Dresden University of Technology

Department of Computer Science

Commutative Polymorphisms of Core Triads

Author:
Michael Wernthaler

Student number:
s8179597

Examined and approved on

_____

by the following examiners:

_____
Prof. Dr. Manuel Bodirsky (supervisor)

_____
Prof. Dr. Sebastian Rudolph (co-supervisor)

Submitted in total fulfilment of the requirements of the degree of Bachelor of Science

April 23, 2021

Declaration

I herewith formally declare that I, Michael Wernthaler, have written the submitted thesis independently pursuant to § 21 paragraph 6 of the Examination Regulations for the Bachelor Degree Program of the facutly of Computer Science at TU Dresden. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. I am aware, that in case of an attempt at deception based on plagiarism, the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

Dresden, April 23, 2021	Michael Wernthaler
Location, Date	Name

Abstract

It has been known for a while that for a given digraph H the complexity of the constraint satisfaction problem for H only depends on the set of polymorphisms of H. From recent results it follows that the H-colouring problem is in P if H has a so-called (4-ary) Siggers polymorphism. In this thesis, we focus on the case where H is a *triad* (a simplified tree-structure) and describe an algorithm with various optimisations that checks the existence of Siggers polymorphisms for triads up to a certain number of vertices.

Contents

Introduction

Let $H = (V, E)$ be a finite digraph. The *H-colouring problem* (also called the *constraint satisfaction problem for H*) is the problem of deciding for a given finite digraph G whether there exists a homomorphism from G to H. Note that if $H = K_k$, the clique with k vertices, then the H-colouring problem equals the famous k-colouring problem, which is NP-hard for $k \geq 3$ and which can be solved in polynomial time if $k \leq 2$.

It has been known for a while that the complexity of the H-colouring problem only depends on the set of *polymorphisms* of H. It follows from results of Bulatov [2] and Zhuk [8] from 2017 that the H-colouring problem is in P if H has a so-called *(4-ary) Siggers* polymorphism, i.e., an operation $s\colon V^4 \to V$ where V denotes the vertices of H and which satisfies for all $a, e, r \in V$

$$s(a, r, e, a) = s(r, a, r, e).$$

Before these results, the complexity of H-colouring problem was open even if H is an orientation of a tree. It is not obvious at all how an orientation of a tree looks like if it has a Siggers polymorphism. In fact, this question is already open if H is a *triad*, i.e., an orientation of a tree which has a single vertex of degree 3 and otherwise only vertices of degree 2 and 1. Jakob Bulin claims that the following triad with 22 vertices has no Siggers polymorphism.

$$10110000, 1001111, 010111$$

Here, 0 stands for forward edge, 1 stands for backward edge, and the three words stand for the three paths that leave the vertex of degree 3 of the triad. He also claims that all smaller triads do have a Siggers polymorphism and conjectures that in this case the Singleton-Arc-Consistency algorithm can solve the H-colouring problem [3, p. 17].

The goal of this thesis will be to verify the claim by Jakub Bulin that the triad 10110000,1001111,010111 has no Siggers polymorphism and that all smaller triads do have a Siggers polymorphism. Moreover, we will present a new-found triad of the same size that has no Siggers polymorphism either. Lastly, we provide a proof of the non-existence of a commutative polymorphism for our new triad that is not based on running a computer program but understandable to humans. In this way, we hope to contribute to understanding the hidden relations between the structure of a triad and the complexity of the corresponding CSP.

The thesis is organised as follows. First, we formally introduce the preliminary notions and notation necessary for reading the thesis. In Section 2 we will present an algorithm that enumerates all core triads up to a fixed number of vertices. Section 3 deals with the verification of Bulin's proposition. Section 4 concludes this work.

1 Preliminaries

In this thesis, we mostly follow the notation in the script for the lecture *Graph Homomorphisms and Universal Algebra* [1] given in summer 2020 by Manuel Bodirsky.

1.1 Graphs

A *directed graph* (also *digraph*) is a pair $G = (V, E)$ of disjunctive sets where $E \subseteq V^2$. We call the elements of $V = V(G)$ *vertices* of G and $E = E(G)$ its *edges*. The digraph G is called *finite* or *infinite* depending on whether $V(G)$ is finite or infinite. However, since this thesis deals exclusively with finite digraphs, we only write graphs instead of finite digraphs. The *transpose* of $G = (V, E)$ is the graph $G^T = (V, E^T)$ where $E^T = \{(y, x) \mid (x, y) \in E(G)\}$.

A graph G' is a subgraph of G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. We write $G - x$ for the subgraph $G' = (V(G) \setminus \{x\}), E(G')$ where $(u, v) \in E(G')$ if and only if $(u, v) \in E(G)$ for all $u, v \in V(G')$.

Two vertices $x, y \in G$ are *adjacent* in G and are called *neighbours* of each other if either $(x, y) \in E(G)$ or $(y, x) \in E(G)$. The number of neighbours of x in G is the *degree* of x.

A *path* (from u_1 to u_k in G) is a sequence $(u_1, \ldots, u_k)$ of vertices of G and a sequence $(e_1, \ldots, e_{k-1})$ of edges of G such that either $e_i = (u_i, u_{i+1}) \in E(G)$ or $e_i = (u_{i+1}, u_i) \in E(G)$, for every $1 \leq i < k$. The number of edges of a path is its *length*. A graph G is called *connected* if for all $x, y \in V(G)$ there is a path from x to y in G. A maximal connected subgraph of G is a *component* of G.

A sequence $(u_0, \ldots, u_{k-1})$ of pairwise distinct vertices and a sequence of edges $(e_0, \ldots, e_{k-1})$ is called a *cycle* (of G) if $(u_0, \ldots, u_{k-1}, u_0)$ and $(e_0, \ldots, e_{k-1})$ form a path. The *length* of a cycle is again the number of its edges. A graph is called acyclic, if it does not contain a cycle.

1.2 Triads

A connected acyclic graph G is called a *tree*. The vertices of degree 1 of a tree are called its *leaves*. Note that any two vertices $v, w \in V(G)$ can be connected by a unique shortest path, so we consider it to be *the path* from v to w.

A *triad* is a tree which has a single vertex of degree 3 (also called *root*) and otherwise only vertices of degree 2 and 1. Most of the time we will use a notation of the form $\mathbb{T} = p_1, p_2, p_3$ where $p_i \in \{\{0, 1\}^n \mid n \in \mathbb{N}^+\}$ for $i \in \{1, 2, 3\}$. Here, 0 stands for forward edge, 1 stands for backward edge, and the three words stand for the three paths that leave the root of the triad. We call p_i an *arm* of $\mathbb{T}$.

1.3 Cores

Let H be a graph. We call a mapping $h\colon V(G) \to V(H)$ a *homomorphism* from G to H, if $(h(u), h(v)) \in E(H)$ whenever $(u, v) \in E(G)$. If such a homomorphism exists between G and H, we say that G *homomorphically maps* to H, and write $G \to H$.

G is called *isomorphic* to H, denoted by $G \simeq H$, if there exists a bijection $\phi\colon V(G) \to V(H)$ with $(x, y) \in E(G) \Leftrightarrow (\phi(x), \phi(y)) \in E(H)$ for all $x, y \in V$. We call such a mapping ϕ an *isomorphism*. We usually do not distinguish between isomorphic graphs, thus we write $G = H$ instead of $G \simeq H$.

An *endomorphism* of a graph H is a homomorphism from H to H. An *automorphism* of a graph H is an isomorphism from H to H. A graph H is called a *core*, if every endomorphism of H is an automorphism. A subgraph G of H is called *a core of* H, if H is homomorphically equivalent to G and G is a core.

1.4 Polymorphisms

Let H_1 and H_2 be two graphs. Then the *(cross-, direct-, categorical-) product* $H_1 \times H_2$ of H_1 and H_2 is the graph with vertex set $V(H_1) \times V(H_2)$; the pair $((u_1, u_2), (v_1, v_2))$ is in $E(H_1 \times H_2)$ if $(u_1, v_1) \in E(H_1)$ and $(u_2, v_2) \in E(H_2)$. The *n-th power* H^n of a graph H is inductively defined as follows. H^1 is by definition H. If H^i is already defined, then H^{i+1} is $H^i \times H$.

Let H be a graph and $k \geq 1$. Then a *polymorphism of H of arity k* is a homomorphism from H^k to H. In other words, a mapping $f\colon V(H)^k \to V(H)$ is a polymorphism of H if and only if $(f(u_1, \ldots, u_k), f(v_1, \ldots, v_k)) \in E(H)$ whenever $(u_1, v_1), \ldots, (u_k, v_k)$ are in $E(H)$.

1.5 The H-colouring Problem

When does a given graph G homomorphically map to a graph H? For every graph H, this question defines a computational problem, which is known as the *H-colouring problem*, also denoted CSP(H). The input of this problem consists of a graph G, and the question is whether there exists a homomorphism from G to H. A common variant of this problem is the *precoloured H-colouring problem* where the input consists of a graph G together with a mapping f from a subset of $V(G)$ to $V(H)$. In this case, the question is whether there exists an extension of f to all of $V(G)$ which is a homomorphism from G to H. It is clear that the H-colouring problem reduces to the precoloured H-colouring problem (it is a special case: the partial map might have an empty domain).

1.6 The Arc-consistency Procedure

The arc-consistency procedure is one of the most studied algorithms for solving constraint satisfaction problems. It found its first mentions in [6],[5] and is often referred to as the *consistency check procedure*.

Let H be a graph, and let G be an instance of CSP(H). The idea of the procedure is to maintain a list $L(x)$ of vertices from $V(H)$ for each vertex $x \in G$. Each element in the list of x represents a candidate for an image of x under a homomorphism from G to H. The procedure successively removes vertices from the lists by performing consistency checks. For this purpose the algorithm uses only two rules: if (x, y) is an edge in G, then

- remove u from $L(x)$, if there is no $v \in L(y)$ with $(u, v) \in E(H)$;
- remove v from $L(y)$, if there is no $u \in L(x)$ with $(u, v) \in E(H)$.

Eventually, we cannot remove any vertex from any list, then the graph G together with the resulting lists is called *arc consistent*. If one of the lists is empty, it is clear that a homomorphism from G to H does not exist, and we reject. The pseudo-code of the entire arc-consistency procedure is displayed in Algorithm 1.

If G is an instance of precoloured CSP for H, we simply run the modification of AC_H which starts with $L(x) := \{c(x)\}$ for all $x \in V(G)$ in the range of the precolouring function c instead of $L(x) := V(H)$.

Note that for any graph H, if AC_H rejects an instance of CSP(H), then it clearly has no solution. The converse implication does not hold in general. For instance, let H be K_2, and let G be K_3. In this case, AC_H does not remove any vertex from any list but obviously there is no homomorphism from K_3 to K_2.

Algorithm 1: $\mathrm{AC}_H(G)$

Input: a graph G
Data: for every $x \in V(G)$ a list $L(x)$ of elements of $V(H)$
Output: arc consistent lists for every $x \in V(G)$

```
repeat
    foreach x ∈ V(G) do
        foreach u ∈ L(x) do
            if there is no v ∈ V(H) such that (u, v) ∈ E(H) then
                Remove u from L(x)
        if L(x) is empty then
            reject
until No list changes
accept
```

However, there are graphs H where the AC_H is a complete decision procedure for $\mathrm{CSP}(H)$ in the sense that it rejects an instance G of $\mathrm{CSP}(H)$ if and only if G does not homomorphically map to H. In this case, we say that AC solves $\mathrm{CSP}(H)$.

2 Enumerating Triads

Our goal of the thesis will be to confirm the following proposition due to Jakub Bulin that will eventually be verified in Section 3.

Proposition 2.1. *The triad* $01001111, 0110000, 101000$ *is a smallest triad without a Siggers polymorphism.*

As a basis for the proof, this section provides an algorithm that enumerates all triads up to a fixed number of vertices.

However, to reduce the number of cases to look at we consider only triads that are cores (*core triads*) since a graph H and its core have the same CSP. Also, note that the core of a triad is either a triad or a path and the CSP of a path is always in P. Thus, we pose the following question.

Question 2.1. *When is a triad a core?*

One way to answer this question is given by the following lemma.

Lemma 2.1. *Let* $\mathbb{T}$ *be a finite tree. The following are equivalent:*

1. $\mathbb{T}$ *is a core;*
2. $\mathrm{End}(\mathbb{T}) = \{id\}$*;*
3. $\mathrm{AC}_{\mathbb{T}}(\mathbb{T})$ *terminates with* $L(v) = \{v\}$ *for all vertices* v *of* $\mathbb{T}$*.*

Proof. 1 ⇒ 2: Let $\mathbb{T}$ be a core. We assume there is another homomorphism $f \in \mathrm{End}(\mathbb{T})$ with $f \neq id$.

Let v and w be two vertices of $\mathbb{T}$. Note that every unique shortest path from v to w maps to the unique shortest path from $f(v)$ to $f(w)$ because f is an automorphism of a tree. It follows that there must be a leaf u on which f is not the identity since otherwise $f = id$. We consider $p = (u_1, \ldots, u_k)$ to be the unique path from u to $f(u)$ which maps to the unique path p' from $f(u)$ to $f(f(u))$. We claim that there has to be a vertex v on p for which $f(v) = v$.

In the simple case we suppose that $f(f(u)) = u$. This implies $f(u_i) = u_{l-i}$ for $i \in \{0, 1, \ldots, l\}$. Since no double-edges are allowed, we conclude that $l = 2m$, which gives us $f(u_m) = u_m$.

For the general case, consider $\{f^k(u) \mid k \in \mathbb{N}\}$ to be of size $n \geq 3$. Since $u_l = f(u_0)$ there is a greatest $m \leq l$ such that $u_{l-i} = f(u_i)$ for every $i \in \{0, 1, \ldots, m\}$. As a consequence, there must be a path q of the form

$$(u_m, \ldots, u_{l-m-1}, f(u_m), \ldots, f(u_{l-m-1}), f^2(u_m), \ldots, f^{n-1}(u_{l-m-1}), f^n(u_m))$$

from u_m to $f^n(u_m) = u_m$ of length $n(l-2m)$. In the following, we suppose that $\mathrm{length}(q) \neq 0$.

Since $\mathbb{T}$ is a tree, there must be a sequence v_{i-1}, v_i, v_{i+1} in q with $v_{i-1} = v_{i+1}$. We also know that f is an automorphism from which it follows that $v_i = f^k(u_m)$ for some $k \in \mathbb{N}$ and therefore $v_{i-1} = f^{k-1}(u_{l-m-1})$ and $v_{i+1} = f^k(u_{m+1})$. Since $f^{n-k+1}(v_{i-1}) = f^{n-k+1}(v_{i+1})$, we get $u_{l-m-1} = f(u_{m+1})$. However, this means that we did not choose m maximal in the first place. It follows that $\mathrm{length}(q) = n(l-2m) = 0$. The latter equation can only be satisfied for $l = 2m$, and again we get $f(u_m) = u_m$.

Now let $\mathbb{T}_i$ be the component of $\mathbb{T} - v$ that contains u and. We then construct an endomorphism h of $\mathbb{T}$ by defining $h(w) = f(w)$ for each vertex $w \in \mathbb{T}_i$. For every other vertex w we define $h(w) = w$. It is easy to see that h is non-injective.

However, this means that $\mathbb{T}$ can not be a core, which means our assumption was wrong and $\mathrm{End}(\mathbb{T})$ cannot contain such a f but only id.

2 ⇒ 1: If $\mathrm{End}(\mathbb{T}) = \{id\}$, then the only homomorphism $h\colon \mathbb{T} \to \mathbb{T}$ is id, which is an automorphism. Hence, $\mathbb{T}$ must be a core.

2 ⇒ 3: Let us assume that $\mathrm{End}(\mathbb{T}) = \{id\}$. To prove that $\mathrm{AC}_{\mathbb{T}}(\mathbb{T})$ terminates with $L(v) = \{v\}$ for all vertices v of $\mathbb{T}$ we use a modified version of the proof of implication 4 ⇒ 2 of Theorem 2.7 in the script for the lecture *Graph Homomorphisms and Universal Algebra* [1].

Initially, we run $\mathrm{AC}_{\mathbb{T}}(\mathbb{T})$. Now let v' be an arbitrary vertex in $\mathbb{T}$. By choosing a vertex u from the list of each vertex v we can construct a sequence $f_0, \ldots, f_n$ for $n = |V(\mathbb{T})|$ where f_i is a homomorphism from the subgraph of $\mathbb{T}$ induced by the vertices at distance at most i to v', and f_{i+1} is an extension of f_i for all $1 \leq i \leq n$. We start by defining f_0 to map v' to an arbitrary vertex $u' \in L(v')$.

Suppose inductively that we have already defined f_i. Let w be a vertex at distance $i+1$ from v' in $\mathbb{T}$. Since $\mathbb{T}$ is an orientation of a tree, there is a unique $w' \in V(\mathbb{T})$ of distance i from v' in $\mathbb{T}$ such that $(w, w') \in E(\mathbb{T})$ or $(w', w) \in E(\mathbb{T})$. Note that $x = f_i(w')$ is already defined. In case that $(w', w) \in E(\mathbb{T})$, there must be a vertex y in $L(w)$ such that $(x, y) \in E(\mathbb{T})$ because otherwise the arc-consistency procedure would have removed x from $L(w')$. We then set $f_{i+1}(w) = y$. In case that $(w, w') \in E(\mathbb{T})$ we can proceed analogously. By construction, the mapping f_n is an endomorphism of $\mathbb{T}$.

Knowing that id is the only endomorphism of $\mathbb{T}$, we get $v' = f_n(v') = u'$. Since v' and u' both were chosen arbitrarily, it follows that $L(v) = \{v\}$ for every vertex $v \in \mathbb{T}$.

$3 \Rightarrow 2$: Let us assume $\mathrm{AC}_{\mathbb{T}}(\mathbb{T})$ derives $L(v) = \{v\}$ for all vertices v of $\mathbb{T}$. Note that $h(v) \in L(v)$ for any homomorphism h and for all $v \in \mathbb{T}$. Thus, we know there cannot be another homomorphism h for which $h(v) \neq v$. Hence, $\mathrm{End}(\mathbb{T}) = \{id\}$. □

From now on we say that AC *derives id* if AC derives $L(v) = \{v\}$ for every vertex v.

We note that to answer Question 2.1 for a triad $\mathbb{T}$ we can simply run $\mathrm{AC}_{\mathbb{T}}(\mathbb{T})$ and see if it derives *id*. If this is the case, then we know that $\mathbb{T}$ is a core.

However, this approach can be inefficient as it is possible to infer knowledge about the triad by considering its individual arms. Below, we will formulate another criterion that will help us answer Question 2.1 differently. For this we need to introduce the notion of a partial triad.

A *partial triad* is a tuple of the form p_1, p_2, p_3 where $p_i \in \{\{0,1\}^n \mid n \in \mathbb{N}\}$ for $i \in \{1,2,3\}$ and $p_i = \varepsilon$ for at least one i. Usually we write down only the words that are not empty, e.g. $1011, 110$. Each partial triad θ can be completed to form a triad $\mathbb{T}$ by adding arms to it. In this case, we say that $\mathbb{T}$ was *derived* from θ. Conversely, we say that a triad $\mathbb{T}$ is *reduced* to a partial triad θ by removing arms from $\mathbb{T}$.

Note that adding arms to a partial triad puts further restrictions upon its root vertex. Therefore, running $\mathrm{AC}_{\theta}(\theta)$ on a partial triad θ is insufficient for making a statement about the homomorphic equivalence of a triad derived from θ. E.g., consider the arm 1000 on which AC does not derive *id*. Yet, $1000, 11, 0$ is still a core.

We can avoid such a phenomenon by defining $\mathrm{ACR}_{\mathbb{T}}$ as the modification of $\mathrm{AC}_{\mathbb{T}}$ that solves the precoloured CSP of $\mathbb{T}$ where we colour the root r by setting $L(r)$ to $\{r\}$. A *rooted core* then names a partial triad θ for which $\mathrm{ACR}_{\theta}(\theta)$ derives *id*. Now we can formulate the following lemma in answer to Question 2.1.

Lemma 2.2. *Let $\mathbb{T}$ be a triad. If $\mathbb{T}$ can be reduced to a partial triad θ such that θ is not a rooted core, then $\mathbb{T}$ cannot be a core.*

Proof. Let h be a automorphism of θ such that $h(r) = r$ for the root r of θ. We then construct an endomorphism h' of $\mathbb{T}$ by defining $h'(v) = v$ for every vertex $v \in V(T) \setminus V(\theta)$. For any other vertex u we take $h'(u) = h(u)$. It is easy to see that h' is non-injective. □

2.1 Algorithm

We are now in position to describe the algorithm of the enumeration procedure. The idea is that core triads are build up from individual arms and that for every preliminary partial triad we check whether it is a rooted core. If this is not the case, we can exclude it from the further building process. The pseudo-code of the entire algorithm is displayed in Algorithm 2.

Now that we have presented the concept of the enumeration procedure, we still can make further improvements in terms of runtime. These are as follows.

Algorithm 2: Algorithm for enumerating core triads

Input: an unsigned integer n
Output: a list of all core triads with at most n vertices

```
// Finding a list of arms
```
armlist $\longleftarrow$ [ ]
foreach *arm* a *with* length$(a) \leq n$ **do**
  if $\mathrm{ACR}_a(a)$ *does not derive id* **then**
    Put a in armlist

```
// Assembling the arms to core triads
```
triadlist $\longleftarrow$ [ ]
forall p_1, p_2 *in armlist with combined at most* n *vertices* **do**
  Let $\theta = p_1, p_2$ be a partial triad
  if $\mathrm{ACR}_\theta(\theta)$ *does not derive id* **then**
    Drop θ and cache the two arms
forall p_1, p_2, p_3 *in armlist with combined at most* n *vertices* **do**
  Let $\mathbb{T} = p_1, p_2, p_3$ be a triad
  if $\mathbb{T}$ *contains a pair of cached arms* **then**
    Drop $\mathbb{T}$ and continue
  if $\mathrm{AC}_{\mathbb{T}}(\mathbb{T})$ *derives id* **then**
    Put $\mathbb{T}$ in triadlist
return *triadlist*

Lemma 2.3. *Let $H = (V, E)$ be a graph and let f be a polymorphism of H. Then f is also a polymorphism of H^T.*

Proof. Let $H = (V, E)$ be a graph and let $H^T = (V, E^T)$ be the transpose of H. A mapping $f\colon V(H)^k \to V(H)$ is a polymorphism of H if and only if $(f(u_1, \ldots, u_k), f(v_1, \ldots, v_k)) \in E(H)$ whenever $(u_1, v_1), \ldots, (u_k, v_k)$ are in $E(H)$. Now let f be a polymorphism of H. Since $(f(v_1, \ldots, v_k), f(u_1, \ldots, u_k)) \in E(H^T)$ whenever $(v_1, u_1), \ldots, (v_k, u_k)$ are in $E(H^T)$, f is also a polymorphism of H^T. □

This lets us reduce the number of triads to look at by half. This is done by considering only those triads whose root vertex has at least two outgoing edges.

We can achieve further improvement regarding the generation of arms based on the fact that we're only interested in those that are rooted cores. In a naive approach, one would generate all arms up to length n and only then check whether they are rooted cores. However, we use a sophisticated recursive procedure in which arms of length n are formed from arms of length $n-1$ by appending a vertex to the root. This allows us to consider only those arms of length $n-1$ that have been proven to be rooted cores.

Another aspect to consider is the implementation of the arc-consistency procedure. In the simplest implementation of the algorithm, called AC-1, the inner loop of the algorithm goes over all edges of the graph, in which case the running time of the algorithm is quadratic in the size of G. In what follows, we describe

one of the simplest implementations of the procedure, called AC-3, which is due to Mackworth [5], and has a worst-case running time that is linear in the size of G. While earlier AC algorithms were often inefficient, successors to the AC-3 are usually a lot harder to implement, which is why we choose it as good general-purpose implementation of AC.

The idea of AC-3 is to maintain a *worklist* which contains a list of arcs (x_0, x_1) of G that might help remove a value from $L(x_0)$ or $L(x_1)$. Whenever we remove a value from a list $L(x)$, we add all arcs that are in G incident to x. Note that then any arc in G might be added at most $2|V(H)|$ many times to the worklist, which is a constant in the size of G. Hence, we iterate the while loop of the implementation for at most a linear number of times. Altogether, the running time is linear in the size of G as well.

In the following table we considered only CSP-instances of a triad H where $G = H$. It shows the running time of AC-1 and AC-3 for a series of linear input sizes. Both algorithms have been implemented in Rust and ran on a system with 1x AMD Ryzen 5 PRO 4650U (6 cores) @ 1.397GHz under Linux. The values were averaged over 100 iterations.

vertex number	12	36	39	48
AC-1	38 µs	789 µs	998 µs	1818 µs
AC-3	45 µs	797 µs	999 µs	1659 µs

As we can see, AC-3 does not improve the running time for small input sizes because the linear complexity is achieved by frequent memory access whose significance becomes smaller as the input size increases. However, we are only interested in generating triads that have at most 22 vertices, so we choose AC-1 as our preferred arc-consistency implementation for Algorithm 2.

3 Finding Polymorphisms

In the previous section we presented an algorithm that enumerates triads up to a certain size; now we dedicate this section to finding polymorphisms, so that we can eventually verify Proposition 2.1. We start with the following Proposition that follows from the Theorem 1.1 of [7].

Proposition 3.1. *If an orientation of a tree has a binary polymorphism f satisfying $f(x, y) = f(y, x)$ for all $x, y \in V$, then it has a Siggers polymorphism.*

Whether the converse implication also holds true is still an open question of great interest that was already posed by Jana Fischer [4] in her Diploma thesis. Based on the proposition we can first check whether a triad has a commutative polymorphism. If this is the case, we do not need to perform the much more costly procedure of checking for a Siggers polymorphism. However, if we find a triad without a commutative polymorphism, we still need to check the existence of a Siggers polymorphism.

3.1 Algorithm

Up to this point, we were not concerned about the case where arc consistency does not solve our CSP-instance since we were looking exclusively for endomorphisms (and in this case there is always at least *id*). Now that we are looking

Algorithm 3: Pruning search

Input: two graphs G and H
Data: for every $x_i \in V(G)$ a list $L(x_i)$ of all vertices of H where $i \in \{1, 2, \ldots, |V(G)|\}$
Output: is there a homomorphism from G to H

```
if AC_H(G) rejects then
    reject
return search(L', G, H, x_1)
Fn search(L, G, H, x_i)
    if i > |V(G)| then
        accept
    foreach u ∈ L(x_i) do
        foreach y ∈ V(G) do
            Let L'(y) be a copy of L(y)
        L'(x_i) ⟵ {u}
        Run AC_H(G) with the lists L'
        if AC_H(G) accepts then
            return search(L', G, H, x_{i+1})
    reject
```

for polymorphisms it is possible that AC does not solve our CSP-instance; in these cases we need to perform a search. But even then the arc-consistency procedure is useful for *pruning the search space* in exhaustive approaches. The algorithm we present does exactly this by using arc consistency as a subroutine as described below.

Suppose that H is such that AC does not solve CSP(H). Initially, we run AC_H on the input graph G. If AC_H derives the empty list, we reject, otherwise we pick some vertex $x \in V(G)$ and set $L(x)$ to $\{v\}$ for some $v \in L(x)$. Next, we run AC again and proceed recursively with the resulting lists. However, if AC_H now derives the empty list, we backtrack and remove v from $L(x)$. Finally, if AC did not derive the empty list and all lists are singleton sets $\{u\}$, the map that sends x to u is a homomorphism from G to H. The pseudo-code of the entire search-procedure can be found in Algorithm 3.

An algorithm that checks the existence of a commutative polymorphism for a given graph H uses this pruning search procedure in the following way.

We start from the second power H^2, and then contract each vertex (u, v) with (v, u). Let G be the resulting graph.

Observation 3.1. *There exists a homomorphism from G to H if and only if H has a commutative polymorphism.*

Therefore, we can take G and H as an input to the pruning search procedure, which will decide whether a homomorphism from G to H exists.

Note that the procedure described above can be generalised further to check the existence of any arbitrary polymorphism for any graph H. It is only necessary to adjust the power of H and the appropriate vertices that must be

contracted. This way we can also verify the existence of a Siggers polymorphism. For the rest of the thesis we will refer to this procedure as *polymorphism search*.

Again, we measure the running time of AC-1 and AC-3; this time over a range of large input variables as typically encountered when checking polymorphisms. The CSP-instance has a constant size independent of the number of vertices of the input graph. All other conditions are identical to those of the previous measurement in Section 2.

vertex number	16^2	19^2	26^2
AC-1	2.356 ms	3.663 ms	7.765 ms
AC-3	1.837 ms	3.018 ms	4.064 ms

This time, the running time improvement of AC-3 over AC-1 is striking to observe, as well as the linear running time of AC-3. Since these input sizes correspond to those when searching for polymorphisms, we choose AC-3 as our preferred arc-consistency implementation for the polymorphism search procedure.

3.2 Results

We let Algorithm 2 generate all core triads with up to 24 vertices. Then, for each of these triads, we used the polymorphism search procedure from Section 3.1 to check the existence of commutative and Siggers polymorphisms. The table below lists the number of core triads for a given number of vertices, as well as the number of those without a commutative polymorphism or without a Siggers polymorphism.

vertex number	core triads	no commutative	no Siggers
21	132,796	0	0
22	294,046	2	2
23	652,772	9	9
24	1,431,813	37	37

The correspondence between the numbers regarding the two polymorphism kinds has further strengthened our conviction that the converse implication of Proposition 3.1 holds true.

Also, our program did not find a triad with less than 22 vertices that has no commutative polymorphism. Combined with Proposition 3.1 this gives us the following lemma.

Lemma 3.1. *Every triad with less than 22 vertices has a Siggers polymorphism.*

Moreover, we used the polymorphism search procedure to verify the following lemma that refers to a triad with 22 vertices.

Lemma 3.2. *The triad* $10110000, 1001111, 010111$ *has no Siggers polymorphism.*

An illustration of the triad is displayed in Figure 1. Lemma 3.1 together with Lemma 3.2 finally verifies Proposition 2.1.

As already seen in the previous table we were able to find another smallest triad with 22 vertices without a Siggers polymorphism, which is given by the following lemma.

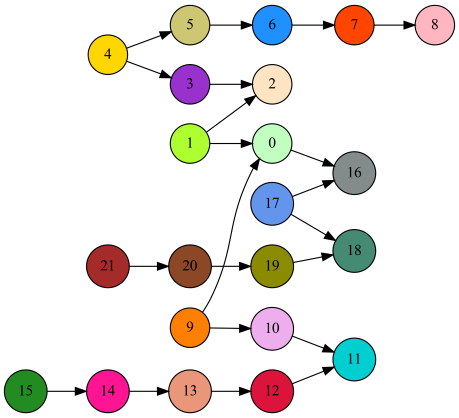

Figure 1: A smallest triad without a Siggers polymorphism

Lemma 3.3. *The triad* $10110000, 0101111, 100111$ *has no Siggers polymorphism.*

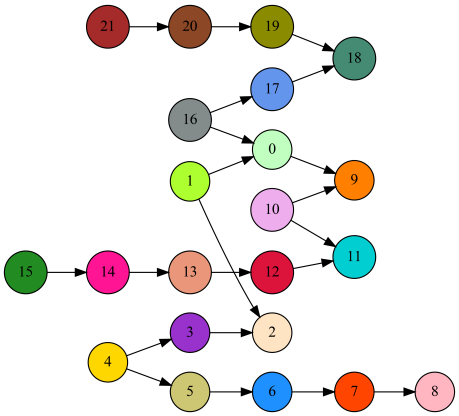

Figure 2: Another smallest triad without a Siggers polymorphism

An illustration of the triad is displayed in Figure 2. Our concern is now to understand the structural properties of this triad that prevent the existence of a Siggers polymorphism. However, proving the non-existence of a Siggers polymorphism by hand is too costly, so we will only prove the weaker result, which reads as follows.

Lemma 3.4. *The triad* $10110000, 0101111, 100111$ *has no commutative polymorphism.*

For the proof we need the following lemma.

Lemma 3.5. *Let* $\mathbb{T}$ *be a core tree and let* h *be a homomorphism from* $\mathbb{T}^k$ *to* $\mathbb{T}$*. Then* $h(v, v, \ldots, v) = v$*.*

Proof. Let $\mathbb{T}$ be a core tree. By definition of the direct product, there is an edge between two vertices $(v, v, \ldots, v)$ and $(w, w, \ldots, w)$ of $\mathbb{T}^k$, if and only if $(v, w) \in E(\mathbb{T})$. Now let h be a homomorphism from $\mathbb{T}^k$ to $\mathbb{T}$. We then construct an endomorphism h' of $\mathbb{T}$ by defining $h'(v) = h(v, v, \ldots, v)$. By Lemma 2.1 we know that h' must be the identity, from which it follows that h must map $(v, v, \ldots, v)$ to v. □

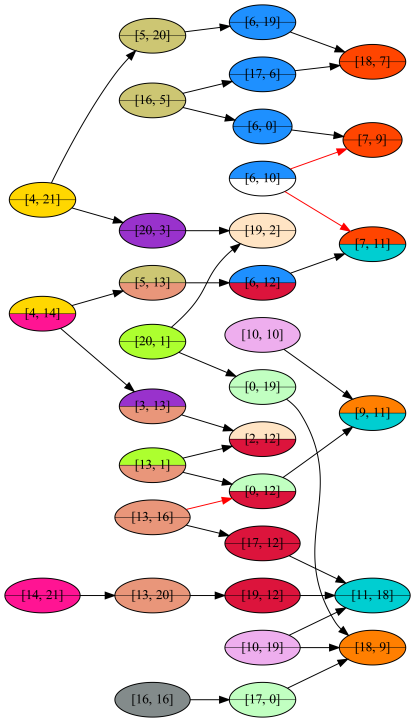

Figure 3: Illustration of the graph $\mathbb{S}'$ from the proof of Lemma 3.4

Proof of Lemma 3.4. As before, we start from the second power $\mathbb{T}^2$, and then contract each vertex (u, v) with (v, u). Let $\mathbb{S}$ be the resulting graph. We will simplify the proof by considering only a subgraph $\mathbb{S}'$ of $\mathbb{S}$ such that $\mathbb{S}'$ has no homomorphism to $\mathbb{T}$, if $\mathbb{S}$ has no homomorphism to $\mathbb{T}$ and every real subgraph of $\mathbb{S}'$ has a homomorphism to $\mathbb{T}$. Thus, all we have to show is that $\mathbb{S}'$ has no homomorphism. An illustration of such a subgraph $\mathbb{S}'$ is displayed in Figure 3.

We start at $[10, 10]$, which by Lemma 3.5 must be mapped to 10. This enforces only two possible mappings on $[9, 11]$, either 9 or 11. In fact, either of the two choices will eventually lead to a contradiction, so we continue by making a case distinction.

In Figure 3 the first case, $[9, 11] \mapsto 9$ is visualised with the top half colour of each vertex whereas the second case, $[9, 11] \mapsto 11$, is represented by the bottom half colour. Each colour represents a mapping to the vertices of $\mathbb{T}$ with that colour. The red arrows indicate a contradiction in finding a homomorphism from $\mathbb{S}'$ to $\mathbb{T}$.

Case 1. *In this case, we map* $[9, 11]$ *to* 9*. It is easy to see that mapping* $[0, 12]$ *to* 10 *is not possible, which means we must map* $[0, 12]$ *to* 0*. For our further traversal we may choose either of the two adjacent vertices* $[13, 1]$ *or* $[13, 16]$*, of which we choose* $[13, 1]$*. Looking ahead, we notice that to get to* $[4, 14]$*, we have to map* $[13, 1]$*,* $[2, 12]$*,* $[3, 13]$ *and* $[4, 14]$ *to* 1*,* 2*,* 3 *and* 4*, respectively. Continuing from* $[4, 14]$*, we see that there have to be three successive forward edges, which forces* $[7, 11]$ *to be mapped to* 7*. Now the mapping for each of the following vertices is straightforward until we arrive at* $[4, 21] \mapsto 4$*. We notice that the path from* $[4, 21]$ *to* $[16, 16]$ *has the exact same orientation as the path from* 4 *to* 16*. Therefore, we must map the former to the latter because we know that* $[16, 16] \mapsto 16$*. In particular, we map* $[18, 9]$ *to* 9*. As a consequence,* $[11, 18]$ *must be mapped to* 11 *as the latter is the only vertex from where we can traverse three consecutive backward edges. It follows that* $[13, 6] \mapsto 13$ *and we finally arrive at a contradiction since there is no edge between* 13 *and* 0*.*

Case 2. *This is the case where we map* $[9, 11]$ *to* 11*. Again,* $[0, 12] \mapsto 10$ *is not possible because* 10 *has no incoming edge, therefore we map* $[0, 12]$ *to* 12*. For our further traversal we may choose either of the two adjacent vertices* $[13, 1]$ *and* $[13, 16]$*. Traversing in direction of* $[13, 1]$ *is straightforward, and we stop at* $[7, 11] \mapsto 11$*. Now we go back to* $[0, 12]$ *and start traversing in the other direction. It is easy to see that* $[11, 18] \mapsto 11$*. Next, we notice that the path from* $[11, 18]$ *to* $[16, 16]$ *has the exact same orientation as the path from* 11 *to* 16*. That leaves us no choice but to map the former to the latter since we know that* $[16, 16] \mapsto 16$*. In particular, we map* $[18, 9]$ *to* 9*. It follows that* $[20, 1] \mapsto 1$ *because otherwise we would not be able to reach* $[4, 21]$*. After mapping* $[4, 21]$ *to* 4*, the following mappings are straightforward until we map* $[7, 9]$ *to* 7*. Having mapped* $[7, 11]$ *to* 11 *earlier, we see that for* $[6, 10]$ *there is no mapping such that* 9 *and* 11 *are adjacent.* □

Combining the results of Lemma 4-6, we end up with the following corollary, which concludes our work.

Corollary 3.1. *The smallest triads with an NP-hard CSP (unless P=NP) are* $10110000, 1001111, 010111$ *and* $10110000, 0101111, 100111$*.*

4 Conclusion

We have presented an algorithm that enumerates all core triads up to a fixed number of vertices. It is reasonable to assume that the generalisation to generate core trees requires only little adjustments to the algorithm that we used to generate core triads. This can be a task for further research.

Furthermore, we showed a successful application of the arc consistency algorithm to check whether a given triad has either a commutative polymorphism or a Siggers polymorphism. In this way, we were able to confirm Bulin's claim that all triads with less than 22 vertices have a Siggers polymorphism.

Lastly, we provided a human-readable proof for the non-existence of a commutative polymorphism for a new smallest triad without a Siggers polymorphism.

5 Acknowledgments

I gratefully acknowledge the support of Florian Starke and thank him for making useful comments and for the supervision throughout the entire project. Also, I would like to thank Michael Sippel for providing a code base that served as a starting point for the program. The research was supported by the Center for Information Services and High Performance Computing [Zentrum für Informationsdienste und Hochleistungsrechnen (ZIH)] TU Dresden by providing its facilities for high throughput calculations.

References

[1] M. Bodirsky. *Graph Homomorphsims and Universal Algebra, Script.* 2020.

[2] A. A. Bulatov. "A Dichotomy Theorem for Nonuniform CSPs". In: *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS).* 2017, pp. 319–330.

[3] Jakub Bulín. "On the complexity of H-coloring for special oriented trees". In: *European Journal of Combinatorics* 69 (2018), pp. 54–75. ISSN: 0195-6698.

[4] J. Fischer. *CSPs of orientations of trees. Master thesis, TU Dresden.* 2015.

[5] Alan K. Mackworth. "Consistency in networks of relations". In: *Artificial Intelligence* 8.1 (1977), pp. 99–118. ISSN: 0004-3702.

[6] Ugo Montanari. "Networks of constraints: Fundamental properties and applications to picture processing". In: *Information Sciences* 7 (1974), pp. 95–132. ISSN: 0020-0255.

[7] M. H. Siggers. "A strong Mal'cev condition for varieties omitting the unary type". In: *Algebra Universalis* (2010).

[8] Dmitriy Zhuk. "A Proof of the CSP Dichotomy Conjecture". In: *J. ACM* 67.5 (Aug. 2020). ISSN: 0004-5411.