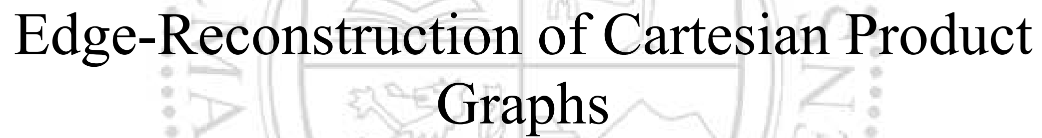
Chair of Applied Mathematics

Doctoral Thesis



Edge-Reconstruction of Cartesian Product Graphs

Marcin Jacek Wardynski

May 2020

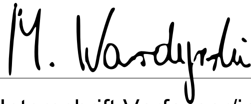
EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe.

Ich erkläre, dass ich die Richtlinien des Senats der Montanuniversität Leoben zu "Gute wissenschaftliche Praxis" gelesen, verstanden und befolgt habe.

Weiters erkläre ich, dass die elektronische und gedruckte Version der eingereichten wissenschaftlichen Abschlussarbeit formal und inhaltlich identisch sind.

Datum 28.05.2020



Unterschrift Verfasser/in
Marcin Jacek, Wardynski

There are, roughly speaking, two kinds of mathematical creativity.
One, akin to conquering a mountain peak, consists of solving a
problem which has remained unsolved for a long time and has
commanded the attention of many mathematicians.
The other is exploring new territory.

— Marek Kac

ABSTRACT

This thesis solves the weak edge-reconstruction problem for Cartesian products. In other words, it is shown that any nontrivial finite or infinite Cartesian product G with edge-set $E(G)$ is uniquely determined up to isomorphisms by any edge-deleted subgraph $\{G - e | e \in E(G)\}$. For finite Cartesian products the thesis also presents an algorithm for the computation of G from $G - e$ in $O(m\Delta^2)$ time, where m is the size of G and Δ its maximal degree. It improves the straightforward algorithm for reconstruction of G, which has complexity $O(mn^2)$, where n is the order of G. Because Δ can be much smaller than n, the improvement can be substantial. The algorithm needs a thoughtful analysis of the properties of well-known Cartesian product relations, like Θ, τ and δ, as well as some typical properties of the Cartesian product itself. The analysis leads to the introduction of a new relation $\bar{\tau}$, which modifies τ and which is essential for the improvement of the reconstruction's complexity. The algorithm is presented in two steps. The first contains a detailed description of its structure and complexity, and the second contains pseudocode for all its parts, together with the needed data structures.

The analogous problem for weak vertex-reconstruction of nontrivial finite or infinite Cartesian products was solved 1996 in [15]. The paper is not algorithmic. An algorithm for the reconstruction for finite Cartesian products was later provided by [6]. It contains a subtle error in the computation of its complexity. As it is frequently cited it will be discussed and corrected it in the Appendix.

The methods of the thesis can also be used to improve the complexity of weak vertex-reconstruction for finite Cartesian products to $O(m\Delta^2)$, which is the same as that for weak edge-reconstruction. I intend to publish the algorithm separately.

PUBLICATIONS

The thesis presents several results that were already introduced in the author's publications. For more details see [13] and [12].

ACKNOWLEDGMENTS

First and foremost, I would like to thank my supervisor, professor Wilfried Imrich, who introduced me to the subject of graph products and invited to the joint research for some new properties of them. Only thanks to his insightful suggestions and support, the reader can hold this work in her hands.

Besides my supervisor, I would like to express my gratitude to my family and friends, who encouraged me in the moments of doubt and pushed towards my goals, sometimes applying varied means.

CONTENTS

I INTRODUCTION
1 INTRODUCTION 3

II PRELIMINARIES
2 PRELIMINARIES 7
- 2.1 Cartesian product 7
- 2.2 Weak Cartesian product 8
- 2.3 Prime factorization and automorphisms 8
- 2.4 Convexity 9
- 2.5 Product relations and product colorings 9
- 2.6 Prime factorizations and the relations σ, Θ and τ 10
- 2.7 The relations δ and an alternate relation τ 10
- 2.8 Twisted Cartesian products 12

III RECONSTRUCTION
3 RECONSTRUCTION 17
- 3.1 Edge-deleted nontrivial Cartesian products are prime 17
- 3.2 Reconstruction when K_2 is not a factor of G 18
- 3.3 Reconstruction when G contains a factor K_2 22
- 3.4 Conclusion 25

IV ALGORITHM
4 ALGORITHM 29
- 4.1 Reconstruction in $O(mn^2)$ time 29
- 4.2 Reconstruction in $O(m\Delta^2)$ time 29
  - 4.2.1 Preprocessing and computation of necessary relations 29
  - 4.2.2 $G - e$ contains edges that are in no chordless squares 30
  - 4.2.3 Each edge of $G - e$ is in a chordless square 31

V ALGORITHM - DETAILED DESCRIPTION
5 ALGORITHM - DETAILED DESCRIPTION 37
- 5.1 Store All Squares 37
  - 5.1.1 Correctness 38
  - 5.1.2 Complexity 38
- 5.2 Edge without square reconstruction 39
  - 5.2.1 Correctness 41
  - 5.2.2 Complexity 41
- 5.3 General Reconstruction 41
- 5.4 Color edges 41

5.4.1 Correctness 44
5.4.2 Complexity 44
5.5 Color squares 44
5.5.1 Correctness 48
5.5.2 Complexity 48
5.6 Refine coloring 49
5.6.1 Correctness 49
5.6.2 Complexity 49
5.7 Clean Missing Square Edge Pairs 50
5.7.1 Correctness 50
5.7.2 Complexity 51
5.8 Define Endpoints of the edge to reconstruct 51
5.8.1 Correctness 53
5.8.2 Complexity 55

VI SUMMARY

6 SUMMARY 59
6.1 Results 59
6.2 Open problems 59

VII APPENDIX

7 WEAK RECONSTRUCTION COMPLEXITY OF CARTESIAN PRODUCTS 63
7.1 Preamble 63
7.2 Introduction 63
7.3 Preliminaries 64
7.4 Complexity analysis 66

BIBLIOGRAPHY 71

LIST OF FIGURES

Figure 2.1 $S_4 \square K_2$ 12
Figure 2.2 Möbius strip 13
Figure 2.3 Different numbers of vertices in components of a twisted Cartesian product 13
Figure 3.1 Product square $abcd$ in G, with removed edge $e = ad$ 17
Figure 3.2 $G - e$ prime, aa' hac color c_B. 18
Figure 3.3 $G - e$ prime, aa' has color c_A. 19
Figure 3.4 Inserted new edge $f = ad$ 19
Figure 3.5 The induced subgraph R of G and the edge ad 20
Figure 3.6 Missing edge $e = aa'$ 21
Figure 3.7 Missing edge $e = a'b'$ 21
Figure 3.8 Missing edge $e = bb'$ 21
Figure 3.9 Missing edge $e = b'c'$ 22
Figure 3.10 Missing edge $e = b'c'$ and no factor K_2 23
Figure 3.11 Missing edge $e = b'c'$ where $B = K_2$ 23
Figure 3.12 $G - b'c' + ad$, compare Fig. 3.11 for the subgraph spanned by $\{a, b, c, d, a', b', c, d'\}$ 24
Figure 3.13 $A - bc = N \square Y$ and $X - bc = N \square B$, edges of N in bold 25
Figure 4.1 e incident to uv in $G - e$ 30
Figure 4.2 e opposite to uv in $G - e$ 31
Figure 4.3 Lemma 8, G has at least three factors 31
Figure 4.4 Lemma 8, the edges ab and aa' have the same color 32
Figure 4.5 Lemma 8, $f = ab$ and $p = aa'$ of different colors – part 1 32
Figure 4.6 Lemma 8, $f = ab$ and $p = aa'$ of different colors – part 2 33
Figure 4.7 Product of a star S_2 by a cube from which an edge was deleted. 34
Figure 5.1 Special cases having no squares 39
Figure 5.2 Missing edge $e = ad$ 44
Figure 5.3 Looking for square $bb'cc'$ 48
Figure 5.4 Found missinge square edges triple for edges uv, uw, wv' 54
Figure 7.1 Multiple cross-edges. 67
Figure 7.2 Step 2.1 – y_1^1 and y_k^1 recognized as up-neighbors of x 68
Figure 7.3 Step 2.2 – y_1^1 and y_k^1 are good candidates 68
Figure 7.4 Step 2.3 – y_1^1 and y_k^1 are not bad candidates. 69

LIST OF ALGORITHMS

Algorithm 1 Algorithm overview 37
Algorithm 2 Store All Squares 37
Algorithm 3 Reconstruction for edge without square 40
Algorithm 4 General Reconstruction 41
Algorithm 5 Color edges 42
Algorithm 6 Order edges 43
Algorithm 7 Color Squares 45
Algorithm 8 Find Other Color 47
Algorithm 9 Merge Colors By Tau 50
Algorithm 10 Clean Missing Square Edge Pairs 51
Algorithm 11 Find endpoints of the edge to reconstruct 52
Algorithm 12 Group missing square edges by first edge and color 52
Algorithm 13 Find Potential Missing Edge Endpoints 53
Algorithm 14 Update missing square edges color grouping 53
Algorithm 15 Skeleton of the algorithm 64
Algorithm 16 Construction 2 65

Part I

INTRODUCTION

1 INTRODUCTION

In 1942 Kelly [16] conjectured that any finite graph on at least 3 vertices is uniquely determined by the multiset of its subgraphs obtained by deleting a vertex and all edges adjacent to it. This is known as the *reconstruction conjecture*. It became popular after 1960, when Ulam [23] asked whether a graph on at least three vertices is determined by its vertex deleted subgraphs. For infinite graphs the conjecture is false, but for finite graphs it is still open, despite the fact that it holds for large classes of graphs. For example, it is true for nontrivial Cartesian products, as has been shown by Dörfler [3].

If one knows that a graph G is a nontrivial Cartesian product, then G can be reconstructed from an arbitrary vertex deleted subgraph of G. This result is due to Sims [21], and was presented in terms of semistability of Cartesian products in [22]. Because of the additional information that the given graph is a vertex deleted Cartesian product, one speaks of *weak reconstruction*. This was extended by Imrich and Žerovnik [15], who showed that the weak reconstruction problem can be solved from a single vertex-deleted subgraph for nontrivial, connected finite or infinite Cartesian products.

There is an algorithm from 1999 due to Hagauer and Žerovnik [6] for the weak reconstruction of finite, connected nontrivial Cartesian products. Its complexity, see [11, 17], is now $O(mn + \Delta^2(m + \Delta^4))$, where n denotes the order of G. With methods that are also used here, see [11], it can be improved to $O(m\Delta^2)$, which is the same as the complexity of the algorithm for weak edge reconstruction in this dissertation.

An edge deleted subgraph of a graph G is formed by deletion of exactly one edge from G. It has the same set of vertices as G. In 1964 Harary [8] conjectured that any two graphs on at least four edges and the same decks of edge deleted subgraphs are isomorphic. This is known as the *edge-reconstruction conjecture*. Just as the reconstruction conjecture it is known to hold for several classes of graphs, in particular for graphs on more than $n(\log_2 n - 1)$ edges, see [19]. For products this was taken up by Dörfler [2], who showed that all nontrivial strong products and certain lexicographic products can be reconstructed from the deck of all edge-deleted subgraphs. He did not treat the edge-reconstruction of Cartesian products.

More important for us is the fact that reconstructability implies edge-reconstructability. This has been shown by Greenwell [5]. By the result of Dörfler [3] about the reconstructability of nontrivial Cartesian products it implies that they are also edge-reconstructible.

Here we show that nontrivial connected Cartesian products are weakly edge-reconstructible, that is, each finite or infinite connected Cartesian product G is uniquely determined up to isomorphisms by any edge deleted subgraph $\{G - e \mid e \in E(G)\}$. Further we show, that from the algorithmic point of view a straightforward algorithm checking all possibilities can deliver a result in $O(mn^2)$ time.

Using more sophisticated methods we then reduce the needed time to $O(m\Delta^2)$. This algorithm and its details comprise the bulk of the dissertation.

Part II

PRELIMINARIES

2 PRELIMINARIES

All graphs considered in this dissertation are finite or infinite undirected graphs without loops or multiple edges. If G is a graph, we shall write $V(G)$ for its vertex set and $E(G)$ for its edge set. $E(G)$ shall be considered as a set of unordered pairs xy of distinct vertices x, y of G.

We begin by collecting the main results and concepts about products that are used in the dissertation. For more detailed information we refer to [7].

2.1 CARTESIAN PRODUCT

The vertex set of the *Cartesian product* $G_1 \Box G_2$ of two graphs G_1 and G_2 is $V(G_1) \times V(G_2)$. Two vertices (u_1, u_2) and (v_1, v_2) are adjacent precisely if $u_1v_1 \in E(G_1)$ and $u_2 = v_2$, or if $u_1 = v_1$ and $u_2v_2 \in E(G_2)$. Hence,

$$\begin{aligned} V(G_1 \Box G_2) &= \{(v_1, v_2) \mid v_1 \in V(G_1) \text{ and } v_2 \in V(G_2)\}, \\ E(G_1 \Box G_2) &= \{(u_1, u_2)(v_1, v_2) \mid u_1v_1 \in E(G_1), u_2 = v_2, \text{ or} \\ &\qquad\qquad u_2v_2 \in E(G_2), u_1 = v_1\}. \end{aligned}$$

Cartesian multiplication has K_1 as a unit, is commutative and associative. This means, if we are given k graphs $G_i, i \in [1:k]$, then we can simply write $G_1 \Box \cdots \Box G_k$ for their product, regardless of the sequence of the factors or the order in which the multiplications are performed. Moreover, we can identify the vertices of $G_1 \Box \cdots \Box G_k$ with the vectors $(x_1, \ldots x_k)$, where $x_i \in V(G_i)$. Then two vertices $x = (x_1, \ldots x_k)$ and $y = (y_1, \ldots y_k)$ are adjacent exactly if there is a $j \in [1:k]$ such that $x_jy_j \in V(G_j)$ and $x_i = y_i$ for $i \in [1:k], i \neq j$.

We call the x_i the *coordinates* of x, and observe that two vertices in a product are adjacent if and only if they differ in exactly one coordinate. We also call x_i the *projection* $p_i(x)$ of x to $V(G_i)$.

It is easy to extend the definition to infinitely many factors. Let G_ι, $\iota \in I$, be a finite or infinite set of graphs and X the set of all functions $x : I \to \bigcup_{\iota \in I} V(G_\iota)$ where $x : \iota \mapsto V(G_\iota)$. Then the Cartesian product

$$G = \prod_{\iota \in I} G_\iota$$

of the graphs $G_\iota, \iota \in I$, has X as its set of vertices and the edges xy are defined as those pairs of vertices x, y, for which there exists an index $\kappa \in I$ such that $x_\kappa y_\kappa \in E(G_\kappa)$ and $x_\iota = y_\iota$ for all $\iota \in I \setminus \{\kappa\}$. We call $x(\iota)$ the *ι-coordinate of* x and also denote it by x_ι.

For finite I this definition coincides with definition of the Cartesian product of finitely many factors.

Another important concept is the notion of *layers*. Let $G = G_1 \square \cdots \square G_k$ be a product of graphs. For any given vertex $a = (a_1, \ldots, a_k)$ in $G_1 \square \cdots \square G_k$ the set of vertices

$$\{(a_1, \ldots a_{i-1}, x, a_{i+1}, \ldots, a_k) \mid x \in V(G_i)\}$$

induces a subgraph of G that is isomorphic to G_i. We call it the *G_i-layer* G_i^a through a.

2.2 WEAK CARTESIAN PRODUCT

It is well known that the product of finitely many graphs is connected if and only if every factor is connected. However, a product of infinitely many nontrivial graphs must be disconnected because it contains vertices differing in infinitely many coordinates. No two such vertices can be connected by a path of finite length, because every edge connects vertices differing in exactly one coordinate.

This gives rise to the notion of the so-called weak Cartesian product: let $a \in V(\prod_{\iota \in I} G_\iota)$. Then the *weak Cartesian product*

$$G = \prod_{\iota \in I}^{a} G_\iota$$

is the connected component of $G = \prod_{\iota \in I} G_\iota$ containing a. Note that $\prod^a G_i = \prod^b G_i$, if and only if a and b differ in at most finitely many coordinates, and that all G_ι are factors of G. Hence the concept of layers naturally extends to the weak Cartesian product.

2.3 PRIME FACTORIZATION AND AUTOMORPHISMS

A nontrivial graph is called *prime*, if it cannot be represented as the Cartesian product of two factors on at least two vertices. Every connected, nontrivial finite or infinite graph G has a representation as a Cartesian or weak Cartesian product of prime graphs, which is unique in the following strong sense: there exists a unique partition $\mathcal{P} = \{E_\iota \mid \iota \in I\}$ of $E(G)$, where I is a finite or infinite index set, such that each E_ι spans a subgraph of G, say H_ι, whose connected components are the layers of a factor, say G_ι, of G. Furthermore, the G_ι are prime, and G is the Cartesian product $\prod_{\iota \in I} G_\iota$ if I is finite, or the weak Cartesian product $\prod_{\iota \in I}^a G_\iota$, for an appropriate $a \in V(\prod_{\iota \in I} G_\iota)$, if I is infinite.

Every automorphism of G preserves the partition $\mathcal{P}$, but may permute its sets. In fact, for every automorphism φ of $\prod_{\iota \in I}^a G_\iota$ there exists a permutation π of I, together with isomorphisms $\varphi_\iota : G_{\pi(\iota)} \to G_\iota$ such that

$$\varphi(x)_\iota = \varphi_\iota(x_{\pi(\iota)}).$$

Please note that this only holds for products of connected prime graphs.

For finite graphs these results are due to Sabidussi [20], for infinite graphs to Miller [18] and Imrich [9].

A special case of them we will need in Section 3.3.

2.4 CONVEXITY

A helpful property for our arguments is convexity. A subgraph $W \subseteq G$ is *convex in G* if every shortest G-path between vertices of W lies entirely in W. Convex subgraphs in products are characterized by the following lemma.

Proposition 1 (Lemma 6.5 [7]). *A subgraph W of $G = G_1 \square \ldots \square G_k$ is convex if and only if $W = U_1 \square \ldots \square U_k$, where each U_i is convex in G_i.*

Every layer G_i^a is convex in G, because

$$G_i^a = \{a_1\} \square \cdots \square \{a_{i-1}\} \square G_i \square \{a_{i+1}\} \square \cdots \square \{a_k\},$$

where $\{a_j\}$ denotes the subgraph of G_j consisting of the single vertex $a_j \in V(G_j)$.

2.5 PRODUCT RELATIONS AND PRODUCT COLORINGS

With every representation of a graph G as a Cartesian product $G_1 \square \cdots \square G_k$ we associate a *product relation* $c(G_1 \square \cdots \square G_k)$, or simply c, on $E(G)$. We say edges e, f are in the relation c if their endpoints differ in the same coordinate. Clearly c is an equivalence relation with k equivalence classes, each of which corresponds to a factor G_i of G. We color the edges of the ith equivalence class with color i and call this the *product coloring* of $G_1 \square \cdots \square G_k$.

The edges of color i induce a subgraph of G whose connected components are isomorphic to G_i. We call them the *G_i-layers* of G. In [7, Lemma 6.3] it was shown that to any two incident edges e and f of a Cartesian product $G_1 \square \cdots \square G_k$ that are in different layers, that is, one in a G_i-layer and the other one in a G_j-layer, where $i \neq j$, there exists exactly one square $efgh$ containing e and f, and that this square has no diagonals.

In the language of product colorings this means that to any two incident edges e and f of a Cartesian product that have different product colors, there exists exactly one square containing e and f, and that this square has no diagonals. This is called the *square property*. Squares without diagonals are also called *chordless*. Chordlees squares $efgh$, where e, f have different colors, are called *product squares*. It is easily seen, but also shown in [7], that opposite edges of product squares have the same color.

For further reference we also observe the following fact: If there is an edge uv, whose endpoints u, v are in different layers with respect to the same factor, say in a G_i-layer G_i^u through u, and a G_i-layer $G-i^v$ through v, then the edges between G_i^u and G_i^v induce an isomorphism. This follows immediately from the definitions of the Cartesian product and its layers.

2.6 PRIME FACTORIZATIONS AND THE RELATIONS σ, Θ AND τ

In 1992 Feder [4] showed that there exists a unique, finest product relation on the edge set of every connected graph G, which he called σ. Let it correspond to the factorization $G_1 \square \cdots \square G_k$. Clearly no G_i can be the product of two or more graphs on at least two vertices each, otherwise σ would not be the finest product relation. This means that each G_i is *indecomposable* or, as we say, *prime*. This means that $G = G_1 \square \cdots \square G_k$ is a representation of G as a product of prime graphs. Because σ is unique, the prime factorization is also unique. As replacement of a factor by an isomorphic one and a change of the order of the multiplication produces a graph isomorphic of G, one says that prime factorization is unique up to the order and isomorphisms of the factors. This was first shown by Sabidussi [20].

Feder also showed that σ is the transitive closure of the union of two relations Θ and τ, defined as follows. We say two edges $e = xy$ and $f = uv$ are in the relation Θ if $d(x,u) + d(y,v) \neq d(x,v) + d(y,u)$, where $d(x,y)$ denotes the distance between x and y.

Furthermore, e and f are defined to be in the relation τ if they share a common endpoint, and if there is no unique chordless square that contains them both.

Note that opposite edges of a chordless square are in the relation Θ, and that edges e,f that are in a triangle are in the relation Θ and also in τ. By Feder's result $\sigma = (\Theta \cup \tau)^*$, where $(\Theta \cup \tau)^*$ denotes the transitive closure of $(\Theta \cup \tau)$.

Clearly it is possible that all edges in a chordless square have the same color with respect to σ, for example if they are contained in a $K_{2,3}$. But, let us recall that in a nontrivial Cartesian product each edge is in a product square, that product squares have no diagonals, and that incident edges of such a square have different colors with respect to the product coloring induced by the given decomposition. Of course this also holds with respect to the coloring induced by σ, because σ is the finest product coloring.

2.7 THE RELATIONS δ AND AN ALTERNATE RELATION τ

Given an edge e in a nontrivial Cartesian product G, there must be a product square containing e, say $efgh$. Then, for the given product coloring, e, g have the same color, just as f, h, but the colors of f and g

are different. Because σ is the finest product relation, this also holds for the coloring induced by σ.

If we could construct the edge-coloring that is induced by σ on $G-e$, then it would suffice for reconstruction to look for a path fgh that is not in a chordless square, and where f, h have the same color, but not f and g, because the missing edge e would have to connect the origin of the path with its endpoint.

Unfortunately we cannot use Θ_{G-e} for the computation of such a coloring, because the metric of $G-e$ differs too much from that of G. Moreover, we cannot use τ_{G-e} either, because in $G-e$ the pairs of edges f, g and g, h are in τ_{G-e}, and then the path fgh is monochromatic, but in G the edges f, g have different colors. To avoid these difficulties we use relations δ and $\bar{\tau}$, an alternate of τ, that are finer than Θ, respectively τ.

We say two edges e and f are in the relation δ, compare [7], if they are equal or opposite edges of a chordless square. Obviously $\delta \subseteq \Theta$ and the metric of G is not needed for its computation.

Furthermore, we say two edges f,g are in the relation $\bar{\tau}$ if they are equal or if there exist chordless squares

$$fpf'p', gp'g'p'' \text{ such that } (f,g) \in \tau \text{ and } (f',g') \in \tau. \tag{2.1}$$

Clearly δ_{G-e} and $\bar{\tau}_{G-e}$ are finer than $\Theta_G|_{G-e}$, respectively $\tau_G|_{G-e}$.

Because $V(G-e) \subset V(G)$ this means that two edges of $G-e$ that have the same color with respect to δ_{G-e}, have the same color with respect to δ_G, and if they have the same color with respect to $\bar{\tau}_{G-e}$, then also with respect to τ_G. Therefore

$$(\delta_{G-e} \cup \bar{\tau}_{G-e})^* \subset (\delta_G \cup \tau_G)^* \subseteq (\Theta_G \cup \tau_G)^* = \sigma_G. \tag{2.2}$$

Thus neither f,g, nor g,h of the product square $efgh$ will have the same color with respect to $(\delta_{G-e} \cup \bar{\tau}_{G-e})^*$, because $(\delta_{G-e} \cup \bar{\tau}_{G-e})^*$ is finer than σ_G.

It may happen though that several paths pqr of length 3 in $G-e$ have the property that p,r have the same $(\delta_{G-e} \cup \bar{\tau}_{G-e})^*$ color, but not p and q. The reason is that the number of colors (equivalence classes) increases when the relations become finer. We shall later see that the number of equivalence classes of $(\delta_G \cup \tau_G)^*$ is nonetheless bounded by the minimum degree of G, although $(\delta_G \cup \tau_G)^*$ may have more colors than σ.

The next lemma bounds the number of equivalence classes of $(\delta_{G-e} \cup \bar{\tau}_{G-e})^*$.

Lemma 2. *The number of equivalence classes of $(\delta_{G-e} \cup \bar{\tau}_{G-e})^*$ can exceed the number of equivalence classes of $(\delta_G \cup \tau_G)^*$ by at most $3\Delta - 5$.*

Proof. We have to consider the effect of the removal of an edge to δ and τ. If e is in triangle, say efg, then edge pairs $\{e,f\}$, $\{f,g\}$ and

$\{g, e\}$ are in the same color class with respect to τ_G. Hence, if f, g are in different $\bar{\tau}_{G-e}$ classes, then the number of colors increases by 1. As e is in at most $\Delta - 1$ triangles, this adds at most $\Delta - 1$ colors.

If e is in a square without diagonals, say $efgh$, then this square may have one or two colors with respect to $(\delta_G \cup \tau_G)^*$, but up to three colors with respect to $(\delta_{G-e} \cup \bar{\tau}_{G-e})^*$. If the square $efgh$ is not the only square without diagonals containing ef and $efh'g'$ is another chordless square containing ef, then $g'h'gh$ is a square, so h and h' have the same color, just as g and g'. This does not further increase the number of colors. We can thus assume that all squares $ef'g'h'$ share only the edge e, and that the total number of colors the edges of these squares, after removal of e, have in $G - e$ is at most $3\Delta - 3$. Because we consider all squares containing e in G, at least one is a product square, which has two colors. Thus the increase is at most $3\Delta - 5$.

Because an edge f cannot both be in a triangle and in a chordless square with e, the maximum number of colors added is bounded by $3\Delta - 5$. □

As an example, consider the Cartesian product $G = S_d \,\square\, K_2$, where S_d is a star with a central vertex that is incident with d edges. See Figure 2.1 for $d = 4$. Suppose e is the edge connecting the two vertices of degree $d + 1$ in G. G has 2 color classes with respect to $(\delta_G \cup \tau_G)^*$ and $G - e$ has as many color classes as it has edges, that is $3\Delta - 3 = 2 + (3\Delta - 5)$. This shows that the result is tight.

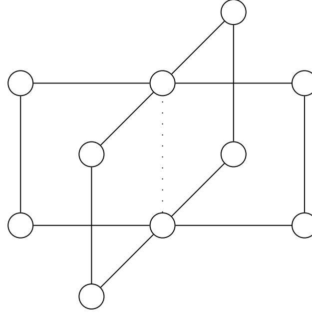


Figure 2.1: $S_4 \,\square\, K_2$

2.8 TWISTED CARTESIAN PRODUCTS

Graphs G on which the relation $\xi = (\delta \cup \tau)^*$ is nontrivial play an important role here. Clearly ξ is nontrivial for any nontrivial Cartesian product, because $\xi \subseteq \sigma$, but it can also be nontrivial for graphs that are prime with respect to the Cartesian product.

Clearly ξ is an equivalence relation and, as in the case of σ, we assign colors to its equivalence classes (resp. the edges in the equivalence classes). This yields a refinement of the coloring with respect to σ. An example of a graph where ξ is a proper refinement of σ is the twisted

ladder depicted in Figure 2.2. It is clearly prime with respect to the Cartesian product.

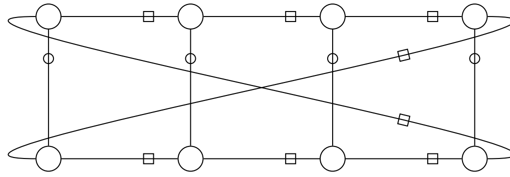


Figure 2.2: Möbius strip

The relation ξ shares many important properties with σ. It was investigated in [14] as the transitive closure of a relation called δ. In order to use the results from that paper, we have to show that our ξ and δ^* of [14] are the same. To avoid confusion, let us use the notation δ_z for the relation δ of [14]. It is defined as follows.

Two edges e and f are said to be in the relation δ_z if one of the following conditions is satisfied:

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

(2) e and f are incident and there is no chordless square spanned by the edges e and f.

(3) $e = f$.

Clearly condition (1) is equivalent to $e \delta f$, and condition (2) stronger than our condition for τ, which forbids only unique chordless squares. Hence $\delta_z \in \delta \cup \tau$. On the other hand, if e and f are in more than one chordless square, then e and f are in the relation δ^*, and hence $\delta_z^* = (\delta \cup \tau)^* = \xi$.

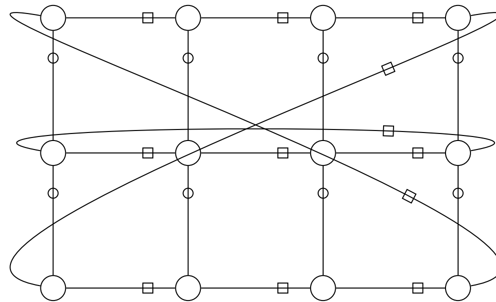


Figure 2.3: Different numbers of vertices in components of a twisted Cartesian product

For us it will be important that δ_z^*, that is, ξ, has the square property and that to any vertex v and any arbitrarily chosen color i, there is an edge e of color i that is incident with v, see [14, Lemma 1]. The latter property means that the number of colors in ξ is bounded by the minimum degree of G, and thus also by Δ.

Because of the similarity with Cartesian products we call graphs, where ξ is nontrivial, *twisted Cartesian products*. Clearly each nontrivial Cartesian product is a twisted Cartesian product.

For convenience we will retain the term *product square* for squares in twisted Cartesian products that are not monochromatic.

One of the properties in which ξ differs from σ is the fact that connected components of the spanning subgraphs who edges have the same color need not have the same number of vertices, compare Figure 2.3, and that these components need not be convex in G. We note in passing that it was shown in [14] that σ is the convex closure of ξ.

Part III

RECONSTRUCTION

3 RECONSTRUCTION

3.1 EDGE-DELETED NONTRIVIAL CARTESIAN PRODUCTS ARE PRIME

Let G be graph and $e \in E(G)$. Recall that the *edge-deleted graph* $G - e$ is defined on the same set of vertices as G and $E(G - e) = E(G) - \{e\}$. We show that edge-deleted nontrivial Cartesian products are prime, and begin with the following lemma.

Lemma 3. *Let G be a nontrivial Cartesian product and $e \in E(G)$. Let $e = ad$ and $abcda$ be a product square in G. If $G - e$ is a Cartesian product, then the path $abcd$ must be monochromatic in any product coloring of $G - e$.*

Proof. Let us assume that the pair of incident edges ab and bc have different colors in $G - e$. This would mean that the edges ab and bc span a product square $abcg$ in $G - e$. This leads to two squares without diagonals spanned over ab and bc in G, contrary to the Unique Square Lemma, see Figure 3.1. (In the figure the □ on the edges ab and cd and the ○ on the edge bc and ad denote product colors in G. Actually this symbolic representation of colors stays the same for all other figures: □ and ○ correspond to colors c_A and c_B, respectively.)

The same argument can be repeated for the pair of edges bc and cd. □

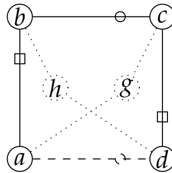


Figure 3.1: Product square $abcd$ in G, with removed edge $e = ad$

Lemma 4. *Let G be a nontrivial Cartesian product and $e \in E(G)$. Then $G - e$ is prime.*

Proof. Let $G = A \,\square\, B$ and let c_A, c_B be the product colors of G. Suppose that e is contained in an A-layer and let $e = ad$, where $abcda$ is a product square of $A \,\square\, B$. This means that ad, bc have color c_A and ab, dc color c_B.

We assume that $G - e$ is not prime, say $G - e = X \,\square\, Y$, with product colors c_X, c_Y, and lead this to a contradiction.

By Lemma 3 all edges of the path $abcd$ in $G - e$ have the same color in any product coloring of $G - e$. We choose the notation such that they are in an X-layer, so their color is c_X.

There must be at least one edge incident to a with color c_Y. Let this edge be aa'. Now we have to consider two different cases regarding the position of aa' in G, namely whether aa' belongs to an A-layer or to a B-layer.

But first we invoke Lemma 3 again to construct edges bb', cc' and dd' that are also colored c_Y, together with edges $a'b'$, $b'c'$ and $c'd'$ that are colored c_X. (Again the introduced colors c_X and c_Y will be represented in all figures by ■ and ●, respectively).

Assume now that aa' belongs to an B-layer with product color c_B; see Figure 3.2. Clearly this implies that aa', bb', cc' and dd' have color c_B. Because ab and $a'b'$ have color c_B, the entire square $aa'b'ba$ has color c_B, which means that it is in a B-layer. Similarly one shows that $cc'd'dc$ also is in a B-layer.

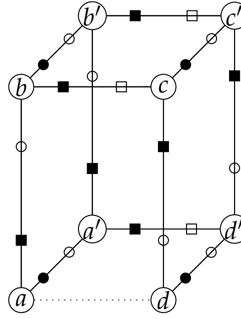


Figure 3.2: $G - e$ prime, aa' hac color c_B.

These B-layers have to be different, because the edge bc has color c_A. Clearly $b'c'$ also has color c_A. As the C_a-colored edges between the layer B^a and B^d induce an isomorphism between these layers, a' must have a neighbor in B^d, say d'' such that $cc'd''dc$ is a square. Because $cc'd'dc$ is a product square in $X \square Y$ this is only possible if $d'' = d'$. But then $a'd'$ is an edge in $X \square Y$. Because the path $a'b'c'd'$ has color c_X, it is also colored c_X and the square $a'b'c'd'a'$ is in $X^{a'}$. As the c_Y-colored edges between $X^{a'}$ and X^a induce an isomorphism, the edge ad must also be in $X \square Y$, contrary to assumption.

Finally, assume that aa' has color c_A; see Figure 3.3. Then aa', bb', cc' and dd' have color c_A. As bc and hence $b'c'$ are also colored c_A. Thus the square $bcc'b'b$ is in the layer A^b, and because the edges ab, $a'b'$, dc and $d'c'$ induce an isomorphism between A^a and A^b, there must be an edge $a'd'$. It is also in $X \square Y$. As before we infer that ad must also be in $X \square Y$, contrary to assumption. □

3.2 RECONSTRUCTION WHEN K_2 IS NOT A FACTOR OF G

In this section we consider reconstruction of nontrivial Cartesian products G from edge-deleted subgraphs for the case when G has

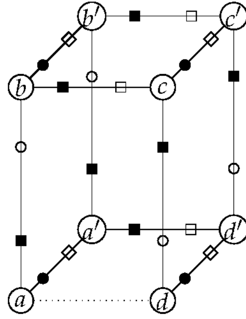


Figure 3.3: $G - e$ prime, aa' has color c_A.

no factor K_2. Contrary to the preceding sections, where we removed edges $e = ad$, we will now add edges $f = ad$.

Lemma 5. *Let G be a nontrivial Cartesian product, $e \in E(G)$, and let $f = ad$ be a pair of vertices not in $E(G)$. If G does not have K_2 as a factor, then $G - e + f$ is prime.*

Proof. Let $G = A \square B$, $e \in E(G)$, and f be a pair of vertices ad not in $E(G - e)$. Suppose $H = G - e + f$ is a nontrivial Cartesian product $X \square Y$. Let the notation be chosen such that f is in an X-layer. Clearly f must be in a product square, say $abcda$, where bc has color c_X, and ab, cd have color c_Y.

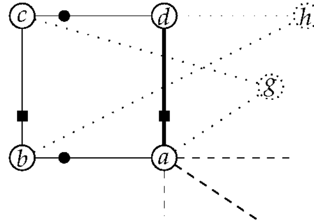


Figure 3.4: Inserted new edge $f = ad$

Suppose the edges ab and bc have different colors in the original graph G. If that were the case, then ab and bc would have to span a product square $abcga$ in G, which would mean that the edges ab and bc would span two different squares in $H = X \square Y$, but this is not possible; compare Figure 3.4. Similarly we argue that there can be no product-square $bcdhb$ in G, hence the path $abcd$ is monochromatic in G. Without loss of generality we can assume that $abcd$ is colored c_A. Note that it is a shortest path because $abcda$ is a product square in H and thus has no diagonals.

Clearly there must be a path $a'b'c'd'$ and edges aa', bb', cc', dd' in G, where $a'b'c'd'$ has color c_A and the other edges have color c_B; compare Figure 3.5. Let R be this subgraph of G. It is the Cartesian product of a path of length 3 by an edge.

Observe that R is an induced subgraph of G. As $abcd$ is a shortest path it is induced. By the isomorphism between the layers A^a and $A^{a'}$ in G the path $a'b'c'd'$ is also induced, and by the definition of Cartesian product the only edges between $abcd$ and $a'b'c'd'$ are aa',bb', cc' and dd'. So it remains to show that neither ad nor $a'd'$ are in G. For $f = ad$ this is so by definition, and if $a'd'$ were in G, then ad would also have to be in G, because of the isomorphism of layers.

We now claim that R must contain e. If not, then R is a subgraph of H. Because $a'b'$ and $c'd'$ have color c_Y, but not $b'c'$, there are product squares $a'b'c'x'a'$ and $b'c'd'y'b$ in H. Neither $x' \neq d'$ nor $y' \neq a'$ can hold, because $a'd' \notin E(G)$.

By convexity x' and y' are in $A^{a'}$. But then, by the isomorphism of layers, we have vertices x, y in A^a and squares $abcxa$ and $bcdyb$. At least one of those squares does not contain e, and is thus in H. It contains two edges that are also in $abcd$, in contradiction to the Unique Square Lemma (applied to squares in H).

Hence, R contains e. Because $abcd$ is in H this leaves the following possibilities for e: $e = aa', dd'$, $e = bb', cc'$, $e = a'b', cc'$, or $e = b'c'$. By the symmetry of R it suffices to treat $e = aa', bb', a'b'$, and $e = b'c'$.

We will show that H is prime in all these cases.

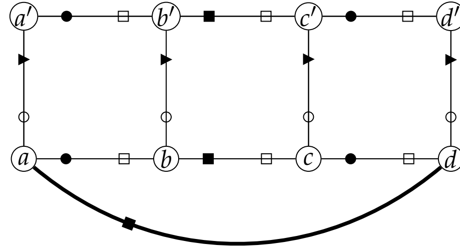


Figure 3.5: The induced subgraph R of G and the edge ad

1. $e = aa'$

   This case is depicted in Figure 3.6. Clearly $b'c'$ and $c'd'$ have different colors in H and there is a product square $b'c'd'y'b'$. Because $a'd'$ is not in $E(G)$ the vertex $y' \neq a'$. By the isomorphism of layers we thus see that there must be a square $bcdyb$ without diagonals in A^a, in contradiction to the uniqueness of the product square $abcda$ in H.

2. $e = a'b'$

   This is depicted in Figure 3.7. We can use exactly the same argument as in the case $e = aa'$.

3. $e = bb'$

   This is depicted in Figure 3.8. Invoking the argument from Lemma 4 again we see that the path $bcc'b'$ has color c_X, whereas

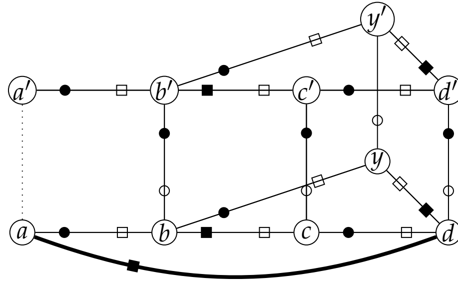


Figure 3.6: Missing edge $e = aa'$

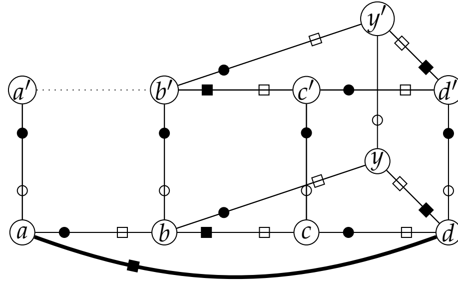


Figure 3.7: Missing edge $e = a'b'$

$c'd'$ has color c_Y. Now we can repeat the argument we used for $e = aa'$ (and $e = a'b'$).

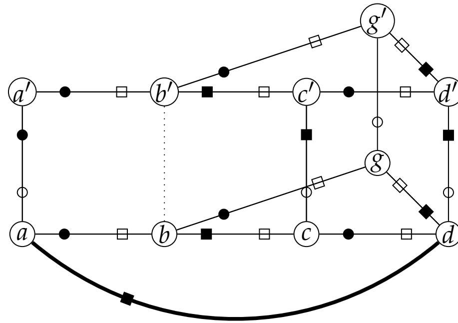


Figure 3.8: Missing edge $e = bb'$

4. $e = b'c'$

   Part of this is depicted in Figure 3.9. We now use the assumption that G has no factor K_2. It implies that $B \neq K_2$. Suppose first that there is an edge aa'' in B^a. Then there are vertices b'', c'', d'' such that the subgraph induced by them and a, b, c, d is isomorphic to R. Let it be R'. By the same arguments which we used when considering R we infer that R' has to contain e, which is not possible.

If there is no such edge aa'', then there must be an edge $a'a''$ in B^a, compare Figure 3.10. By the same arguments as in the proof of Lemma 4 we infer that the paths $b'bcc'$ and $b'b''c''c'$ have to be monochromatic in H, and by convexity their colors have to be the same. If there existed an edge between a'' and d'' in G, there would have to be one between a' and d' (in G) too. But this case we have already excluded. But then we observe that $b''c''$ and $c''d''$ have different colors in H and that there is a product square $b''c''d''y''b''$ in H, which leads to a contradiction as in the cases treated before. □

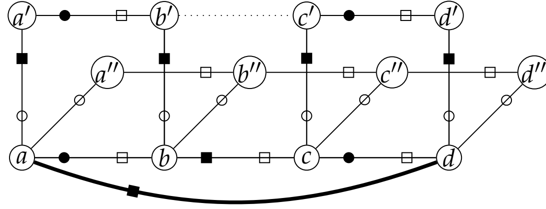


Figure 3.9: Missing edge $e = b'c'$

3.3 RECONSTRUCTION WHEN G CONTAINS A FACTOR K_2

We still have to treat the case when G has a K_2 as a factor, but we begin with the observation that in the proof of Lemma 5 we only needed the assumption that G had no factor K_2 in the case $e = b'c'$, where we assumed that $B \neq K_2$. Hence, we have to investigate the case when $B = K_2$. We begin with a somewhat technical lemma.

Lemma 6. *Suppose $G = A \Box B$ and we have the situation of Item 4 in the proof of Lemma 5, except that $B = K_2$. If the factor Y in the presentation $H = X \Box Y$ is not K_2, then the reconstruction is unique. Otherwise it is unique up to isomorphisms.*

Proof. Let us have a look at Figure 3.11. Recall that $H = X \Box Y$ and that bc has color c_X. Clearly $X \neq K_2$.

If $Y \neq K_2$, then we observe that $G - e = H - f$ and that $G = H - f + e$. We can thus interchange the roles of G and H. Since $H = X \Box Y$, where neither X nor Y are a K_2, we have unique reconstruction by the previous arguments.

Now suppose that B and Y are isomorphic to a K_2. If we remove bc from $G - e$, then we have deleted two edges corresponding edges of the A-layers of G through a and a'. The resulting graph is still a Cartesian product. To be more precise, it is the product of $A - p_A(e)$, that is, A minus the projection $p_A(e)$ of e into A, by K_2.

On the other hand, if we remove f and bc from $H = X \Box Y$ we remove two corresponding edges of the X-layers of H through a and

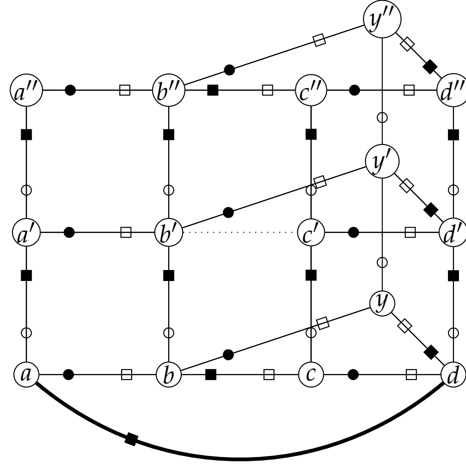


Figure 3.10: Missing edge $e = b'c'$ and no factor K_2

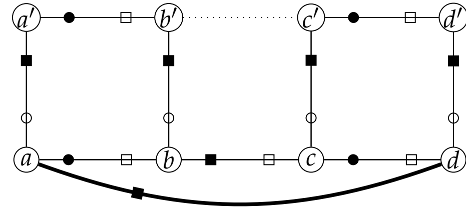


Figure 3.11: Missing edge $e = b'c'$ where $B = K_2$

b. The resulting graph is still a Cartesian product. In this case it is the product of $X - p_X(f)$ times K_2.

Notice that both $p_A(e)$ and $p_X(f)$ are equal to bc. In both cases the resulting graphs are the same, namely $M = G - \{e + bc\}$, but we have two representations as a Cartesian product, namely $M = (A - p_A(e)) \,\square\, B$ and $M = (X - p_X(f)) \,\square\, Y$.

(1) Let us assume first that M is connected. Then its prime factorization is unique up to isomorphisms, and the fact that $B \cong Y \cong K_2$ implies that $A - p_A(e) \cong X - p_X(f)$.

But we can say even more. Both B and Y are prime factors of M, and their sets of layers are different. From the results in Section 2.3 we then infer that B and Y are distinct prime factors of M and

$$M = N \,\square\, B \,\square\, Y,$$

where $A - p_A(e) \cong X - p_X(f) \cong N \,\square\, K_2$; see Figure 3.13.

The case $G - b'c' + ad = (N \,\square\, B + bc) \,\square\, Y$ is depicted in Figure 3.12.

Recall that the edge bc connects vertices of an $(A - p_A(e))$-layer of M and also vertices in an $(X - p_X(f))$-layer of H. In the first case we consider B^b and B^c and, by adding e, join the intersection of these layers with $(A - p_A(e))^{a'}$. In the second case we consider Y^b and Y^c, and by adding f, join the intersection of these layers with $(X - p_X(f))^a$.

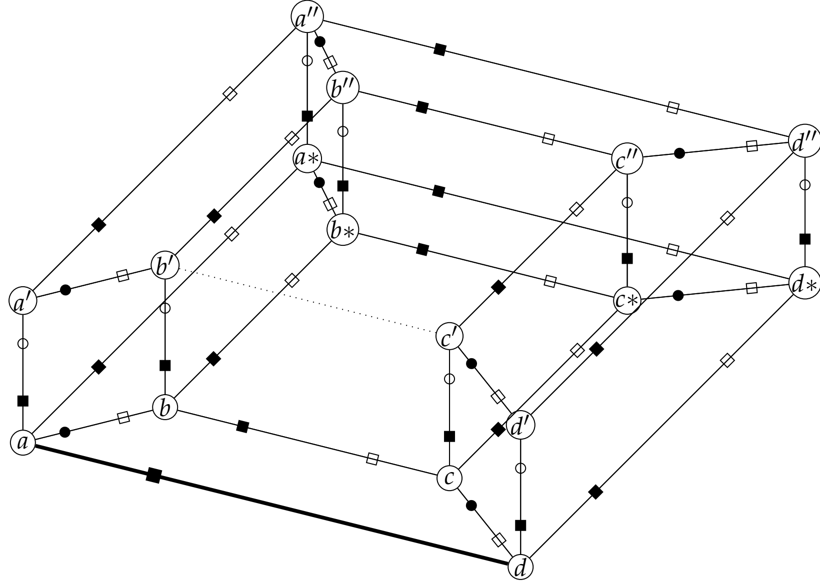


Figure 3.12: $G - b'c' + ad$, compare Fig. 3.11 for the subgraph spanned by $\{a, b, c, d, a', b', c, d'\}$

We show now that the resulting graphs are isomorphic. Both B and Y are factors of M. Let $M = B \square Y \square Z$, so we use three coordinates for M. If $V(B) = V(Y) = \{0, 1\}$, then each vertex v of M has the coordinates (v_1, v_2, v_z), where $v_1, v_2 \in \{0, 1\}$ and $z \in V(Z)$.

It is easy to see that the mappings $(v_1, v_2, v_z) \mapsto (v_1 + 1, v_2, v_z)$, $(v_1, v_2, v_z) \mapsto (v_1, v_2 + 1, v_z)$, or $(v_1, v_2, v_z) \mapsto (v_2, v_1, v_z)$, additions modulo 2, are automorphisms.

Let $b = (0, 0, z_b)$ and $c = (0, 0, z_c)$, where $z_b z_c \in E(Z)$. Then $b' = (1, 0, z_b)$, and $c' = (1, 0, z_c)$. Furthermore, $a = (0, 1, z_b)$, $d = (0, 1, z_c)$, and $a' = (0, 0, z_b)$.

Thus, the automorphism $(v_1, v_2, v_z) \mapsto (v_2, v_1, v_z)$ of M fixes bc and interchanges $b'c'$ with ad. Recall that bc, $b'c'$ are in G, that bc, ad in H, and that G, H and M have the same sets of vertices. Thus G and H are isomorphic.

(2) If M is not connected it has two connected components, each component has unique prime factorization. We can use the same arguments as before, but can choose the layers of cd and ba independently of each other, which yields more possibilities for the reconstructed graph.

For example, N could consist of two isolated vertices, and G of bc together with the squares $aa'b'ba$ and $dcc'd'd$. For f we could then take $ad, ad', a'd, a'd', b'd$ or $c'a$.

Nonetheless, all reconstructed graphs are isomorphic.

We wish to remark that our arguments also hold for finite and infinite connected graphs G that are nontrivial Cartesian or weak Cartesian products.

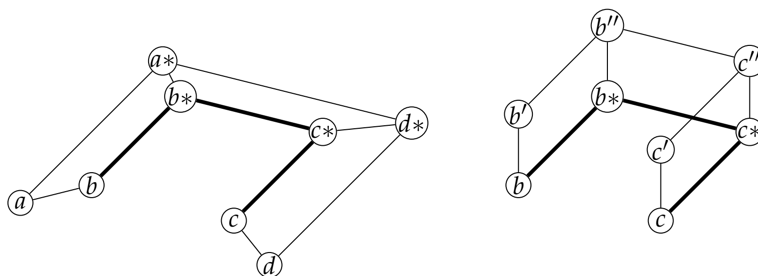


Figure 3.13: $A - bc = N \,\square\, Y$ and $X - bc = N \,\square\, B$, edges of N in bold

3.4 CONCLUSION

We formulate our result as a theorem.

Theorem 7. *Let G be a connected, nontrivial Cartesian product. Then G can be uniquely reconstructed up to isomorphisms from any edge-deleted subgraph $H = G - e$, where $e \in E(G)$.*

Furthermore, in H the endpoints of the deleted edge e are uniquely determined, unless G has a representation $G = A \,\square\, K_2$, where e is in an A-layer, and where G has at least one other factor K_2. In that case one can characterize all possibilities for the insertion of an edge f into H such that $H + f$ is a Cartesian product, and all reconstructions are isomorphic.

Part IV

ALGORITHM

4 ALGORITHM

4.1 Reconstruction in $O(mn^2)$ time

Given a graph $G - e$ one can try all possible extensions by an edge f and check whether they yield a Cartesian product. If the order of G is n, there are $O(n^2)$ possibilities for f. Because prime factorization can be done in linear time and space in the size m of G by [10], the reconstruction is possible in $O(mn^2)$ time and space.

Within the same time and space complexities one can also determine all possible reconstructions.

4.2 Reconstruction in $O(m\Delta^2)$ time

Whenever we insert an edge in $G - e$ to test whether the new graph is a Cartesian product we invoke the algorithm of [10], which determines the prime factors of a graph with m edges in $O(m)$ time.

4.2.1 *Preprocessing and computation of necessary relations*

We clearly have to compute the relations δ, τ, $\bar{\tau}$ and transitive closures of various unions of them. For the transitive closures we simply note that the complexity of computing the transitive closure of a relation ρ is $O(|\rho|)$, where $|\rho|$ denotes the number of pairs in the relation ρ.

Let us focus on δ_G now. For its computation we need the squares of G. There is an algorithm of Chiba and Nishizeki [1] that computes to certain pairs of vertices v_1, v_2 a list $\{v_3, v_4, \ldots\}$, such that each square containing v_1 and v_2 is of the form $v_1 v_i v_2 v_j$, where v_i, v_j are from the list, and where the computed triples

$$\{v_1, v_2, \{v_3, v_4, \ldots\}\}$$

encode all squares of G. The total size of the list is $m\,a(G)$, where $a(G)$ is the arboricity of G, and the list can be computed in $O(m\,a(G))$ time. For us it is important to note that $a(G)$ is bounded by the maximal degree of $\Delta(G)$ of G.

It is easy to see that all edges between any two of the vertices in such a triple $\{v_1, v_2, \{v_3, v_4, \ldots\}\}$ have the same color with respect to δ^* unless the sublist $\{v_3, v_4, \ldots\}$ contains exactly two vertices that are not adjacent. For this case, we note that $v_1 v_i$ and $v_j v_2$ are in the relation δ, and also $v_1 v_j$ and $v_i v_2$. These chordless squares $v_1 v_i v_2 v_j v_1$ are the candidates for product squares in ξ, and thus also candidates for product squares in σ.

Note that δ^* and the list of candidates for product squares can be computed in $O(m\Delta)$ time, and that the list contains at most $m\Delta$ elements. We can also store these squares in such a way that one can check in constant time, whether a pair e,f of incident edges is in a unique chordless square.

Now to the computation of τ. We consider all edges e, and for every incident edge f we check whether e and f are in a chordless square. This can be done in $O(m\Delta)$ time, and the length of the list of pairs of edges that are in the relation τ is at most $m\Delta$.

For the computation of $\bar{\tau}$ we can proceed as follows. For each pair of edges f,g in τ we consider all edges p' incident with the common vertex of f and g. Then we look for chordless squares $fpf'p'$ and $gp'g'p''$, and then for a chordless square $f'g''f''g'$. If we find such a configuration, then the pair f,g is not in $\bar{\tau}$, otherwise it is.

There are $m\Delta^2$ cases to check, and each check takes constant time. Hence $\bar{\tau}$ can be computed in $O(m\Delta^2)$ time, and contains at most $m\Delta$ elements.

Hence, preprocessing takes at most $O(m\Delta^2)$ time.

4.2.2 *$G - e$ contains edges that are in no chordless squares*

The length of the list of chordless squares is at most $m\Delta$. Hence, if there are edges that are in no chordless square, we can find one in $O(m\Delta)$ time. Note that it is possible that no edge of $G - e$ is in a chordless square, see Figure 2.1.

Suppose uv is an edge of $G - e$ that is in no chordless square. Clearly uv is in a product square in G. This square must contain e, otherwise uv is in a chordless square without diagonal in $G - e$. There are two possibilities for such a product square: either e is incident with uv, or opposite to uv.

In the first case the product square in G is of the type $uvywu$ or $uvwyu$, see Figure 4.1. For the type $uvwyu$, y is the other endpoint of e. It has distance 2 from v. Thus there are at most Δ^2 possibilities for y, that is, for the insertion of e. The same holds for the other type.

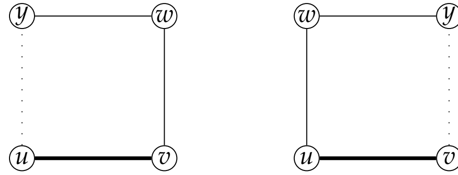


Figure 4.1: e incident to uv in $G - e$

In the second case, where e is opposite to uv, one endpoint of e is adjacent to u, and the other to v, see Figure 4.2. Again, this yields at most Δ^2 possibilities.

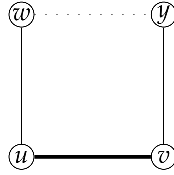


Figure 4.2: e opposite to uv in $G - e$

If uv is known, this means that e can be found in $O(m\Delta^2)$ time, because one can check in $O(m)$ time whether a single insertion of an edge yields a Cartesian product. Hence the overall complexity of this part is $O(m\Delta^2)$.

4.2.3 *Each edge of $G - e$ is in a chordless square*

We need the following two lemmas.

Lemma 8. *Let G be a nontrivial Cartesian product, and $efgh$ a product square of G. If each edge of $G - e$ is in a chordless square, then f, h are in relation $\bar{\bar{\xi}}$ in $G - e$, but not f, g.*

Proof. Let us assume first that G has at least three factors. Given a product square $efgh$, where e, f, g, h are, respectively, ad, ab, bc, cd, there must be an edge aa' of G that is incident with ad and ab, and whose color is different from that of ad and ab. The subproduct of ad,ab and aa' in G is a cube. It is easy to see (via the relation δ) that, even after removal of ad from the cube, $f = ab$ and $h = cd$ have the same color, see Figure 4.3.

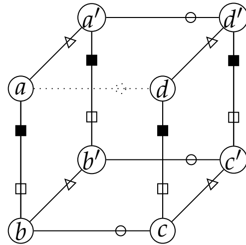


Figure 4.3: Lemma 8, G has at least three factors

This leaves the case when G has only two factors, say $G = A \square B$. By assumption there is a chordless square $abb'a'a$ in $G - e$. It is also a chordless square in G, hence bb' must be different from bc by the square property on G. Because G has only two factors the color of aa' (in G) is either the color of ab or of bc.

Suppose ab and aa' have the same color, see Figure 4.4. Then $abb'a'a$ is monochromatic. We can choose the notation such that it is in an A-layer. The edges ad and bc have the same color and connect the A-layer containing $abcda$ with the A-layer containing cd. Because the edges between adjacent layers induce an isomorphism, there must

be edges $a'd'$ and $b'c'$ such that the edges $ad, bc, b'c', a'd'$ induce an isomorphism between $abb'a'a$ and a chordless square $dcc'd'd$. These two squares, together with the edges $ad, bc, b'c', a'd'$ are a cube. If we remove $e = ad$ from it, then one sees as in the previous case, that $f = ab$ and $h = cd$ still have the same color.

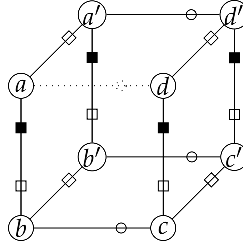


Figure 4.4: Lemma 8, the edges ab and aa' have the same color

Now suppose ab and aa' have different colors. There must also be a chordless square containing bc and one containing cd, say $bxycb$ and $cdd'c'c$. If bc and bx have the same color, or if cd and cc' have the same color, then the same arguments as before show that $f = ab$ and $h = cd$ have the same color.

Hence we can assume that $\{ab, aa'\}$, $\{bc, bx\}$ and $\{cd, cc'\}$ are pairs of edges of different colors (in G). But then we have product squares to $\{bb', bx\}$ and to $\{cy, cc'\}$, say $bxwb'b$ and $cyzc'c$, yielding the configuration depicted in Figure 4.5. It is a Cartesian product of a path of length 3 by one of length 2, the "top middle edge" being e.

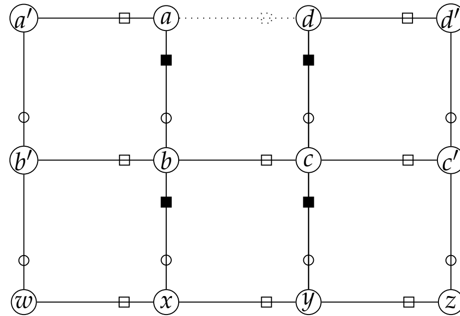


Figure 4.5: Lemma 8, $f = ab$ and $p = aa'$ of different colors – part 1

If $\{ab, bx\}$ are in a square without diagonal in G, say $abxx'a$, see Figure 4.6, then this is also the case for $\{dc, cy\}$, let it be $dcyy'd$, and there is a chordless square $xx'y'yx$. These squares are also in $G - e$ and ensure that the colors of xx', yy' and $f = ab$, $h = cd$ are the same.

If the pair $\{ab, bx\}$ is not in a square without diagonal, see Figure 4.5 again, then $ab\tau_{G-e}bx$, but also $a'b'\tau_{G-e}b'w$. Hence $ab\bar{\tau}_{G-e}bx$. By a similar argument $cd\bar{\tau}_{G-e}cy$. Thus $\{ab, bx\}$ have the same $\bar{\xi}_{G-e}$ and also the pair $\{cd, cy\}$. Because $\{bx, cy\}$ is a pair of opposite edges in a chordless square, bx and cy have the same color, and thus also $f = ab$ and $h = cd$. □

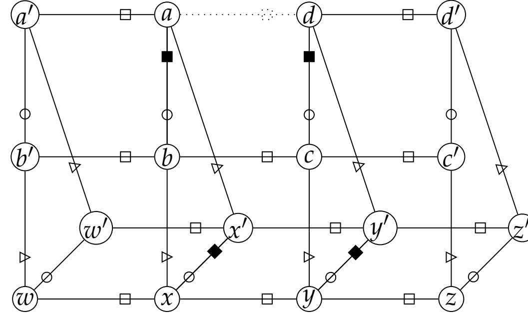


Figure 4.6: Lemma 8, $f = ab$ and $p = aa'$ of different colors – part 2

Lemma 9. *Let G be a nontrivial Cartesian product and $e \in E(G)$. Suppose fgh is a path that is not in a chordless square in $G - e$, and where the ξ_{G-e} colors of f and g are different, but those of f and h the same. If the completion of fgh by an edge e' to a square does not yield a Cartesian product, that is if $G - e + e'$ is not a Cartesian product, then f and g belong to the same color class with respect to σ_G.*

Proof. Let G satisfy the conditions of the Lemma. If f and g belong to different color classes with respect to σ_G, then they have to be part of a product square in G. This product square also has to contain h, because f and h have the same color. Let this square be $fghe'$. If f, g do not belong to a square in $G - e$, then e' must be the removed edge e. But then the addition of an edge e' that completes fgh to a square to $G - e$ must yield a Cartesian product. □

Theorem 10. *Let $G - e$ be an edge deleted subgraph of a nontrivial Cartesian product. Then G can be reconstructed in $O(m\Delta^2)$ time.*

Proof. In Section 4.2.2 we have already treated the case when $G - e$ has an edge that is in no chordless square. So we can assume that all edges of $G - e$ are in chordless squares.

By Lemma 8 the missing edge e has to be inserted as the fourth edge of a path fgh that completes fgh to a chordless square. The path fgh is characterized by the properties that the $\bar{\xi}_{G-e}$ colors of f and g are different, and those of f and h the same.

Clearly f, g are in τ_{G-e}, but not in $\bar{\tau}_{G-e}$. Because both τ_{G-e} and $\bar{\tau}_{G-e}$ have at most $m\Delta$ elements, we can find all such pairs f, g in $O(m\Delta)$ time. Now we check for all of the at most Δ edges h that form a path fgh of length 3 with fg whether they have the same color as f. There can be at most one such edge because $\bar{\tau}_{G-e}$ refines σ_G, and in G this is true because of the square property.

Hence in $O(m\Delta^2)$ time we can construct the list of all such configurations. Note that the length of the list is at most $m\Delta$.

Now we parse the list. We complete fgh to a square by insertion of an edge e' and check, whether $G - e + e'$ is a nontrivial Cartesian product. If this is not the case, the colors of f and g are refinements

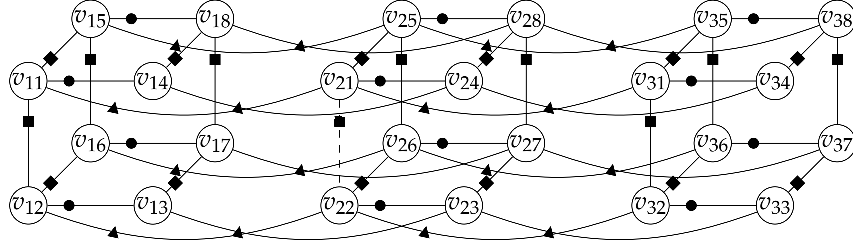


Figure 4.7: Product of a star S_2 by a cube from which an edge was deleted.

of one product color, that is, of a σ_G-color. We call such a triplet *misleading*, merge the colors and go to the next configuration, where fgh are already checked with respect to the merged colors. By [14] the number of colors of ξ_G is at most Δ, and by Lemma 2 the number of colors in $\bar{\bar{\xi}}_{G-e}$ can exceed this by at most $3\Delta - 5$, so we cannot delete more that $4\Delta - 5$ colors. Hence, although parsing the list may take $O(m\Delta)$ time, we only check at most $4\Delta - 5$ times whether an inserted edge yields a product. So this part takes $O(m\Delta) + O(m\Delta^2)$ time. Together with the construction of the list we end up with the time complexity $O(m\Delta^2)$. □

Figure 4.7 shows an example of a graph $G - e$ which is colored with respect to $\bar{\tau}_{G-e}$. It contains misleading triples, like $\{v_{14}v_{11}, v_{11}v_{12}, v_{12}v_{13}\}$ or $\{v_{34}v_{31}, v_{31}v_{32}, v_{32}v_{33}\}$, but also the correct ones, namely $\{v_{21}v_{11}, v_{11}v_{12}, v_{12}v_{22}\}$ and $\{v_{21}v_{31}, v_{31}v_{32}, v_{32}v_{22}\}$. G is the product of a of a star S_2 by an edge deleted cube, and the misleading triples come from the second factor.

Part V

ALGORITHM - DETAILED DESCRIPTION

5 ALGORITHM - DETAILED DESCRIPTION

The algorithm described in the section below allows the edge-reconstruction of a Cartesian product of at least two nontrivial graphs from which a single edge has been removed.

The algorithm handles a special case of a single edge deleted subgraphs having no remaining squares separately, and introduces a general approach otherwise.

Algorithm 1 Algorithm overview

```
Require: G − e
  storeAllSquares( )
  if ¬edgesWithoutSingleSquare.empty() then
      edgeWithoutSquareReconstruction( )
  else
      generalReconstruction( )
  end if
```

5.1 STORE ALL SQUARES

The step of storing all squares existing in graph $G - e$ allows us to accelerate plenty of different checks which take place at the later stages of the main algorithm. The way we collect all squares is pretty straightforward and not the fastest one, but by far the clearest one. As it does not influence the overall complexity of our algorithm we stick to it for the sake of simplicity.

Algorithm 2 Store All Squares

```
Require: storedSquares, edgesWithoutSingleSquare
  procedure storeAllSquares
      for all [u, v] ∈ E(G − e) do
          for all [u, w] ∈ E(G − e) do
              for all y adjacent to v and w do
                  storedSquares.add(uvwy)
              end for
          end for
          if storedSquares.get(uv).empty() then
              edgesWithoutSingleSquare.add(uv)
          end if
      end for
  end procedure
```

To be able to access all squares of a given edges pair in constant time, we need to store the squares in a three-dimensional array where each following dimension corresponds to the next of three vertices incident to edges e and f, that is u, v, w.

The pair of edges we use to find squares may span one square as well as many squares, thus each entry in the array is prepared for it and contains a list to collect all of them.

The edges of the graph are bidirectional, and to store a complete image of all squares into an array we have to assume, for the runtime of the procedure, that edge $uv \neq vu$.

This procedure fulfills also another purpose, which is storing all edges that don't belong to any square. As soon as we encounter even a single edge not belonging to a square, we can skip the general reconstruction procedure in the favor of a straightforward, but checking many possibilities one, which reconstructs a square for the found edge.

5.1.1 *Correctness*

It should be rather clear, that as this routine takes all pairs of incident edges and looks among plausible vertices for those, which are incident to both endpoints, it traverses and stores all existing squares of the input graph.

It should be also clear, that if in this way no square has been found for an edge uv, this edge has no square at all.

5.1.2 *Complexity*

The first loop iterates over all edges of the graph and takes each edge twice, each time changing the order of vertices of the edge. This gives us $2m$ possible pairs of edge endpoints. For each pair of endpoints, we iterate over the neighborhood of one of them, let's say v, and as a vertex of the highest degree has no more neighbors than Δ, the number of iterations is also limited to Δ. This way we can find our u, v and w. Now to find y we can iterate again over the neighborhood of another vertex incident to u, this time vertex w. There can be no more neighbors of w than Δ and the check whether y is adjacent to u can be done in constant time using the adjacency matrix. The additional check for the existence of squares for uv is also done in constant time and so is also the storing of the edge in edgesWithoutSingleSquare collection. Summing it up the time complexity looks as follows: $O(2m\Delta\Delta C + C) = O(m\Delta^2)$.

The calculation of the space complexity needs a bit closer attention because the mentioned three-dimensional array won't be filled completely and so we can reduce the complexity. The first two dimensions of the array consume n^2 space, but only $2m$ of the cells are going

to receive entry, and only for them, the third dimension will exist. This third dimension needs to be a vector of the length of graph's order – n, but also in this case we are not going to fill the complete vector with data, but rather than that, we are going to use Δ cells out of it, as v cannot have more neighbors. This way we achieved the list of possible squares for the pair of edges e and f. The length of this list is also limited by Δ, as there cannot be more squares than w has neighbors. The final missing piece of the space complexity for the collection storedSquares is the size of a single square, which is constant. Wrapping it all up $O(n^2 + m(n + \Delta\Delta C) = O(m(n + \Delta^2)$.

The space complexity of the other collection used in this procedure, edgesWithoutSingleSquare, is smaller and equal to $O(m)$, as there are only m edges in the graph.

It is worth to mention that there exists an algorithm for finding all squares in $O(ma(G))$ introduced by Chiba and Nishizeki. For more details, see [7].

5.2 EDGE WITHOUT SQUARE RECONSTRUCTION

We have already presented a brief yet complete description for this part of the algorithm in Section 4.2.2, so in this section, we are just going to extend it by some figures and the pseudocode notation of it.

Figure 4.1 and 4.2 shows three possible colocations between the found edge without square and the edge to be reconstructed. Fgure 4.2 presenting the case for the first part of the procedure and Figure 4.1 for the rest of it.

It is also worth to mention, that this part of the algorithm inherently solves the special case when the whole graph $G - e$ has no squares at all. This scenario is only possible if G was a product of only two factors, and these factors were either $K_2 \square K_2$ or $K_2 \square S_d$, where graph S_d is a star with d edges incident to the vertex in the center. Figure 5.1 depicts both of these cases.

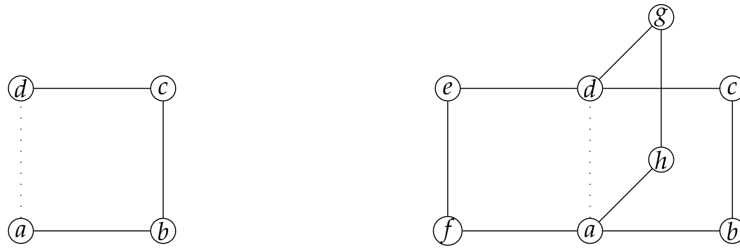


Figure 5.1: Special cases having no squares

Algorithm 3 Reconstruction for edge without square

```
Require: edgesWithoutSingleSquare
  procedure EDGEWITHOUTSQUARERECONSTRUCTION
    uv ← edgesWithoutSingleSquare.getAny()
    for all w ∈ N_{G−e}(u) do
      for all y ∈ N_{G−e}(v) do
        if ISMISSINGEDGEFOUND(wy) then
          return wy
        end if
      end for
    end for

    for all w ∈ N_{G−e}(u) do
      for all y ∈ N_{G−e}(w) do
        if ISMISSINGEDGEFOUND(vy) then
          return vy
        end if
      end for
    end for
    for all w ∈ N_{G−e}(v) do
      for all y ∈ N_{G−e}(w) do
        if ISMISSINGEDGEFOUND(uy) then
          return uy
        end if
      end for
    end for
  end procedure

  procedure ISMISSINGEDGEFOUND(f)
    reconstructedG ← reconstruct(f)
    notPrime ← factorize(reconstructedG)
    return notPrime
  end procedure
```

5.2.1 *Correctness*

Each edge of the original Cartesian product G need to belong to a product square. Having found an edge without a square in $G - e$ and trying all possible ways of reconstructing a square for it must bring us finally to a correct solution. We know, that we have reconstructed a proper square when the factorization procedure returns at least two factors for the given graph $G - e + f$.

5.2.2 *Complexity*

Each for-loop can iterate maximally over Δ vertices, reconstruction of a single edge is done in constant time, and the factorization need $O(m)$, so the overall complexity is $O(m\Delta^2)$.

5.3 GENERAL RECONSTRUCTION

This section describes the general case of reconstruction, which is going to be started if no edge not belonging to a square in $G - e$ has been found.

Algorithm 4 General Reconstruction

```
procedure GENERALRECONSTRUCTION
    COLOREDGES( )
    MERGECOLORSBYTAU( )
    CLEANMISSINGSQUAREEDGEPAIRS( )
    FINDEDGETORECONSTRUCT( )
end procedure
```

Briefly, in the first step our algorithm colors all edges in the graph, merges colors belonging to the same factors, and collects all pairs of edges that are not being a part of a square.

The second step iterates over edges and merges colors using the properties of the relation $\bar{\tau}$.

The third step reduces the number of edges to be analyzed for reconstruction, by removing from the list of edges not being a part of a square all edges which are of the same color.

The fourth step analyzes the remaining pairs of edges not being a part of any square and based on them, it picks out vertices, which are the endpoints of e in G.

5.4 COLOR EDGES

To color the whole graph, we start by coloring any arbitrary square and then, we spread the coloring using the properties of relations δ

and $\bar{\tau}$. For every case where the already existing colors are not enough, we introduce a new color.

One can see, that the actual application of the relations δ and $\bar{\tau}$ happens in the subroutine from Algorithm 7, whereas Algorithm 5 only delivers to it bundles of squares based on the same two edges.

The pairs of edges which are not a part of any square are stored in an auxiliary structure. This structure contains a list of all such edges as well as a three-dimensional set of vectors for instant access to all of them.

Algorithm 5 Color edges

```
Require: storedSquares, adjacencyMatrix
  procedure COLOREDGES
      missingSquareEdges ← new Structure()
      upcomingVertices ← new Queue()

      uvwy ← storedSquares.getAny()
      uv.color ← 1, wy.color ← 1
      uw.color ← 2, vy.color ← 2
      upcomingVertices.push(u)

      while ¬upcomingVertices.empty() do
          u ← upcomingVertices.poll()
          edges ← ORDEREDGES(u)

          for all pair of edges uv, uw incident to u do
              if uv.color = uw.color then
                  Continue
              end if

              squares ← storedSquares.get(uv, uw)
              if squares.empty() ∧ adjacencyMatrix[v][w].empty() then
                  missingSquareEdges.add(uv, uw)
                  Continue
              end if

              COLORSQUARES(squares)

              upcomingVertices.push(v)
              upcomingVertices.push(w)
          end for
      end while
  end procedure
```

In Algorithm 5 we start by choosing an arbitrary square from an already stored list of squares and coloring its opposite edges using the

Algorithm 6 Order edges

```
procedure orderEdges(Vertex u)
    coloredEdges ← new List()
    uncoloredEdges ← new List()
    for all edges uv of u do
        if ¬uv.color.empty() then
            coloredEdges.add(uv)
        else
            uncoloredEdges.add(uv)
        end if
    end for
    edges ← coloredEdges.appendAll(uncoloredEdges)
    return edges
end procedure
```

same colors, as we need any starting point. Then we start an iteration over all vertices where the vertex u from the chosen square is our starting point, and each vertex queued for processing is a neighbor of the one which is currently selected. The vertices to be processed are pushed into a queue, which does not allow any duplicates.

The processing of every single vertex starts from ordering edges from colored to uncolored ones by Algorithm 6. The procedure partitions all its edges into two groups, colored and uncolored ones, and after that, all edges are put back again into a single list, but this time all colored edges are at the beginning of the list. Putting colored edges in the front of the list, and starting coloring from them allows as to spread existing coloring better. Out of this list, we select edges pairwise, and if they are of different colors, we look for squares spanned over them. If one or more squares have been found, we color all of them calling Algorithm 7.

Our input graph $G - e$ contains also incident edges that don't span any square. In G, it would be a sign that these two edges should belong to the same σ_G class, so we could merge their colors. However, during the reconstruction, we have to use coloring induced by relation $\bar{\xi}_{G-e}$ as these two edges could form a square in G and only because of the deletion of e there is no square spanned on them any longer. See Figure 5.2, where edges ab, bc are not spanning a square because of the missing edge ad. That is why we store such edges into the *missingSquareEdges* collection for further analysis, unless they already span a triangle, as we are not interested in reconstructing squares with diagonals.

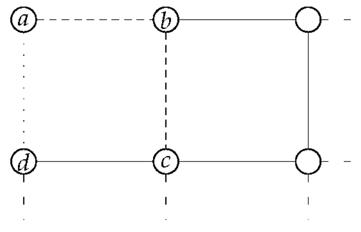


Figure 5.2: Missing edge $e = ad$

5.4.1 *Correctness*

The procedure described in this section does not include the crucial part of the coloring of squares, namely the actual coloring itself. As the coloring of all square has been moved to a subroutine, it is enough for the correctness of this routine, when it considers all pairs of incident edges in the graph, what it does.

5.4.2 *Complexity*

The time complexity can be calculated in a very similar way to the time complexity of the procedure to store all squares of the graph. We iterate over each edge of the graph, and for each one of them we iterate again, but this time over edges incident to the already selected edge. Using the structure of the stored square, one can find all squares containing given two edges in constant time. The last factor of this multiplication is the complexity of coloring all squares containing any two edges and, as we will show it in the next section, it is $O(\Delta)$ hence the time complexity for this procedure is equal to $O(m\Delta^2)$.

Space complexity is defined by the most complex structure in the routine, and it is the three-dimensional set of vectors allowing access to any pair of edges not building a square, in constant time. Just like by storing the squares we have known, that out of n^2 cells of first two dimensions we use only m of them, we know it now as well, and each such cell contains a vector to the endpoints of the second edges plus the list of them. Additionally, we keep each cell in a plain list, as it doesn't impact the complexity and becomes handy by iteration over each entry. The space complexity looks then as follows: $O(n^2 + m(n + \Delta + \Delta)) = O(mn)$.

5.5 COLOR SQUARES

The procedure described in the following section plays a crucial role in the whole algorithm. It colors the edges of each square using the relation δ supported by the relation $\bar{\tau}$.

Algorithm 7 Color Squares

```
Require: squareOppositeEdges, colorsCounter, adjacencyMatrix
  procedure colorSquares(squares)
      allSquaresColored ← true
      for all uvwy ∈ squares do
          if uv and uw two incident colored edges then
              wy.color ← uv.color
              vy.color ← uw.color
          else if uv colored and neither uw nor vy colored then
              if wy.color = ∅ then
                  wy.color ← uv.color
              end if
              otherColor ← findOtherColor(uvwy)
              uw.color ← otherColor
              vy.color ← otherColor
          end if
          if ¬adjacencyMatrix[u][y].empty()
            ∨ ¬adjacencyMatrix[v][w].empty() then
              uy.color ← uv.color
              vw.color ← uv.color
              mergeColors(uv, uw)
              continue
          end if
          if squares.size() > 1 then
              mergeColors(uv, uw)
              continue
          end if
          squareOppositeEdges.store(uv, wy)
          squareOppositeEdges.store(uw, vy)
      end for
  end procedure
```

The essential relation for this procedure, relation δ, colors opposite edges of a square. If the given square has already two incident edges colored, we just copy the already existing colors to the opposite, not yet colored edges.

In another case, when the square has only one edge colored, we look for a color of the second edge applying the relation $\bar{\tau}$, which is done in the subroutine *findOtherColor*.

After the first two steps, all edges of the square should have been colored, so we apply the following two Cartesian product properties to merge some of them:

- Product square cannot have diagonals
- Two incident edges cannot be a part of more than one product square

That is why if this square has any diagonals or there are more squares than one, we merge the colors.

There is also a possibility to use the relation $\bar{\tau}$ again and merge even more colors in this step, but to make the algorithm more coherent and readable, this part has been moved to a separate routine, that is going to be described later.

In the last two lines of the for-loop, we use a structure *squareOppositeEdges*, which is described below, to store square opposite edges for quicker access, but only if everything went well and we deal with a square having exactly two colors.

The structure *squareOppositeEdges* must be introduced to achieve the desired time complexity and it is going to store all pairs of opposite edges of each square and arrange them in the following way: firstly, by each of the opposite edges, creating two entries, one for each opposite edge, and secondly, by the color of the other two edges in the square. Given an edge and a color of the other two edges of the square, the structure should be capable of returning the list of all suitable opposite edges in constant time, and that is why we need to use again a three-dimensional set of vectors.

The procedure *findOtherColor* is the first place, where we apply the relation $\bar{\tau}$ to our algorithm. For this purpose we rename inside of the procedure the given square $uvwy$ to $aa'b'b$, ab, $a'b'$ remaining the uncolored edges, and using the aforementioned relation $\bar{\tau}$ we try to find a colored square $bb'c'c$ – Figure 5.3, so that the color of the edges bc, $b'c'$ could be applied also to the edges ab, $a'b'$. If the procedure fails to find such a square, we have to introduce a new color into the coloring.

Having mentioned the relation $\bar{\tau}$, let us take a closer look at it. Consider the square $abcd$ from Figure 5.2 which could have been a product square in G. The original relation τ would assign edges ab, bc, and cd to one equivalence class, which results in a wrong coloring.

Algorithm 8 Find Other Color

```
Require: squareOppositeEdges, storedSquares, colorsCounter
  procedure findOtherColor(aa'bb')
    otherColor ← ∅
    for all cc' ∈ squareOppositeEdges.get(bb') do
      if cc' = aa' then
        continue
      end if
      abcd ← storedSquares.get(ba, bc)
      a'b'c'd' ← storedSquares.get(b'a', b'c')
      if abcd.empty() ∧ a'b'c'd'.empty() then
        otherColor ← bc.color
      end if

      if otherColor ≠ ∅ then
        break
      end if
    end for

    if otherColor = ∅ then
      colorsCounter ← colorsCounter+1
      otherColor ← colorsCounter
    end if
    return otherColor
  end procedure
```

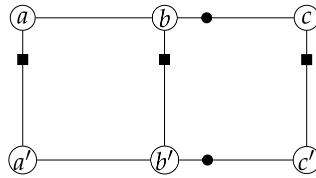


Figure 5.3: Looking for square $bb'cc'$

This example shows why we need to introduce the relation $\bar{\tau}$ instead of applying the relation τ.

Let us consider a pair of edges ab, bc being in the relation τ in $G - e$. If these edges are also in the relation τ in G, then there must be an edge from each vertex a, b and c of different color than edges ab, bc. These new edges should lead to $a'b'$, $b'c'$, parallel to the edges ab, bc and spanning squares $aa'b'b$ and $bb'c'c$. Now if neither the edges $a'b'$, $b'c'$, nor the edges ab, bc don't span a square, it's clear that the edges ab, bc are indeed in the relation $\bar{\tau}$ in G.

5.5.1 *Correctness*

The procedure uses the properties of product square, and also of the relations δ and $\bar{\tau}$, which have been already explained and which correctness has been already proved.

5.5.2 *Complexity*

Two incident edges can have maximally Δ squares, so we have to run the loop at most Δ times. We can check the coloring of all edges in a single square or color two edges based on the other two in constant time, but checking the relation $\bar{\tau}$ needs some more effort.

We can access the list of all square-opposite edges to the colored edge in constant time, and the list can have maximally the length of Δ. For each entry, we have to look for squares spaned on two edges, and because we have stored all squares in the appropriate structure, we can find the squares we are looking for in constant time.

The other two cases, checking the diagonals and the number of squares, can be done in constant time and so also the necessary merge of colors. Merge of colors can be executed in constant time if we introduce an auxiliary array mapping the original coloring to the one after the merge rather than overriding the color of each edge separately.

One more thing to note is that looking for edges in the relation $\bar{\tau}$ if it was necessary, can happen only once because after that both uv and uw are going to be colored and the cost of the coloring of the other squares stays constant.

Summarizing the time complexity is $O(\Delta + \Delta) = O(\Delta)$.

Space complexity is again defined by the most memory consuming structure, *squareOppositeEdges*. We know that we have to store opposite square edges for each edge (so we need once again an array of the size n^2) and that each such opposite edge should be accessible in a constant time based on the opposite edge endpoints – so a vector of size n needed. Besides, each edge can have maximally Δ such opposite edges so the space complexity for this structure is $O(n^2 + m(n + \Delta)) = O(mn)$.

5.6 REFINE COLORING

At this stage of the algorithm, all edges have been already colored and all auxiliary structures have been populated with the complete set of data. The coloring that we have calculated so far is as coarse as it is only possible regarding relation δ, but there is still a possibility to make the coloring even coarser using the relation $\bar{\tau}$.

We have used already the relation $\bar{\tau}$ in Algorithm 8, but it was only to optimize the number of new colors we are introducing. At that stage, not all edges were colored, and that is why the relation $\bar{\tau}$ couldn't deliver the final result. (It could be possible as well to move the complete application of the relation $\bar{\tau}$ into Algorithm 9 and let Algorithm 8 to always generate a new color, when it is asked for one).

The objective now is to check whether the extension of the coloring according to the relation $\bar{\tau}$ and delivered by Algorithm 8 is always unique. When it is not the case, and there was more than one possibility to extend the colors from other edges, it means that these edges could be as well colored using only one color, and current colors could be merged.

The routine from Algorithm 9 is really straightforward to follow. It takes every single edge bb' from $E(G - e)$ and based on all its square opposite edges pairs aa', cc' it checks whether the relation $\bar{\tau}$ holds for edges ab, bc and $a'b'$, $b'c'$ just like in the procedure *findOtherColor*. If the checked edges are in the relation $\bar{\tau}$, then their colors could be merged.

5.6.1 *Correctness*

The argumentation presented above should be clear, and the correctness of this merge is strongly based on the correctness of *findOtherColor* routine, and if it is correct, so is the merge.

5.6.2 *Complexity*

We iterate over all edges, and for each of them we get a list of *squareOppositeEdges* of the maximal length Δ, then we pick all possible pairs of edges out of the list, which could be done in $(\Delta(\Delta - 1))/2$ ways. The

Algorithm 9 Merge Colors By Tau

```
Require: squareOppositeEdges, storedSquares, adjacencyMatrix
  procedure mergeColorsByTau
    for all bb' ∈ E(G − e) do
      aa'cc'All = squareOppositeEdges[b][b']
      for all pair of edges aa', cc' ∈ aa'cc'All do
        ab ← adjacencyMatrix[a][b]
        bc ← adjacencyMatrix[b][c]
        if ab.color ≠ bc.color then
          a'b' ← adjacencyMatrix[a'][b']
          b'c' ← adjacencyMatrix[b'][c']

          abcd ← storedSquares.get(ba, bc)
          a'b'c'd' ← storedSquares.get(b'a', b'c')
          if abcd.empty() ∧ a'b'c'd'.empty() then
            mergeColors(ab, bc)
          end if
        end if
      end for
    end for
  end procedure
```

further processing is done in constant time, so the time complexity of this procedure equals $O(m\Delta^2)$.

5.7 CLEAN MISSING SQUARE EDGE PAIRS

After the last routine has finished the processing, we reached the coarsest coloring of the graph $G - e$, which we can achieve without knowing the endpoints of the removed edge e.

The procedure Algorithm 10, which runs at this stage of the reconstruction, doesn't bring a big value concerning the reconstruction itself, but it cleans the data in *missingSquareEdges* after all possible merges have been done and, in this way, reduces the number of entries needed for further processing, because this collection will be of our greatest interest to find the endpoints of the missing edge.

The routine above doesn't need any detailed explanation. We just iterate over each pair of edges out of *missingSquareEdges* and remove those with the same color.

5.7.1 *Correctness*

This routine only updates the state of the *missingSquareEdges* to the newest known coloring.

Algorithm 10 Clean Missing Square Edge Pairs

```
Require: storedSquares, missingSquareEdges
  procedure cleanMissingSquareEdgePairs
    for all uv ∈ missingSquareEdges do
      for all uw ∈ missingSquareEdges[u][v] do
        if uv.color = uw.volor then
          missingSquareEdges.remove(uv, uw)
        end if
      end for
    end for
  end procedure
```

5.7.2 *Complexity*

The maximal number of the elements in *missingSquareEdges* is equal to the maximal number of incident pairs of edges in the graph which is $O(m\Delta)$, and as we iterate over all the elements, so is also the time complexity.

Space complexity doesn't change.

5.8 define endpoints of the edge to reconstruct

In this final step of the reconstruction, we define two vertices of $G - e$ which, once connected, will form with the rest of the graph Cartesian product. The idea is to find a triple of edges, which are not a part of a product square in $G - e$ but were a part of a product square in G. The collected by us *missingSquareEdges* list contains the triple of edges, which should be extended by one more edge spanning this way a product square but because it may also contain edge triples that should be a part of a single factor, and which are not because of the coloring introduced so far being finer, than the coloring of G, we have to find a way to distinguish the correct missing square edges from the wrong ones.

In the first step of Algorithm 11, which has been extracted into the Algorithm 12, we iterate over all missing square edges pairs and group them by the first edge and the color of the second edge for quicker access in the further part.

Next, we iterate over the entries from *missingSquareEdges* and forming triples out of them in Algorithm 13. Then we connect the endpoints of the found triple by the edge f and check, whether its insertion results in a correct Cartesian product. We do the check factorizing the graph $G - e + f$ and looking for an outcome having at least two factors. When we find a suitable edge f, the further processing is going to be suspended and the resulting edge f is returned, but if the proposed edge f doesn't give a correct result, then colors of edges out

Algorithm 11 Find endpoints of the edge to reconstruct

```
Require: missingSquareEdges
  procedure FINDEDGETORECONSTRUCT
    GROUPMISSINGSQUAREEDGES( )
    for all {uv, uw} ∈ missingSquareEdges do
      if uv.color ≠ uw.color then
        f ← FINDPOTENTIALMISSINGEDGE(uv, uw)
        if f ≠ ∅ then
          reconstructedG ← reconstruct(f)
          notPrime ← factorize(reconstructedG)
          if notPrime then
            return f
          else
            UPDATECOLORGROUPING(uv, uw)
            MERGECOLORS(uv, uw)
          end if
        end if
      end if
    end for
  end procedure
```

Algorithm 12 Group missing square edges by first edge and color

```
Require: missingSquareEdges
  procedure GROUPMISSINGSQUAREEDGES
    missingSquareEdgesByColor ← newStructure()
    for all {uv, uw} ∈ missingSquareEdges do
      missingSquareEdgesByColor[u][v][uw.color].add(uw)
    end for
  end procedure
```

of the triple used to select endpoints of f are merged, the structure *missingSquareEdgesByColor* is adjusted according to the new existing colors and the algorithm proceeds to another combination of colors. Algorithm 14 looks after the grouping in *missingSquareEdgesByColor* to stay up to date.

Algorithm 13 Find Potential Missing Edge Endpoints

```
Require: missingSquareEdgesByColor
  procedure FINDPOTENTIALMISSINGEDGE(uv, uw)
      vw' ← missingSquareEdgesByColor[v][u][uw.color]
      if vw' ≠ ∅ then
          return {w, w'}
      else
          wv' ← missingSquareEdges[w][u][uv.color]
          if wv' ≠ ∅ then
              return {v, v'}
          else
              return ∅
          end if
      end if
  end procedure
```

Algorithm 14 Update missing square edges color grouping

```
Require: missingSquareEdgesByColor
  procedure UPDATECOLORGROUPING(uv, uw)
      for all edgesByColor ∈ missingSquareEdgesByColor do
          if uv.color < uw.color then
              edgesByColor[uv.color].addAll(edgesByColor[uw.color])
              edgesByColor[uw.color].remove()
          else
              edgesByColor[uw.color].addAll(edgesByColor[uv.color])
              edgesByColor[uv.color].remove()
          end if
      end for
  end procedure
```

5.8.1 *Correctness*

The correctness of this part of the reconstruction relies heavily on the Cartesian product property, which says that two edges colored differently must span a product square.

Knowing this statement, we can just try out all the possibilities we have, and it is what we do by going through all of the entries from *missingSquareEdges*.

There was only one edge e removed from the graph G, so all the product squares in G containing e can be projected to triples of edges in $G - e$ and these are the triples we are looking for. Going one by one over missing square edges pairs, we try to extend the pair by an edge to make it to a triple. We have shown in the initial part of this publication, that such a triple should have two edges of the same color on both ends and a single edge of another color in the middle, what makes finding the third edge to the triple a bit easier, but still we have to consider both edges out of the initial pair as potential middle edge in the triple.

Figure 5.4 shows an example for finding a triple. For a missing square edges pair uw, uv there will be no triple taking the edge uv as the triple's middle edge, because of the missing edge vv', but changing the triple middle edge to uw will give us a correct triple uv, uw, wv'.

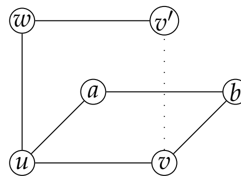


Figure 5.4: Found missinge square edges triple for edges uv, uw, wv'

Having found the desired edges triple, which doesn't belong to any product square in $G - e$, we extend it by an edge f to make the missing product square present in the product. Factorization of the obtained in this way graph $G - e + f$ tells us, whether the reconstruction was correct. However, if factorization recognizes $G - e + f$ as a prime graph, we can be sure that found by us triple never belonged to a product square in G and we can merge the colors contained in the triple.

In this case, we just move to the next pair of missing square edges, taking the new coloring into account. Analyzing all possible edges triples gives us certainty, that at some point the triple selected by us, will give us correct endpoints for the edge f to reconstruct Cartesian product.

There are also some optimizations possible, which can give us the answer, whether selected by us endpoints are correct for the edge to be reconstructed f, which are:

- Two or more triples of the same colors point to different endpoints for f.
- There is no third edge forming a triple with already selected pair of missing square edges.

These cases, however, don't exist in every input graph $G - e$, and that is why they won't improve the overall complexity of the algorithm.

The Figure 4.7 from the Section 4.2.3 shows already an example of a graph $G - e$ after the coloring, which contains misleading triples.

5.8.2 *Complexity*

The grouping of missing square edges pairs is done in $O(m\Delta)$ time because that is the maximal number of such pairs.

The for-loop from the beginning of the Algorithm 11 iterate over all entries in *missingSquareEdges* which number cannot be bigger than $O(m\Delta)$. Having a pair of edges in different colors from the entry, we extend it in constant time by a third edge forming with the previous two a triple of edges, which two endpoints indicate possible endpoints of the edge to reconstruct f. Having found such a candidate edge, we insert it into the graph $G - e$ in constant time and check the correctness of selected f with the factorization algorithm from [10] in $O(m)$. If $G - e + f$ is a nontrivial Cartesian product, then nothing more needs to be done, as f is the missing edge we were looking for. In another case, we delete the edge f out of the graph and merge colors belonging to the triple, both in constant time. We also update the grouping in the structure *missingSquareEdgesByColor* going through all of its entries, which cannot be bigger, than the number of edges m.

One should notice, that each edges triple pointing to potential endpoints of the missing edge f must contain edges of two different colors, so the reconstruction of an edge and factorization can be performed only $O(\Delta)$ times because this is the number of colors we have starting the Algorithm 11 and after each failure, we merge two colors.

Wrapping it all up the time complexity equals to $O(m\Delta + \Delta(m + m)) = O(m\Delta)$.

The newly introduced structure for storing missing square edges pairs by the first edge and the second edge's color has moderate complexity. At its first level, we have a matrix where each populated cell corresponds to the first edge of each entry from *missingSquareEdges*, and its complexity is equal to the complexity of an adjacency matrix, which is $O(n^2)$. The cell itself contains a vector with a cell for each color index, and inside of each such a cell we put a linked list of second edges. We know, that at this point we have no more than Δ colors, and each edge cannot have more than Δ incident edges, so the total complexity can be expressed by $O(n^2 + m(\Delta + \Delta))$.

Part VI

SUMMARY

6 SUMMARY

6.1 RESULTS

Our research delivers some profound results regarding the edge-reconstruction of Cartesian product graphs.

In Part II we point out, that using the results from [3] about reconstructability of finite Cartesian products and [5] showing that reconstructability implies edge-reconstructability, we may be certain that the edge-reconstructability is possible.

Furthermore, in Part III we show two other important qualities of the edge-reconstruction. First, that the edge-reconstructability is possible also for infinite Cartesian products and second, that an edge-reconstruction of a Cartesian product being initially a product of only two factors and one of them being K_2 may deliver many results, all of them equal up to isomorphisms.

In Part IV we briefly present a trivial edge-reconstruction algorithm of time complexity $O(mn^2)$, and continue with a more sophisticated one, which uses the properties of relations δ, $\bar{\tau}$ and delivers the result for the edge-reconstruction in $O(m\Delta^2)$ time.

In Part V we are giving the algorithm even a deeper look and analyze every single step of the algorithm, as well as all needed data structures that make it possible to achieve the desired time complexity.

6.2 OPEN PROBLEMS

The problem of reconstruction can be also presented, as a composition of two problems, namely recognition and weak reconstruction. The problem of edge-reconstructability for finite Cartesian products may be considered as solved, but for the infinite ones, we have merely solved the problem of the weak reconstruction leaving the problem of recognition for further studies.

The second area containing a set of open problems are twisted Cartesian products. Here we can observe the following cases for further study:

It is not known whether the deck of vertex deleted subgraphs of a twisted Cartesian product characterizes membership in the class, nor whether twisted Cartesian products are uniquely determined by single vertex deleted subgraphs.

The analogous problems for edge deleted subgraphs of twisted Cartesian products are also open.

Finally, if weak vertex- or edge-reconstruction is unique, the question whether there exist efficient reconstruction algorithms arises.

Part VII

APPENDIX

7 WEAK RECONSTRUCTION COMPLEXITY OF CARTESIAN PRODUCTS

7.1 PREAMBLE

This appendix pertains to the vertex-reconstruction of Cartesian products. It corrects a subtle error in the derivation of the complexity of the reconstruction algorithm in [6] and can be read independently of the remainder of the thesis.

7.2 INTRODUCTION

In [23] Ulam asked whether a graph G is uniquely determined up to isomorphisms by its deck, that is, by the set of all graphs $G - x$ obtained from G by deleting a vertex x and all edges incident to it. This led to the *Reconstruction Conjecture*, also known as *Ulam's Conjecture*, that any two graphs on at least three vertices with the same deck are isomorphic. Actually the conjecture was already formulated 1942 for finite graphs in the Ph.D. Thesis of Kelly [16], but this went unnoticed for a long time. For infinite graphs the conjecture is false, but for finite graphs it is still open. When reconstructing a class of graphs, the problem partitions into the subproblems *recognition* and *weak reconstruction*. The first consists of showing that membership in the class is determined by the deck, and the latter that nonisomorphic members of the class have different decks.

For the class of nontrivial finite Cartesian products, that is, Cartesian product of at least two nontrivial factors, Dörfler [3] proved the validity of Ulam's conjecture. This was supplemented by Sims and Holton [21, 22], who showed that the weak reconstruction problem can be solved from a single vertex deleted subgraph for nontrivial connected, finite Cartesian products. In [15] Imrich and Žerovnik extended this to infnite, nontrivial connected Cartesian products.

Later Hagauer and Žerovnik [6] pubished a paper in which they claimed that each nontrivial Cartesian product G can be reconstructed in $O(mn(\Delta^2 + m \log n)$ time from any vertex deleted subgraph $G - x$, where m is the size, n the order, and Δ the maximal degree of G. In this note we correct an error in the computation of the complexity of the algorithm of Hagauer and Žerovnik, which as such increases the complexity to $O(mn(\Delta^4 + m \log n)$.

However, we wish to mention that results in [10] and [17] allow to somewhat reduce the complexity again to $O(mn + \Delta^2(m + \Delta^4))$. We do not present the details, because the methods of the thesis allow

a further improvement to $O(m(n+\Delta^2)$. It will be the subject of a separate publication.

7.3 PRELIMINARIES

The original publication [6] describes the reconstruction algorithm in a very clear and detailed way as well as contains a solidly carried out proof of its correctness, including all of the needed definitions. For readers interested in this matter, we recommend going back to the aforementioned publication, as here we are going to introduce only the minimal information allowing us to follow the algorithm steps and analyze its complexity.

Algorithm 15 Skeleton of the algorithm

```
if G − x ≃ C_8 then
    G = P_3 □ P_3
    return G
end if
for all x′ ∈ V(G − x) do
    G = Construction 1(x′)
    if G is Cartesian product then
        return G
    end if
end for
for all s ∈ V(G − x) do
    for all u, v ∈ N(s) do
        G = Construction 2(s, u, v)
        if G is Cartesian product then
            return G
        end if
    end for
end for
```

We can distinguish three different parts of the Algorithm 15, each focused on a different input graph case.

The first one delivers only the correct result for a particular case where the input graph is isomorphic to the cycle of eight vertices.

The second part runs *Construction 1* for each vertex out of $V(G-x)$ and gives a correct reconstruction if the original graph G was a product containing at least one K_2 factor among its nontrivial factors.

The third part finds a reconstruction of the input graph $G-x$ in any other case, and this is the part in whose complexity we are interested.

The first thing the algorithm does for any given vertex s as preprocessing, is ordering the vertices in the BFS-order taking the vertex s for the start vertex. This way all vertices are assigned to levels, and the level of any vertex $z \in V(G-x)$ is equal to the $d_{G-x}(s,z)$.

Algorithm 16 Construction 2

Require: vertices s, u, v

Start Insert cross-edges

for vertex y at level 2 which is a neighbor of u and has exactly one more neighbor w at level 1, and $vw \in E(G-x)$ **do**

Insert edge xy

end for

for vertex y at level 2 which is a neighbor of v and has exactly one more neighbor w at level 1, and $uw \in E(G-x)$ **do**

Insert edge xy

end for

End

Start Insert up-edges

for vertex y at level 3 with at least two down-edges, for which $u, v \in I_{(G-x)}(s, y)$ **do**

Insert edge xy

end for

for vertex y at level 3 with exactly one down-edge, and which is a good candidate **do**

Insert edge xy

end for

for vertex y at level 3 with exactly one down-neighbor w, where y is not a bad candidate and w has no up-neighbor, which is a good candidate **do**

Insert edge xy

end for

End

Check if the new graph is a Cartesian product graph

Moreover, the BFS-levels allow us to group edges of each vertex, and with edges also this vertex' neighbors, into three groups: up-, cross-, down-edges and up-, cross-, down-neighbors. If an edge e leads from a vertex to another vertex of a higher, equal, or lower level, we speak about up-, cross- and down-edge respectively. For neighbors it works via analogy.

The *interval* $I_{G-x}(z_1, z_2)$ between two vertices $z_1, z_2 \in V(G-x)$ is the set of vertices lying on any shortest path between z_1 and z_2.

The last term we need to define is the term of candidate, and its good and bad versions. A vertex y at level 3 is a *candidate* (for being connected to x) if it has down-degree one in $G-x$, $|I_{G-x}(s,y)| = 4$ and either u or v is in $I_{G-x}(s,y)$ (i.e. the interval between s and w, the unique down-neighbor of y, is a path in $G-x$ containing u or v). A candidate y is *vw-candidate* if $I_{G-x}(s,y) = \{s,v,w,y\}$. A *vw-candidate* is a *good candidate* if there is an up-neighbor q of y such that $I_{G-x}(s,q)$ contains both u and v. A *vw-candidate* y is a *bad-candidate* if y is not good and there is an up-neighbor z of y such that $N(s) \cap (I_{G-x}(s,q) - \{u,v\}) \neq \emptyset$.

7.4 COMPLEXITY ANALYSIS

The first thing regarding the procedure *Construction 2*, which needs a small clarification, is the breath-first search done for each vertex s in a preprocessing before the call of the procedure itself. The question that one can ask is: when the call of *Construction 2* is already contained in a loop over all vertices s, can the preprocessing happen in the same loop? To answer this question, let's take a look at the pieces of information which are computed during this preprocessing.

For each vertex in $G-x$, the edges incident to it are going to be grouped into three groups: down-, cross- and up-edges. This grouping makes it easier to traverse the vertices of the graph in the BFS-order during the runtime of *Construction 2* and because this grouping is reusable among all pairs of vertices $u, v \in N(s)$ we can recompute it once for each iteration over s.

The distances between vertices, which are also a result of the preprocessing, need more attention. As one will be able to see in the further analysis, it is not enough to calculate only the distances from s to other vertices, because to achieve a better complexity *Construction 2* needs precomputed distances between many different pairs of vertices lying in all possible levels. That is why even before iterating over all triples of vertices s, u, v, and calling *Construction 2*, we need to loop over every single vertex, run BFS procedure for each of them and store the distances to other vertices in a matrix.

The time complexity of the breath-first search is $O(m)$, and repeating it for each vertex brings us to $O(nm)$. This precomputation does not have any influence on the overall complexity of the algorithm because,

as one can see, all calls of *Construction 1* have the same complexity of $O(nm)$.

The matrix for storing the distances between each vertex is a two-dimensional matrix, where the first dimension reflects all vertices used as a starting point for BFS, and the second dimension stores the distances from the starting vertex to all other vertices. It is easy to see, that the size of the array is $O(n^2)$.

Having clarified the exact shape and complexity of the preprocessing, let's jump straight to the cumbersome complexity of *Construction 2*.

The complexity estimated for the reconstruction of cross-edges in Step 1 is correct for a single vertex y and indeed constant. The checks like whether y has only one more down-neighbor w and whether $vw \in E(G - x)$ can be done in constant time using the auxiliary structures of grouped edges and the distance matrix. However, there can be up to $O(\Delta)$ such y vertices, and for each of them, we have to perform the same set of checks.

This observation sets the time complexity of Step 1 to $O(\Delta)$ but doesn't change the overall complexity of the procedure.

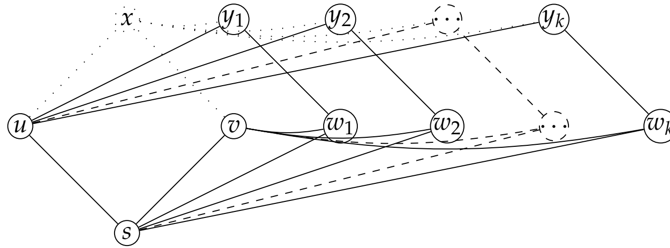


Figure 7.1: Multiple cross-edges.

Step 2 is comprised of three different sub-steps, which allow us to reconstruct the up-edges of x. Each sub-step takes a specific vertex y from level 3 as a parameter and depending on the number of down-edges, either Step 2.1 or Step 2.2 or Step 2.3 is going to be called.

Step 2.1 takes a vertex y with two and more down-neighbors and checks if $u, v \in I_{G-x}(s, y)$. This test is relatively easy, we know that u and v are direct neighbors of s, so if only $d(y, u) = 2$ and $d(y, v) = 2$ we can be sure, that the desired condition is met. (Actually, as we probably have used either u or v to get to the vertex y, only the distance check for the other vertex is necessary). Having the distances between every two vertices precomputed, it is clear, that this distance check is done in constant time.

In the Figure 7.2 u, v belong to $I_{G-x}(s, y_1^1)$ and $I_{G-x}(s, y_k^1)$, which is not true for other two relevant intervals, $I_{G-x}(s, y_1^l)$ and $I_{G-x}(s, y_k^l)$. That is why only y_1^1 and y_k^1 are recognized as up-neighbors of x.

Step 2.2 and 2.3 take as an input parameter a vertex y with only a single down-neighbor, which is a candidate. We say that y is a vw-candidate if $I_{G-x}(s, y) = \{s, v, w, y\}$. We know already, that s is a single down-neighbor of v as well as w is a single down-neighbor of y.

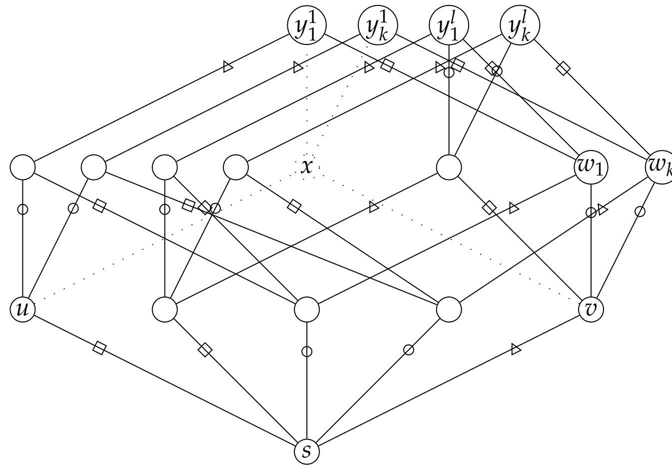


Figure 7.2: Step 2.1 – y_1^1 and y_k^1 recognized as up-neighbors of x

That is why only the check whether v is a single down-neighbor of w defines y as vw-candidate or not and can be done in constant time thanks to the edges that have been pre-grouped.

For vertices y recognized as vw-candidates we can proceed with Step 2.2, whose objective is to find a y which is a good candidate. To perform this test, we have to examine all up-neighbors of y, say q, and check if for any q $u, v \in I_{G-x}(s, q)$. To answer this question, it is again enough to check some distances, namely $d(q, u) = 3$ and $d(q, v) = 3$. Of course, it can be done in constant time for each single q and for all vertices q being an up-neighbor of a single y in $O(\Delta)$ time.

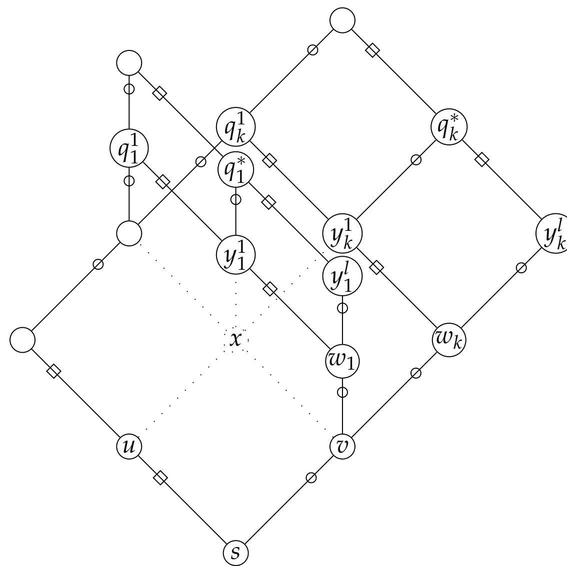


Figure 7.3: Step 2.2 – y_1^1 and y_k^1 are good candidates

Figure 7.3 shows y_1^1 and y_k^1 as good candidates because the interval from the vertex q_1^1, and q_k^1 respectively, to the vertex s contains both u and v. Other vertices q don't have this property.

Step 2.3 is needed only when among some sets of vertices y, being vw-candidates, no good candidates have been found. In this case, we have to reexamine the aforementioned candidates and find the ones which are not bad.

To check whether a given vw-candidate y is a bad one we have to again iterate over its up-neighbors q and in each iteration check if $N(s) \cap (I_{G-x}(s,q) - \{u,v\}) \neq \emptyset$. This check requires from us to go over all vertices $r \in N(s) - \{u,v\}$ and for each r test if $d(q,r) = 3$. Having found any such r, we can mark y as a bad candidate and move to the next one.

The time complexity for a single y is $O(\Delta^2)$ as we have to consider all its up-neighbors q, which could be in the number of $O(\Delta)$, and for each q iterate again over the neighborhood of s, also with the potential size of $O(\Delta)$, and finally do the distance check for the pair q,r in constant time.

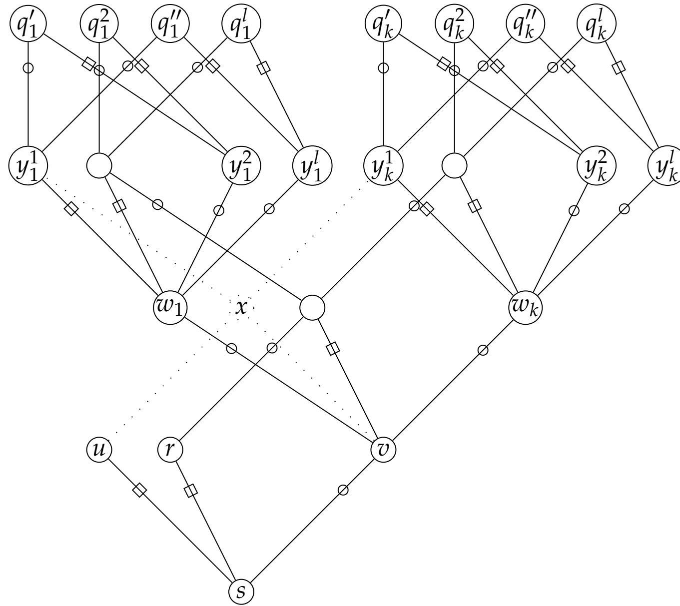


Figure 7.4: Step 2.3 – y_1^1 and y_k^1 are not bad candidates.

Figure 7.4 shows y_1^1 and y_k^1 as candidates not being bad because only these two vertices have up-neighbors q, such that $r \notin I_{G-x}(s,q)$.

Having all sub-steps of Step 2 summarized, we see, that the most expensive sub-step is the one excluding each single y being a bad candidate in $O(\Delta^2)$ time. That is exactly the time proposed by Hagauer and Žerovnik in [6], but we shouldn't forget that this is the time complexity for a single vertex y and it must be multiplied by the number of vertices y being in our concern.

We see that all vertices y are the second-line neighbors of u and v, and we can use both of them to find appropriate vertices y. Each of u and v can have $O(\Delta)$ suitable up-neighbors w with following $O(\Delta)$ suitable up-neighbors y, so the total number of vertices y is equal to $O(\Delta^2)$ and thus the complexity of Step 2 equals to $O(\Delta^4)$. (One can also take all vertices from level 3 and consider them as vertices y but then there is no better way to estimate their number than $O(n)$).

The figures included along the analysis of each sub-step help to visualize the reasoning concerning the number of suitable vertices y. Considering the Figure 7.2 and adding to vertex s edges of type $\bigcirc$ or $\square$ increases the number of vertices w and y respectively. In the graphs from Figure 7.3 and Figure 7.4, adding to the vertex v $\bigcirc$-edges increases the number of vertices w and adding to any vertex w $\bigcirc$-edges increases the number of vertices y.

The complexity of Step 3 has already been mentioned, and as it is the call to recognize a graph as a Cartesian product, its time complexity is $O(m)$.

Taking the above calculation into consideration, one can see that the actual time complexity of *Construction 2* is $O(\Delta) + O(\Delta^4) + O(m)$.

BIBLIOGRAPHY

[1] N. Chiba and T. Nishizeki. “Arboricity and subgraph listing algorithms.” In: *SIAM J. Comput.* 14 (1985), pp. 210–223.

[2] W. Dörfler. “On the edge-reconstruction of graphs.” In: *Bull. Austral. Math. Soc.* 10 (1974), pp. 79–84.

[3] W. Dörfler. “Some results on the reconstruction of graphs. Infinite and finite sets (Colloq., Keszthely, 1973; dedicated to P. Erdős on his 60th birthday).” In: *Colloq. Math. Soc. János Bolyai* 10 (1975), pp. 361–363.

[4] T. Feder. “Product graph representations.” In: *J. Graph Theory* 16 (1992), pp. 467–488.

[5] D.L. Greenwell. “Reconstructing graphs.” In: *Proc. Amer. Math. Soc.* 30 (1971), pp. 431 –433.

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

[7] R. Hammack, W. Imrich, and S. Klavžar. *Handbook of product graphs - Second edition*. CRC Press, 2011.

[8] F. Harary. “On the reconstruction of a graph from a collection of subgraphs. In Theory of Graphs and its Applications (Proc. Sympos. Smolenice, 1963).” In: *Publ. House Czechoslovak Acad. Sci., Prague* (1964), pp. 47–52.

[9] W. Imrich. “Über das schwache kartesische Produkt von Graphen.” In: *J. Combinatorial Theory Ser. B* 11 (1971), pp. 1–16.

[10] W. Imrich and I. Peterin. “Recognizing Cartesian products in linear time.” In: *Discrete Mathematics* 307 (2007), pp. 472–483.

[11] W. Imrich and M. Wardyński. “An Algorithm for the Weak Reconstruction of Cartesian-Product Graphs.” In: ().

[12] W. Imrich and M. Wardyński. “An Algorithm for the Weak Edge-Reconstruction of Cartesian Product Graphs.” In: *submitted for publication* (2020).

[13] W. Imrich and M. Wardyński. “Weak Edge-Reconstruction for Cartesian Product Graphs.” In: *submitted for publication* (2020).

[14] W. Imrich and J. Žerovnik. “Factoring Cartesian-Product Graphs.” In: *J. Graph Th.* 18 (1994), pp. 557–567.

[15] W. Imrich and J. Žerovnik. “On the weak reconstruction of Cartesian-product graphs.” In: *Discrete Math.* 150 (1996), pp. 167–178.

[16] P.J. Kelly. "On Isometric Transformations." PhD thesis. Madison: The University of Wisconsin, 1942.

[17] T. Kupka. "A Local Approach for Embedding Graphs into Cartesian Products." PhD thesis. Ostrava, Czech Republic: Tech. Univ. Ostrava, 2013.

[18] D.J. Miller. "Weak Cartesian product of graphs." In: *Colloquium Math.* 21 (1970), pp. 55–74.

[19] V. Müller. "The edge reconstruction hypothesis is true for graphs with more than $n \log_2 n$ edges." In: *J. Combinatorial Theory Ser. B* 22 (1977), pp. 281 –283.

[20] G. Sabidussi. "Graph Multiplication." In: *Math. Z.* 72 (1960), pp. 446–457.

[21] J. Sims. "Stability of the cartesian product of graphs." MA thesis. University of Melbourne, 1976.

[22] J. Sims and D.A. Holton. "Stability of cartesian products." In: *Combin. Theory Ser.* 25 (1980), pp. 258–282.

[23] S. M. Ulam. *A Collection of Mathematical Problems*. 1960, p. 29.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Thank you very much for your feedback and contribution.

Final Version as of May 28, 2020 (version 1.0).