is constant.

If w has more recognized primal neighbors we process case B. Here we just add all pairs formed by cv_1, cv_2, cu to absence list. Again, the number of operations is constant by usage of $tempLabel$ and matrices $incidenceList$ and $absenceList$.

In Step 3 we assign pairwise different temporary local colors to the primal edges. Assume the neighbors of the center c are labeled by $1, 2, \ldots, \deg(c)$, then we set value $u.tempLabel$ to cu. In Step 4a we iterate over all entries of the $absenceList$. For all pairs of edges that are in the absence list we check whether they still have different temporary local colors and if so, we merge their temporary local colors by adding a respective edge in the local color graph. Analogously we treat all pairs of edges contained in the $incidenceList$ in Step 4b. Here we merge temporary local colors of primal edges cu and cv when the pair (cu, cv) is missing. To treat all entries of the $absenceList$ and $incidenceList$ we need to perform $\deg(c)^2$ iterations. Recall, the temporary local color of the primal edge cu is equal to the index of the connected component in the local color graph, in which vertex $u.tempLabel$ is contained. Thus, the temporary local color of this primal edge can be accessed in constant time. As we already showed, the number of all operations on the local color graph is bounded by $O(\deg(c) \log_2 \deg(c))$. Hence, the overall time complexity of both Steps 3 and 4 is $O(\deg(c)^2)$.

In Step 5 we map temporary local colors of primal edges to temporary global colors. For this purpose, we use the attribute $mapLocalColor$. The temporary global color of every edge can be accessed by the extended adjacency list, the edge list and the global color graph in constant time. Since we need to iterate over all primal vertices, we can conclude that Step 5 takes $O(\deg(c))$ time.

In Step 6 we perform analogous operations for any vertex from Stack $\mathbb{T}_c$ as in Step 5. In the worst case, we add all vertices that are at distance two from the center to the stack. Hence, the size of the stack is bounded by $O(\deg(c)\Delta)$. Recall that the first and second primal neighbor w_1 and w_2 of every vertex v from the stack can be directly accessed by the attributes $firstPrimalNeighbor$ and $secondPrimalNeighbor$. On the other hand, the temporary global colors of non-primal edges vw_1 and vw_2 can be accessed directly by the

attributes $firstEdge$ and $secondEdge$. Thus, all needful information can be accessed in constant time. Consequently, the time complexity of this step is bounded by $O(\deg(c)\Delta)$.

In the last step, Step 7, we iterate over all edges of the recognized PSP. Note, the list of all primal edges can be obtained from the extended adjacency list. To get all non-primal edges we iterate over all vertices from the stack and use the attributes $firstEdge$ and $secondEdge$, which takes $O(\deg(c)\Delta)$ time. The remaining operations can be done in constant time.

To summarize, Algorithm 4.1 runs in $O(\deg(c)\Delta)$ time. Consequently, Step 5 of Algorithm 4.2 runs in $O(\sum_{c\in W}\deg(c)\Delta) = O(m\Delta)$ time, which defines also the total time complexity of Algorithm 4.2. The most space consuming structures are the edge list and the extended adjacency list ($O(m)$ space) and the temporary matrices $absenceList$ and $incidenceList$ ($O(\Delta^2)$ space). Hence, the overall space complexity is $O(m+\Delta^2)$. □

Since quasi Cartesian products are defined as graphs with non-trivial δ^*, Theorem 4.10 and 4.12 imply the following corollary.

Corollary 4.13. *For a given connected graph $G = (V, E)$ with* bounded *maximum degree Algorithm 4.2 (with slight modifications) determines whether G is a quasi Cartesian product in $O(|E|)$ time and $O(|E|)$ space.*

4.3 Parallel Processing

The local approach allows the parallel computation of $\delta^*(G)$ on multiple processors. Consider a graph G with vertex set $V(G)$. Suppose we are given a decomposition of $V(G) = W_1 \cup W_2 \cup \cdots \cup W_k$ into k parts such, that $|W_1| \approx |W_2| \approx \cdots \approx |W_k|$, where the subgraphs induced by $W_1, W_2, \ldots, W_k$ are connected, and the number of edges whose endpoints lie in different partitions is small (we call such a decomposition *good*).

Algorithm 4.3 (*Parallel recognition of δ^**)

Input: *A graph G, and a good decomposition $V(G) = W_1 \cup W_2 \cup \cdots \cup W_k$.*
Output: *Relation δ^*_G.*

1. *For every partition W_i concurrently compute global coloring $\mathfrak{d}_{|S_v}(W_i)$ ($i \in \{1, 2, \ldots, k\}$):*
   (a) *Take all vertices of W_i and order them in BFS to get sequence Q_i.*
   (b) *Set $W' := \emptyset$.*
   (c) *Assign pairwise different temporary global colors to edges incident to first vertex in Q_i.*
   (d) *For any vertex v from sequence Q_i do:*
      i. *Use Algorithm 4.1 to compute $\mathfrak{d}_{|S_v}(W' \cup v)^*$.*
      ii. *Move all edges that were treated in previous step and have at least one endpoint not in partition W_i to stack $\mathbb{T}_i$.*
      iii. *Add v to W'.*
2. *Run concurrently for every partition W_i to merge all global colorings ($i \in \{1, 2, \ldots, k\}$):*
   (a) *For each edge from stack $\mathbb{T}_i$, take all its assigned global colors and merge them.*

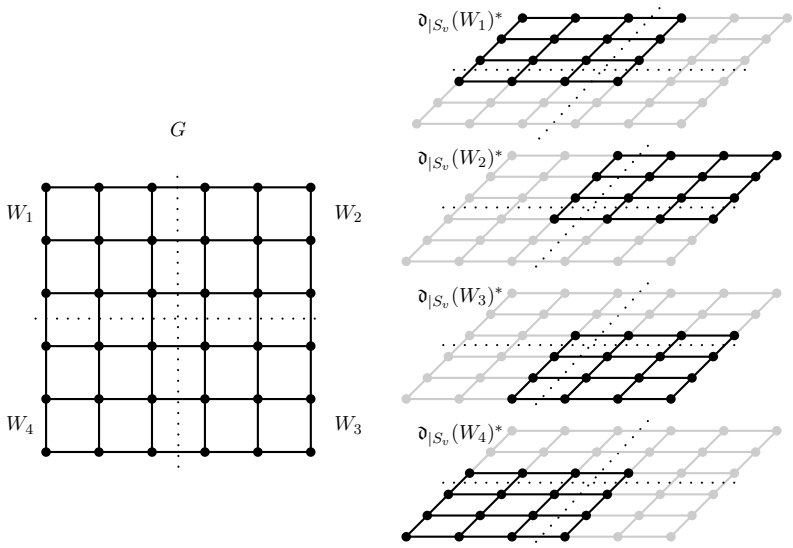


Figure 3: Example - Parallel recognition of δ^*.

Then algorithm 4.2 can be used to compute the colorings $\mathfrak{d}_{|S_v}(W_1)^*, \mathfrak{d}_{|S_v}(W_2)^*, \ldots, \mathfrak{d}_{|S_v}(W_k)^*$, where every instance of the algorithm can run in parallel. The resulting global colorings are used to compute $\mathfrak{d}_{|S_v}(V(G))^* = (\mathfrak{d}_{|S_v}(W_1)^* \cup \mathfrak{d}_{|S_v}(W_2)^* \cup \cdots \cup \mathfrak{d}_{|S_v}(W_k)^*)^*$. The sketch of the parallelization is summarized in Algorithm 4.3.

Figure 3 shows an example of decomposed vertex set of a given graph G. The computation of global colorings associated with the individual sets of the partition can be done then in parallel. The edges that are colored by global color when the partition is treated are highlighted by bold black color. Thus we can observe that many edges will be colored by more then one color.

Notice that we do not treat the task of finding a good partition. With the methods of [4] this is possible with high probability in $O(\log n)$ time, where n is the number of vertices.

References

[1] W. Dörfler, Some results on the reconstruction of graphs, in: *Infinite and finite sets (Colloq., Keszthely, 1973; dedicated to P. Erdős on his 60th birthday), Vol. I*, North-Holland, Amsterdam, pp. 361–363. Colloq. Math. Soc. János Bolyai, Vol. 10, 1975.

[2] J. Feigenbaum, *Product graphs: some algorithmic and combinatorial results*, Technical Report STAN-CS-86-1121, Stanford University, Computer Science, 1986, phD Thesis.

[3] J. Hagauer and J. Žerovnik, An algorithm for the weak reconstruction of Cartesian-product graphs, *J. Combin. Inform. System Sci.* **24** (1999), 87–103.

[4] S. Halperin and U. Zwick, Optimal randomized EREW PRAM algorithms for finding spanning forests, *J. Algorithms* **39** (2001), 1–46, doi:10.1006/jagm.2000.1146.

[5] R. Hammack, W. Imrich and S. Klavžar, *Handbook of Product Graphs*, Discrete Mathematics and its Applications, CRC Press, 2nd edition, 2011.

[6] M. Hellmuth, A local prime factor decomposition algorithm, *Discrete Math.* **311** (2011), 944–965.

[7] M. Hellmuth, W. Imrich, W. Klöckl and P. F. Stadler, Approximate graph products, *European J. Combin.* **30** (2009), 1119 – 1133.

[8] M. Hellmuth, W. Imrich and T. Kupka, Partial star products: A local covering approach for the recognition of approximate Cartesian product graphs, *Math. Comput. Sci* **7** (2013), 255–273.

[9] W. Imrich and S. Klavžar, *Product graphs*, Wiley-Interscience Series in Discrete Mathematics and Optimization, Wiley-Interscience, New York, 2000.

[10] W. Imrich and I. Peterin, Recognizing Cartesian products in linear time, *Discrete Math.* **307** (2007), 472 – 483.

[11] W. Imrich, T. Pisanski and J. Žerovnik, Recognizing Cartesian graph bundles, *Discrete Math.* **167-168** (1997), 393–403.

[12] W. Imrich and J. Žerovnik, Factoring Cartesian-product graphs, *J. Graph Theory* **18** (1994), 557–567, doi:10.1002/jgt.3190180604.

[13] T. Kupka, *A local approach for embedding graphs into Cartesian products*, Ph.D. thesis, VSB-Technical University of Ostrava, 2013.

[14] T. Pisanski, B. Zmazek and J. Žerovnik, An algorithm for k-convex closure and an application, *Int. J. Comput. Math.* **78** (2001), 1–11, doi:10.1080/00207160108805092.

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