Binary Commutative Polymorphisms of Core Triads

Michael Wernthaler

March 3, 2021

Contents

Abstract

It has been known for a while that for a given digraph H the complexity of constraint satisfaction problem for H also known as the H-colouring problem only depends on the set of polymorphisms of H. From recent results it follows that the H-colouring problem is in P if and only if H has a so-called (4-ary) Siggers polymorphism. In this paper, 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 size.

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 [5] 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.

$$01001111, 0110000, 101000$$

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 an orientation of a tree has a Siggers polymorphism if and only if it has a binary polymorphism f satisfying $f(x, y) = f(y, x)$ for all $x, y \in V$.

This paper seeks to confirm the conjecture by Jakub Bulin, that the triad

$01001111, 0110000, 101000$ 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 binary-commutative polymorphism for our new triad, that isn't based on running a computer program, but understandable to humans. In this way, we hope to contribute to understanding the hidden relations between polymorphisms and the complexity of CSP for triads.

The paper is organised as follows. First, we formally introduce the preliminary notions and notations necessary for reading the paper. 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 conjecture. Section 4 concludes this work.

1 Preliminaries

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 $E = E(H)$ its *edges*. The graph G is called *finite* or *infinite* depending on whether $V(G)$ is finite or infinite. However, since this paper 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 $(e_1, ..., e_{k-1})$ of edges of G such that $e_i = (u_i, u_{i+1}) \in E(G)$, 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 $(u_0, ..., u_{k-1})$ of pairwise distinct vertices and a sequence of edges $(e_0, ..., e_{k-1})$ is called a *cycle* (of G) if $(u_0, ..., u_{k-1}, u_0)$ and $(e_0, ..., e_{k-1})$ form a path. The *length* of a circle 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 u_1 to u_k.

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 $H = (V, E)$ be another graph. A *homomorphism* from G to H is a mapping $h\colon V(G) \to V(H)$ such that $(h(u), h(v)) \in E(H)$ 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 finite 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 digraph 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 $(f(u_1, ..., u_k), f(v_1, ..., v_k)) \in E(H)$ whenever $(u_1, v_1), ..., (u_k, v_k)$ are arcs in $E(H)$.

1.5 The H-colouring Problem

When does a given digraph G homomorphically map to a digraph H? For every digraph H, this question defines a computational problem, called the *H-colouring problem*, also denoted CSP(H). The input of this problem consists of a finite digraph 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 finite digraph 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 [4], [3] and is often referred to as the *consistency check procedure*.

Let H be a finite digraph, 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.

Algorithm 1: $AC_{\mathbb{T}}$ ($\mathbb{T}$ is a triad)

Input: Digraph $\mathbb{G}$, initial lists $L\colon G \mapsto P(T)$
Data: For every $x \in V(G)$ a list $L(x)$ of elements of $V(H)$
Output: Is there a homomorphism $h\colon \mathbb{G} \mapsto \mathbb{T}$ such that $h(v) \in L(v)$ for all $v \in G$

```
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)
            end
        end
        if L(x) is emtpy then
            reject
        end
    end
until No list changes
accept
```

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.

However, there are digraphs 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 paper will be to prove the following conjecture due to Jakub Bulin, that will eventually be verified in Section 3.

Conjecture 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, i.e. not homomorphically equivalent to smaller triads. Thus, we pose the following question.

Question 1. *When is a triad* $\mathbb{T}$ *homomorphically equivalent to a smaller triad?*

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

Lemma 1. *Let* T *be a finite tree. The following are equivalent.*

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

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

Let v and w be two vertices of T. Note, that every unique shortest path from v to w maps to the unique shortest path from $f(v)$ to $f(w)$, since f is an automorphism of a tree. It follows, that there must be a leaf u on which f is not the identity, because otherwise $f = id$.

We consider $p = (u_1, ..., u_k)$ 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 $i \in \{0, 1, ..., l\}$. Since no double-edges are allowed, we conclude that $l = 2m$, which gives us $f(u_m) = u_m$.

For the general case, we consider $\{f^k(u) \mid k \in \mathbb{N}\}$ to be of size $n \geq 3$. Because of $f(u_0) = u_l$ there is a greatest $m \leq l$ such that $f(u_i) = u_{l-i}$, for every $i \in \{0, 1, ..., m\}$. As a consequence, there must be a path $p = (v_1 ... v_h)$ from u_m to $f^n(u_m) = u_m$ of length $n(l - 2m)$. In the following we suppose that $length(p) \neq 0$. Since T is a tree, it follows that there must be a $v_{h2} = f^j(u)$ (*TODO* corollary that $f^j(u)$ is a leaf?) such that $v_{h2-i} = v_{h2+i}$ (*TODO* do I have to define i?). However, this implies that $f^{j-1}(u) = f^{j+1}(u)$, which obviously can't be the case. It follows that $length(p) = n(l-2m) = 0$. The latter equation can only be satisfied for $l = 2m$, and again we get $f(u_m) = u_m$.

Now let T_i and T_j with $i \neq j$ be the components of $T - v$ that contain u and $f(u)$, respectively. We then construct an endomorphism h of T by defining $h(v) = f(v)$ for each vertex $v \in T_i$. For every other component we define h as id. It's easy to see that h is non-injective.

However, this means that T can't be a core, which means our assumption was wrong and $End(T)$ cannot contain such a f, but only id.

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

2. $\Rightarrow$ 3: Let's assume that $End(T) = \{id\}$. To prove that $AC_T(T)$ terminates with $L(v) = \{v\}$ for all vertices v of T we use a modified version of the proof of implication 4 $\Rightarrow$ 2 of Theorem 2.7 in the script of [1].

Initially, we run $AC_T(T)$. Now let v' be an arbitrary vertex in T. By choosing a vertex u from the list of each vertex v we can construct a sequence $f_0, ..., f_n$ for $n = |V(T)|$, where f_i is a homomorphism from the subgraph of 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 T. Since T is an orientation of a tree, there

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

Knowing that id is the only endomorphism of 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 $v \in T$.

3. $\Rightarrow$ 2: Let's assume $AC_T(T)$ derives $L(v) = \{v\}$ for all vertices v of T. It's obvious, that always $\{id\} \subseteq End(T)$. However, we know there can't be another homomorphism h for which $h(v) \neq v$, since $h(v) \in L(v)$ for all $v \in T$. Hence, $End(T) = \{id\}$. ■

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 1 we can simply run $AC_{\mathbb{T}}(\mathbb{T})$ 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 we can infer knowledge about the triad by considering its individual arms. For instance, there's the case, in which $\mathbb{T}$ has two identical arms. We can easily see, that $\mathbb{T}$ is not a core without the need to apply the AC-procedure to the entire triad. Below, we will formulate another criterion that will help us answer Question 1 in a different way. For this we need to introduce the notion of a *partial triad*.

Definition 2.1. *A* partial triad *is a triad 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 $\mathbb{T}$ can be *reduced* to a partial triad θ by removing arms from $\mathbb{T}$. Let $\theta_1 = p_1, p_2, p_3$ and $\theta_2 = q_1, q_2, q_3$ be two partial triads such that the total number of non-empty words is ≤ 3 **TODO**.

Note, that adding arms to a partial triad puts further restrictions upon its root vertex. Therefore, running $AC_\theta(\theta)$ 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 this phenomena by defining $ACR_{\mathbb{T}}$ as the modification of $AC_{\mathbb{T}}$, that solves the pre-coloured CSP of $\mathbb{T}$, where we colour the root r by setting $L(r)$ to $\{r\}$. A *rooted core* (RC) then names a partial triad θ for which $ACR_\theta(\theta)$ derives id. Now we can formulate the following lemma in answer to Question 1.

Lemma 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's 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 for generating core triads is displayed in Algorithm 2. Note, that the algorithm seen there generates triads up to a maximal arm length m. However, to find polymorphisms of triads up to n vertices, we must apply subtle changes to Algorithm 2, which are as follows.

- At the beginning, we generate the list of arms so that it contains only those whose number of vertices is $\leq n$.
- For any partial triad with two arms we not only check if the resulting partial triad is a (rooted) core, but also if its number of vertices is $\leq n$.
- For any triad we not only check if the resulting triad is a core, but also if its number of vertices is $\leq n$.

Algorithm 2: Algorithm for finding core triads

Input: An unsigned integer m
Output: A list of all core triads whose arms each have a length $\leq m$

```
// Finding a list of RCAs
armlist ⟵ [ ]
foreach arm a with length(a) ≤ m do
    if ACR_a(a) does not derive id then
        Put a in armlist

// Assembling the RCAs to core triads
triadlist ⟵ [ ]
forall p_1, p_2 in armlist do
    if ACR_{p_1p_2}(p_1p_2) does not derive id
     then
        Drop the pair and cache the two arms
foreach p_1, p_2, p_3 in armlist do
    Let 𝕋 = p_1, p_2, p_3 be a triad
    if 𝕋 contains a pair of cached arms then
        Drop 𝕋 and continue
    if AC_𝕋(𝕋) derives id then
        Put 𝕋 in triadlist
return triadlist
```

Since these modifications are slight, we do not notate a separate algorithm.

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

Lemma 3. *Let $H = (V, E)$ be a digraph and let f be a polymorphism of H. Then f is also a polymorphism of H^T.*

Proof: Let $H = (V, E)$ be a digraph 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 $(f(u_1, ..., u_k), f(v_1, ..., v_k)) \in E(H)$ whenever $(u_1, v_1), ..., (u_k, v_k)$ are arcs in $E(H)$. Now let f be a polymorphism of H. Since $(f(v_1, ..., v_k), f(u_1, ..., u_k)) \in E(H^T)$ whenever $(v_1, u_1), ..., (v_k, u_k)$ 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. Of each pair of triads, whose transpose is equal to the other triad, our algorithm considers the one, whose first edge of the first arm is a backward edge.

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 m 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 the following we describe one of the simplest implementations of the procedure, called AC-3, which is due to Mackworth [3], 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 generally much more difficult to implement **TODO**.

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 $2\mathsf{V}(\mathsf{H})$ 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.

vertices	12	36	39	48
AC-1	38.3 us	797.4 us	998.3 us	1.818 ms
AC-3	45.7 us	789.8 us	999.3 us	1.659 ms

As we can see, AC-3 does not improve the running time for small input sizes, since the linear

complexity is achieved by frequent memory access, whose significance becomes smaller as the input size increases. However, we're 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 last 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 prove Conjecture 1. We start with the following Theorem by Jakub Bulin.

An orientation of a tree has a Siggers polymorphism if it has a binary polymorphism f satisfying $f(x, y) = f(y, x)$ for all $x, y \in V$.

(**TODO** can I cite this?) Bulin conjectures that the converse implication also holds true. From this it follows that we can restrict the search for triads without a Siggers polymorphism to those that have no binary-commutative polymorphism. However, if we find such a triad, we need to check the existence of a Siggers polymorphism.

3.1 Algorithm

Up to this point, we weren't 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's always at least id). Now that we are looking 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 3.1.

Algorithm 3: Pruning search

Input: two graphs G and H
Output: is there a homomorphism from G to H

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

The algorithm that checks the existence of a binary-commutative polymorphism for a given graph H uses this pruning search procedure as a subroutine as follows.

We start from the second power H^2, and then contract each vertex (u, v) with (v, u). Let G be the resulting graph. Note that there exists a homomorphism from G to H if and only if H has a binary-commutative polymorphism. Therefore, we can take G and H as an input to algorithm which will decide whether a homomorphism (and consequently a polymorphism) 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 arity of the potency of H and the appropriate vertices that must be contracted. This way we can also verify the existence of a Siggers polymorphism.

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.

vertices	196	361	576
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 in these cases.

3.2 Results

Finally, we are in position to verify Bulin's conjecture. For this purpose we let Algorithm 2 generate all triads up to the number of vertices 21 and then check for polymorphisms of each triad with the help of the procedure described. As expected, our program does not find a triad without a Siggers polymorphism, which proves Conjecture 1.

More interestingly, our program was able to confirm the following proposition.

Proposition 1.
The triad $\mathbb{T} = 10110000, 0101111, 100111$ *has no Siggers polymorphism.*

While until now only one smallest triad without a Siggers polymorphism was known, we have now found another triad of the same size, that is different from the one found by Bulin. A rendering of the triad is displayed in Figure 1.

Unfortunately, running the arc-consistency procedure does not give us the desired depth of understanding of the structural properties of $\mathbb{T}$ that prevent (**TODO** is "prevent" the correct term?) the existence of Siggers polymorphism. Also, the complexity of proving the absence of a Siggers polymorphism by hand is too great, so in the following we stick with a proof for a binary-commutative polymorphism.

For the proof we need the following lemma.

Lemma 4. *Let* T *be a core tree and let* h *be a homomorphism from* T^k *to* T*. Then* $h(v, v, ..., v) = v$*.*

Proof: Let T be a core tree. By definition of the direct product, there is an edge between two vertices $(v, v, ..., v)$ and $(w, w, ..., w)$ of T^k, if and only if $(v, w) \in E(T)$. Now let h be a homomorphism from T^k to T. We then construct an endomorphism h' of T by defining $h'(v) = h(v, v, ..., v)$. By Lemma 1 we know that h' must be the identity, from which it follows, that h must map $(v, v, ..., v)$ to v. ■

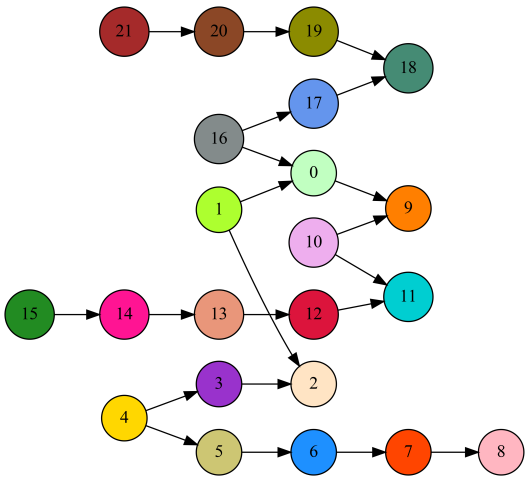

Figure 1: Another smallest triad without a Siggers polymorphism

Proof: [**TODO** Proof of Proposition 1] 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 subgraph. 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. A rendering of $\mathbb{S}'$ a minimal subgraph is displayed in Figure 2.

We start at $[10, 10]$, which by Lemma 4 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 2 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 color represents a mapping to the vertices of $\mathbb{T}$ with that color.

Case 1. *In this case we map* $[9, 11]$ *to* 9*. It's easy to see that mapping* $[0, 12]$ *to* 10 *isn't possible, since* 10 *has no incoming edge, which means we must map* $[0, 12]$ *to* 0*. For our further traversal we may choose either of the two adjacent vertices*

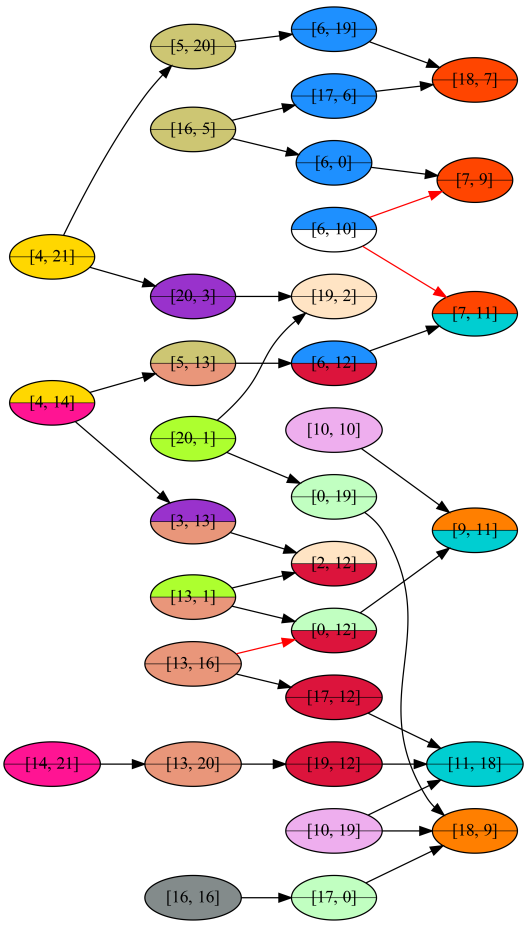

Figure 2: A minimal subgraph of the binary-product, without a homomorphism

$[13, 1]$ *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 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 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, since 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, in which we map* $[9, 11]$ *to* 11*. Again,* $[0, 12] \mapsto 10$ *isn't possible, since* 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's 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, *TODO* since we know that* $[16, 16] \mapsto 16$*. In particular, we map* $[18, 9]$ *to* 9*. It follows, that* $[20, 1] \mapsto 1$*, since otherwise we wouldn't 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.*

4 Conclusion

We have presented an algorithm that enumerates all core triads up to a fixed path-length. 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 a commutative polymorphism and provided a human-readable proof for the (**TODO** absence/non-existence) of the latter a certain triad.

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*. 2020.

[2] A. A. Bulatov. *A dichotomy theorem for nonuniform CSPs*. 2017.

[3] A. K. Mackworth. *Consistency in networks of relations*. 1977.

[4] U. Montanari. *Networks of constraints: Fundamental properties and applications to picture processing*. 1974.

[5] D. N. Zhuk. *A proof of CSP dichotomy conjecture*. 2017.