NOVI SAD J. MATH.
VOL. 44, NO. 2, 2014, 207-224

ON OPTIMAL STRONG MAL'CEV CONDITIONS FOR CONGRUENCE MEET–SEMIDISTRIBUTIVITY IN A LOCALLY FINITE VARIETY

Jelena Jovanović[1]

Abstract. In this paper we examine four–element and five–element digraphs for existence of certain polymorphisms that imply congruence meet–semidistributivity in a locally finite variety. The results presented here occurred as an integral part of my research for optimal strong Mal'cev conditions describing the property mentioned. Most of the programming needed to obtain these results was done in C programming language, but for some results we also used the model builder Paradox. The source codes and Paradox inputs will be presented here as well.

AMS Mathematics Subject Classification (2010): 08B05

Key words and phrases: Variety, Hobby-McKenzie types, Omitting types, Polymorphisms of digraphs

1. Introduction

The various conditions that are equivalent to congruence meet–semidistributivity in locally finite varieties of algebras have been examined in several papers and books so far – see [19], [5], [20], [21],[15], [9]. Congruence meet-semidistributive varieties proved to be a very general yet very well behaved class of varieties. This condition is, for example, equivalent to congruence neutrality ([20], [21]), or to having no covers of types **1** or **2** in congruence lattices of finite algebras in the variety (holds for locally finite varieties, [5]). It also implies the truth of Park's Conjecture, [12] (as proved by Ross Willard in [15]), and also Quackenbush's Conjecture, [13] (which holds trivially in congruence distributive case due to Jónsson's Lemma, [6], and is proved for the congruence meet-semidistributive case by Kearnes and Willard in [8]). Recently, the research in the Constraint Satisfaction Problem has shown that congruence meet-semidistributivity of the variety generated by the algebra of compatible operations is equivalent to the condition that the particular algorithm called 'local consistency checking' would faithfully solve the Constraint Satisfaction Problem. This property is called 'bounded width' and there is a lot of literature on the concept. This result, due to L. Barto and M. Kozik [2], is certainly among the strongest known partial results for the Dichotomy Conjecture, and probably the hardest to be proven so far.

Characterization of various semantical properties of all algebras in a variety and/or their congruence lattices by equivalent syntactical conditions was started by A. I. Mal'cev in [10]. Because of that, the properties which can be characterized in this way are called Mal'cev properties, and the corresponding syntactical conditions - Mal'cev conditions. Particularly, if a property can be described by a fixed number of term

[1] University of Belgrade, Faculty of Mathematics, email: jelena.jovanovic55@gmail.com

operations of fixed arities satisfying a fixed number of linear equations (like the original Mal'cev condition for congruence permutability), we call it a *strong* Mal'cev property. On the other hand, a usual Mal'cev property is equivalent to satisfying one strong Mal'cev condition (for some $n \in \omega$) from a given countable sequence of strong Mal'cev conditions (for every $n \in \omega$) in which each member implies the next one (meaning that the properties are increasingly more general), like in the case of Jónsson's condition for congruence distributivity, [6]. Obviously, strong Mal'cev conditions are preferable, if available, as then we can use the operations in a computer search, however, there exist properties which are Malcev properties, but are proved not to be strong Mal'cev properties. The condition most commonly used for congruence meet-semidistributivity of a variety (not necessarily locally finite), until recently, was the one proved by Willard, [15], but the research in the Constraint Satisfaction Problem has recently uncovered that the congruence meet-semidistributivity of a locally finite variety is a strong Mal'cev property. The best, i. e. syntactically strongest we know of, is due to M.Kozik, A.Krokhin, M.Valeriote and R.Willard [9]. We tried to see if it is also the best possible, see [17], and we identified a single candidate system which implies congruence meet-semidistributivity and is syntactically stronger and with fewer operations and/or of smaller arity than the condition proved in [9]. We proved that either this system we found is indeed equivalent to congruence meet-semidistributivity of a locally finite variety, or the condition from [9] is the best possible (proved in [17]). However, which of these alternatives is true we have not managed to ascertain yet. In this paper we examine the algebras of polymorphisms of four–element and five–element digraphs searching for a counterexample for our system.

2. Background

In this paper an *algebra* denotes a structure $\mathbf{A} = (A, F^{\mathbf{A}})$, where F is a signature, or language, consisting only of operation symbols of various arities, A is a nonempty set, and for each symbol $f \in F$ of arity k the corresponding element $f^{\mathbf{A}} \in F^{\mathbf{A}}$ is a mapping $f^{\mathbf{A}} : A^k \to A$. The set of *term operations* of $\mathbf{A}$ is the set of all operations obtained from $F^{\mathbf{A}}$ and projection operations via finitely many compositions. All algebras of the same signature which identically satisfy a set of equations are called a variety. An algebra $\mathbf{A}$ is locally finite if for any finite subset X of A, the set of all results of term operations applied to elements of X (that is, the subalgebra generated by X) is also finite. A variety is locally finite if every algebra in it is.

There is a natural connection between operations and relations on the same set. It says that a k-ary relation and an n-ary operation are compatible if for any n vectors from the relation, the vector obtained by pointwise application of the operation is again in the relation. The classic results of universal algebra often connect the properties of the compatible equivalence relations, which form a lattice under inclusion called the congruence lattice, and other properties of algebras. In the paper [17] we mention the meet-semidistributivity of the congruence lattices of all algebras in a variety, which is the lattice implication $x \wedge z = y \wedge z \Rightarrow (x \vee y) \wedge z = x \wedge z$. An equivalent condition of the congruence meet-semidistributivity of a locally finite variety is omitting types of covers **1** and **2** in the congruence lattices of all finite algebras of the variety, [5]. For

any other definitions and basic results which are not found in this introductory part, the reader is referred to [4] for basic universal algebra and [5] for tame congruence theory.

Definition 2.1. *Let* $\mathbf{A}$ *be a finite algebra and* α *a minimal congruence of* $\mathbf{A}$ *(i.e.* $0_{\mathbf{A}} < \alpha$ *and if* β *is a congruence of* $\mathbf{A}$*, with* $0_{\mathbf{A}} < \beta \leq \alpha$*, then* $\beta = \alpha$*.)*

- *An* α*–minimal set of* $\mathbf{A}$ *is a subset* U *of* $\mathbf{A}$ *that satisfies following two conditions:*
  - $U = p(\mathbf{A})$ *for some unary polynomial* $p(x)$ *of* $\mathbf{A}$ *that is not constant on at least one* α*–class*
  - *with respect to containment,* U *is minimal having this property.*
- *An* α*–neighbourhood (or* α*–trace) of* $\mathbf{A}$ *is a subset* N *of* $\mathbf{A}$ *such that:*
  - $N = U \cap (a/_\alpha)$ *for some* α*–minimal set* U *and* α*–class* $a/_\alpha$
  - $|N| > 1$.

We can easily see that a given α–minimal set U must contain at least one, and possibly more, α–neighbourhoods.The union of all α–neighbourhoods in U is called the body of U, and the remaining elements of U form the tail of U. What is important here is that algebra $\mathbf{A}$ induces uniform structures on all its α–neighbourhoods, meaning they (the structures induced) all belong to the same of five possible types. Let us now define an induced structure.

Definition 2.2. *Let* $\mathbf{A}$ *be an algebra and* $U \subseteq \mathbf{A}$*. The algebra induced by* $\mathbf{A}$ *on* U *is the algebra with universe* U *whose basic operations consist of the restriction to* U *of all polynomials of* $\mathbf{A}$ *under which* U *is closed. We denote this induced algebra by* $\mathbf{A}|_U$.

Theorem 2.3. *Let* $\mathbf{A}$ *be a finite algebra and* α *a minimal congruence of* $\mathbf{A}$.

- *If* U *and* V *are* α*–minimal sets then* $\mathbf{A}|_U$ *and* $\mathbf{A}|_V$ *are isomorphic and in fact there is a polynomial* $p(x)$ *that maps* U *bijectively onto* V.
- *If* N *and* M *are* α*–neighbourhoods then* $\mathbf{A}|_N$ *and* $\mathbf{A}|_M$ *are isomorphic via the restriction of some polynomial of* $\mathbf{A}$.
- *If* N *is* α*–neighbourhood then* $\mathbf{A}|_N$ *is polynomially equivalent to one of:*
  1. *A unary algebra whose basic operations are all permutations (unary type);*
  2. *A one–dimensional vector space over some finite field (affine type);*
  3. *A* 2*–element boolean algebra (boolean type);*
  4. *A* 2*–element lattice (lattice type);*
  5. *A* 2*–element semilattice (semilattice type);*

Proof. The theorem in this form is given in [3], and the proof can be found in [5]. □

The previous theorem allows us to assign a type to each minimal congruence α of an algebra according to the behaviour of the α–neighbourhoods (for example, a minimal congruence whose α–neighbourhoods are polynomially equivalent to a vector space is said to have affine type or type 2).

Taking this idea one step further, given a pair of congruences (α, β) of $\mathbf{A}$ with β covering α (i.e. $\alpha < \beta$ and there are no congruences of $\mathbf{A}$ strictly between the two), one can form the quotient algebra $\mathbf{A}/_\alpha$, and then consider the congruence $\beta/_\alpha = \{(a/_\alpha, b/_\alpha) : (a, b) \in \beta\}$. Since β covers α in the congruence lattice of $\mathbf{A}$, $\beta/_\alpha$ is a minimal congruence of $\mathbf{A}/_\alpha$, so it can be assigned one of the five types. In this way we can assign to each covering pair of congruences of $\mathbf{A}$ a type (unary, affine, boolean, lattice, semilattice, or **1**, **2**, **3**, **4**, **5**, respectively). Therefore, going through all covering pairs of congruences of this algebra we obtain a set of types, so–called typeset of $\mathbf{A}$, denoted by $typ\{\mathbf{A}\}$. Also, for $\mathcal{K}$ a class of algebras, the typeset of $\mathcal{K}$ is defined to be the union of all the typesets of its finite members, denoted by $typ\{\mathcal{K}\}$.

A finite algebra or a class of algebras is said to omit a certain type if that type does not appear in its typeset. For locally finite varieties omitting certain types can be characterized by Maltsev conditions, i.e. by the existence of certain terms that satisfy certain linear identities, and there are quite a few results on this so far. We shall present two of them concerning omitting types 1 and 2.

Definition 2.4. *An n–ary term t, for $n > 1$, is a near–unanimity term for an algebra $\mathbf{A}$ if the identities $t(x, x, \ldots, x, y) \approx t(x, x, \ldots, y, x) \approx \cdots \approx t(x, y, \ldots, x, x) \approx t(y, x, \ldots, x, x) \approx x$ hold in $\mathbf{A}$.*

Definition 2.5. *An n–ary term t, for $n > 1$, is a weak near–unanimity term for an algebra $\mathbf{A}$ if it is idempotent and the identities $t(x, x, \ldots, x, y) \approx t(x, x, \ldots, y, x) \approx \cdots \approx t(x, y, \ldots, x, x) \approx t(y, x, \ldots, x, x)$ hold in $\mathbf{A}$.*

Theorem 2.6. *A locally finite variety $\mathcal{V}$ omits the unary and affine types (i.e. types 1 and 2) if and only if there is some $N > 0$ such that for all $k > N$, $\mathcal{V}$ has a weak near–unanimity term of arity k.*

Proof. The proof can be found in [11]. □

Theorem 2.7. *A locally finite variety $\mathcal{V}$ omits the unary and affine types if and only if it has 3–ary and 4–ary weak near–unanimity terms, v and w respectively, that satisfy the identity $v(y, x, x) \approx w(y, x, x, x)$.*

Proof. The proof can be found in [9]. □

Therefore, omitting types 1 and 2 for a locally finite variety (or, equivalently, congruence meet-semidistributivity for a locally finite variety) can be described by linear identities on 3–ary and a 4–ary term, both idempotent. In the paper [17] we examined whether the same can be done by two at most 3–ary idempotent terms. We came to this result:

3. A system possibly describing congruence meet–semidistributivity, i.e. omitting unary and affine types

Proposition 3.1. *If it is possible to describe congruence meet–semidistributivity (i.e. omitting types* **1** *and* **2***) in a locally finite variety by two ternary terms* p *and* q*, it can be done by this system:*

$$\begin{cases} p(x,x,y) \approx p(x,y,y) \\ p(x,y,x) \approx q(x,x,y) \approx q(x,y,x) \approx q(y,x,x) \end{cases} \tag{1}$$

It is easy to see that this system implies the property mentioned (the proof can be found in [17]), but it remained unresolved whether it actually describes the property. In attempt to find a counterexample , i.e. a finite algebra that generates a variety which satisfies congruence meet–semidistributivity but not the system above, we endeavor to examine small digraphs, i.e. corresponding algebras of polymorphisms.

Definition 3.2. *A digraph is a pair* $G = (V, E)$*, where* G *is a finite set of vertices and* $E \subseteq V \times V$ *is a set of edges.*

Definition 3.3. *An n–ary polymorphism of a digraph* $G = (V, E)$ *is a mapping* $f : V^n \to V$ *which preserves edges, that is for any* $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n) \in E$ *the pair* $(f(a_1, \ldots a_n), f(b_1, \ldots b_n))$ *is also in* E*.*

Definition 3.4. *An algebra of polymorphisms for a given digraph* $G = (V, E)$ *is an algebra with the universe* V *whose basic operations are polymorphisms of this digraph.*

Further research eplained here leans on results obtained by Libor Barto and David Stanovsky as presented in [16]. As said before, congruence meet–semidistibutivity of a locally finite variety is equivalent to omitting covers of types **1** or **2** in the congruence lattices of finite algebras in the variety (see [5]), and is also referred to as 'bounded width' due to the fact that 'local consistency checking' algorithm can faithfully solve the Constraint Satisfaction Problem in this case. So we say that a finite algebra is of 'bounded width' if and only if it generates a congruence meet–semidistributive variety.

4. Two–element and three–element digraphs

There are 10 non–isomorphic digraphs on two vertices and each of them has an 3–ary near–unanimity polymorphism, also called 'nu3' or a majority polymorphism ([16]). A finite algebra having a majority term–operation generates a congruence meet–semidistributive variety, but also satisfies the system (1) (this is proved in [17], Example 2). Therefore, algebras of polymorphisms of two–element digraphs are of no further interest to us.

As for three–element digraphs, we need to explain the case a bit more thoroughly, again referring to results in [16]. We shall first define a two–semilattice operation:

Definition 4.1. *A 2–semilattice (2–sml) operation is an idempotent operation satisfying* $f(x,y) \approx f(y,x)$ *and* $f(f(x,y),x) \approx f(x,y)$.

Fact 4.2. *A finite algebra having a 2–semilattice term–operation generates a congruence meet–semidistributive variety, but also satisfies system (1).*

This is proved in [17], Example 1.

Results given in [16] include figures of posets comparing the strength of polymorphisms. We can conclude the following about three–element digraphs, i.e. their algebras of polymorphisms: if they have a bounded width polymorphism, they either have a nu3 (majority), or a 2semilattice polymorphism (or both). Here 'bounded width polymorphism' stands for existence of both a 3–wnu and a 4–wnu polymorphisms sharing the same binary operation (Definition 2.5 and Theorem 2.7 above). Therefore for all these algebras of polymorphisms the following holds: if they generate a congruence meet–semidistributive variety (i.e. have a 'bounded width polymorphism') they also have either a nu3 or a 2–sml polymorphisms (or both), thereby satisfying system (1). This means they are of no further interest in our research.

5. Four–element digraphs

If we take a look at the strength of polymorphisms of these digraphs (figure 10 in [16]) we can conclude the following: nu3 and 2–sml polymorphisms both imply 'bounded width' polymorphism (are stronger). Also, the existence of a 2–sml polymorphism is equivalent to the existence of a weak near–unanimity polymorphism of arity 2, or wnu2 (in the sense that a digraph has the first one if and only if it has the second one). From Figure 11 in the same paper it can be seen that there are 29 digraphs on four vertices for which a 'bounded width' polymorphism is minimal and there are no digraphs of this size having wnu3 or wnu4 polymorphism as minimal. Since a 'bounded width polymorphism' is stronger than these two, we came to the following conclusion: if we exclude digraphs having a nu3 (majority) polymorphism and digraphs having a wnu2 polymorphism, remaining digraphs have a 'bounded width' polymorphism if and only if they have a wnu3 polymorphism (and there should be only 29 of them satisfying this condition). We managed to isolate these 29 digraphs and then tested them for existence of polymorphisms p and q as in the system (1). In the result we found out they all satisfy this system, so there is no counterexample of size four.

5.1. The procedure

C–source code examining four–element digraphs does the following (it can be found in [18]):

1. generates all digraphs of size 4
2. detects non–isomorphic ones

3. sets the matrix of subalgebras for non–isomorphic digraphs

4. then identifies ones having a nu3 polymorphism (a majority polymorphism) and excludes them from further examination (they are of bounded width but satisfy the system (1), so are of no interest here)

5. among the remaining digraphs it identifies ones having a wnu2–polymorphism (i.e. a binary idempotent commutative operation) and excludes these too for the same reason as above

6. the remaining digraphs are then tested on the existence of a wnu3 term, which gave us 29 digraphs having a 'bounded width' polymorphism as a minimal one

7. we then test these digraphs for existence of polymorphisms from the system (1)

8. in the final result all of these digraphs satisfy the system mentioned, so we have no counterexample among four–element digraphs

We shall provide here a bit more information on each item in the previous list (quite a detailed explanation is given with the source code in [18], as well as a number of comments within this code).

1. We represented a 4–element digraph by a 16–element array (of character type), members being '1' for an edge and '0' for no edge. The presumed order of edges in this array would be (0,0), (0,1), ...,(3,2), (3,3). Generating all digraphs of size four was done in the following way: we looked at each digraph as it was a binary number having exactly the same digits, so we could obtain the next digraph just by adding a binary 1 to the current one. We started, of course, from the array containing 16 zeroes (that is 16 characters '0'). This way we generated all 65536 digraphs with four vertices.

2. When detecting non–isomorphic digraphs we used the following procedure: we take the first digraph (from the array of all digraphs) having '0' on its isomorphism flag (not a copy), and then examine for isomorphism all digraphs with greater indexes and the same number of edges. Each time we find an ismorphic copy, we set its flag to '1' (is a copy). When all the copies are identified we proceed through the array to the next digraph having '0' on this flag and do the same. The procedure used is basically a 'brute force' one (improved by the fact that we only examine digraphs with the same number of edges for isomorphism), but it works fast enough since the number of 4–element digraphs is not too big. This way we detect all 3044 non–isomorphic digraphs.

3. As we are looking at idempotent polymorphisms only, all one-element subsets are subuniverses. Therefore, a four element digraph can have at most ten non-trivial subalgebras, so we formed a ten–column matrix of character type to keep information on subalgebras for each digraph (the number of rows is actually 65536, which is the number of all 4–element digraphs, but we only set values in rows corresponding to non–isomorphic digraphs). The presumed order of subalgebras is $\{0,1\}$, $\{0,2\}$, $\{0,3\}$, $\{1,2\}$ ...,$\{1,2,3\}$, and the matrix may contain only '0' for 'not known to be a subalgebra' and '1' for 'is a subalgebra'. Subalgebras are set by the function doing four checks, which means we did not check all possible conditions for subalgebras, but this worked just fine. Our function does the following checks for each one of non–isomorphic digraphs and for each of the sets above (say, for example, $\{0,2\}$):

   - we check whether there is a node in this digraph such that, say, 0 and 2 are the only nodes with edges leading to this node – if this is a case then $\{0,2\}$ is a subalgebra of the digraph given, so we set the corresponding element of the matrix to '1' and skip the following checks for the same subset;
   - if the first check did not give us the subalgebra result we do the next one – whether there is a node in the digraph having only two edges leading from it and exactly to 0 and 2 – if so $\{0,2\}$ is a subalgebra so '1' is being set on the proper matrix element, and the remaining two checks skipped;
   - the third condition – whether 0 and 2 are the only two nodes having paths of length one leading to them – if so we have a subalgebra, ..., we skip the last check;
   - the last condition – whether 0 and 2 are the only two nodes having paths of length one leading from them – if so we have a subalgebra, ..., if not we are finished with the set $\{0,2\}$ and the corresponding element of the matrix has not changed (there remains '0', not a subalgebra).

4. Examining digraphs for a majority polymorphism was done by a pair of recursive functions doing backtracking on a 24–element array. This array is of structure type, each of its elements containing a three element string of arguments for a majority polymorphism (the order is "012", "013", ..., "321") and a character type variable which should contain a value assigned to the polymorphism function in these arguments. Initially all the values are set to 'a', which is, of course, not an allowed value for a polymorphism. The functions operating on this array does the following: the first function, named f_m, sets the first value (that would be '0') on the element of the array being set, then calls the 'check' function to test whether the value assigned is compatible with the relation represented by the digraph and also with previously assigned values of this polymorphism (that is array). If we have compatibility the value is allowed, so f_m function would just move to the next element of the array being set and call itself again. In the opposite case, that is no–compatibility, f_m function attempts to assign the value '1' and then calls for a check, if it is allowed it goes to the next element of the array.... If none of the values '0', '1', '2', '3' can

be assigned to the current element of the array f_m function calls the other recursive function, named 'backwards', to reset previously defined elements of the array (that is all from the beginning to the current one, not including the current one) if possible. When these values are reset f_m will start again on the current element, beginning with '0'. If it was impossible to reset the preceding values this function will finish, leaving 'a' as a value on the current element of the array and also on all the following elements , which would subsequently be detected as 'no majority polymorphism'. So basically there are two recursive functions – one going forward, if possible, the other going backwards, that is resetting once set values in the array when detected that we can not move forward any more. When these functions are finished we have one of two possible outcomes: if all 24 values are set it means we have a mojority polymorphism on the digraph being examined, and if there is the value 'a' on the last element of the array it means there is no majority polymorphism (it is sufficient to test just the last value in the array once the functions finish). This worked just fine and fast enough in a sequential mode and we obtained the result – there are 1690 digraphs having this polymorphism.

5. Examining digraphs for a wnu2 polymorphism was done in the following way: first we create a matrix containing all idempotent commutative binary operations on four elements. A binary idempotent commutative operation needs only be defined on six pairs of arguments, i.e. $(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)$, so each operation was presented by a six–element row of this matrix containing its values in these pairs of arguments. For each digraph we go through the matrix and check compatibility – this is done by the function f_{pair}. We shall give a short example of how checking compatibility works: say we are examining an operation (presented by a row of the matrix mentioned) having 0 as a value in $(1,2)$, that is $f(1,2)=0$. For all nodes x and y of this digraph such that there is an edge from x to 1 and from y to 2 there should be an edge from $f(x,y)$ to 0. The other way round should also hold, that is for all nodes x and y such that there is an edge from 1 to x and from 2 to y there should be an edge from 0 to $f(x,y)$. Once we find a compatible operation we proceed to the next digraph.

6. The existence of wnu3 polymorphism, however, could not be examined in the way we did it for a majority polymorphism – namely backtracking explained above was in this case to be done on a 36–element array because a wnu3 polymorphism needs to be defined on the same 24 arguments like a majority polymorphism, that is "012", "013", ..., "321", but also on these: "001", "002",..., "330", "331", "332". The method used for a majority term did not work fast enough (sequential mode), because the backtracking array was too long, thereby generating too many recursive calls of the functions mentioned. We had to shorten 'the backtracking part' of this array, that is to assign some values to some of the arguments for this polymorphism and then attempt backtracking on the rest. The way we did this was the following:

    - we identified 2–element subalgebras of the digraphs (these were the re-

maining digraphs, that is the ones not having a majority nor a wnu2 polymorphism)

- The digraphs can have anywhere in between 0 and 6 two–element subalgebras, so we divided them into categories according to this number. We shall explain the further procedure on two of these categories – for digraphs having all six two–element subalgebras, and for those having, say, five of these.
- For digraphs having six two–element subalgebras we looked at these arguments for a wnu3 polymorphism: ”001”, ”002”, ”003”, ”110”, ’112”, ”113’,...,”330”, ”331”, ”332”. In each of these twelve arguments the polymorphism can only have one of the two possible values : 0 or 1 in ”001”, 0 or 2 in ”002” etc. We then generated a matrix containing all possible values for these twelve arguments (12 columns and 2^{12} rows). These twelve strings of arguments are in the beginning of the backtracking array, so we assigned row by row of this matrix to be the values of the wnu3 polymorphism for these specific arguments and attempted backtracking from the thirteenth member of the array to the end for each of this rows. This enhanced the procedure greatly, since the recursive functions doing backtracking were executed on a 24–member array. These functions had to be altered a bit in comparison to functions searching for a majority polymorphism – namely they only set the array from some point on, never changing the beginning of it, but when setting a particular value they check compatibility with all previous values including the ones from the matrix.
- In the case of five two–element subalgebras the things get a bit more complicated: there are six possible choices of five subalgebras and for each of these choices we have to create a new matrix of values and to alter the beginning of the backtracking array. For example if subalgebras are $\{0,1\}$, $\{0,2\}$, $\{0,3\}$, $\{1,2\}$, $\{1,3\}$ the matrix would have five columns and 2^5 rows, having only 0 and 1 in the first column, 0 and 2 in the second and so on. Moreover we have to put ten corresponding elements, that is ”001”, ”002”, ”003”, ”110”, ”112”, ”113”, ”220”, ”221”, ”330”, ”331”, in the beginning of the backtracking array (putting elements ”223” and ”332” on positions eleven and twelve). Then we do the same as in the previous case, that is we assign a row of the matrix to be the values on first ten members and attempt backtracking from the eleventh member on. We have to create a new matrix of values and to permute some elements of the backtracking array for each choice of five subalgebras, which means this part of the code needs to be executed within six iterations. In the functions doing backtracking we only need to change a starting point from the index 13 to the index 11 in the backtracking array. This is explained thoroughly in [18]. The backtracking part of the array is in this case 26 members long and it works fast enough.
- In all the remaining cases, that is four, three, two or one two–element subalgebra we did exactly the same as explained above. In each of these

cases we had to create a new matrix of possible values for each choice of subalgebras, and also to alter the beginning of the backtracking array so that the corresponding elements would come first. The functions searching for a wnu3 term would also start from different positions in this array. This means we had to iterate this part of the code quite a number of times to obtain the final results.

- The procedure described worked very well– luckily enough all the digraphs having a wnu3 polymorphism were among ones actually having two–element subalgebras, so we managed to find all 29 of them. What strikes us as interesting is that assigning values to just two elements of the 36–element array and doing backtracking on the rest of it, on the length 34 that is, was fast enough (this is the case with a single 2–element subalgebra). As mentioned before, backtracking on the length 36 was impossible (that is did not finish even after several hours for particular digraphs).
- This way we isolated 29 digraphs not having a majority nor a wnu2 polymorphism, but having a wnu3 polymorphism. As already said these are the ones having a 'bounded width' polymorphism as a minimal one, therefore were candidates for a counterexample.

7. In this step we tested these digraphs for existence of polymorphisms p and q from system 1. It was done in the following way:

   - first we find a wnu3 term for the digraph being examined; this is to be the q term from the system 1.
   - then we assign 12 values of this term to the elements at the beginning of the backtracking array for the p term (these would be the values of the wnu3 term for the following arguments: "010", "020", "030", "101", "121", "131", "202", "212", "232", "303", "313" and "323". In these arguments p and q have the same values according to the system 1).
   - next we do backtracking on the remaining of the array (that is we try to assign remaining 36 values for p). The functions doing backtracking and all the checks needed are exactly the same as described before (in the cases of majority term and wnu3 term), so we shall not comment them again. They can be found within the source code in [18].
   - in the result all 29 digraphs were found to satisfy the system 1, that is to have terms p and q.

   Interestingly enough, this backtracking on 36 elements when searching for the term p worked just fine, as opposed to the attempt on the same length when examining digraphs for a wnu3 polymorphism. Also, for each of these 29 digraphs the term p was found for the very first wnu3 term being examined (when we take the least wnu3 as the term q for a particular digraph, the least by its values that is, we can find the term p paired with this one so as to satisfy the system 1).

8. as already said we established there was no counterexample among 4–element digraphs.

5.2. Complexity

Now we shall estimate time complexity for each of the steps (i.e sections of code) mentioned above.

1. The time needed for generating all digraphs of size four (or of any given size) is a linear function of the number of digraphs of given size. If we denote a number of digraphs with N generating would take $C * N$ time units, C being some constant value. However, the number of digraphs is actually $N = 2^{2^n}$, n being a number of vertices, so this step takes $C * 2^{2^n}$ time units.

2. When searching for non–isomorphic digraphs we used a method close to the 'brute force' one as already said, and by 'brute force' we mean comparing a digraph for an isomorphism with all digraphs that follow this one in the array containing all digraphs. We improved this a bit by comparing only digraphs having the same number of edges, but basically the complexity of this method remained the same – if N is a number of all digraphs of given size this step takes $C * N^2$ time units, C being some constant value. We can also express this as a function of number of vertices, so it would be $C * (2^{2^n})^2$. This worked fine for 4–element digraphs, but for 5–element ones we used a better (and more clever) method as explained in the section 6.

3. The function checking subalgebras takes constant amount of time for each digraph, which gives us $C * N$ time units, where C is some constant value and N is the number of non–isomorphic digraphs.

4. Searching for a majority polymorphism was done by a pair of recursive functions, as already explained above. The first one, called f_m in the original code in [18], sets the value of the current element of the backtracking array (that is a value for this polymorphism on a given $3-tuple$ of arguments), if possible, and then moves forward in this array. The second one, called 'backwards', resets all previously set values in this array to new ones when setting the current value is not possible. Let us look at the time complexity here, analyzing only the worst possible case of course: say we are setting a value in the element of the array having index i. If it is impossible to set it considering previously set values the function 'backwards' can be called by the function f_m as many as 4^i times before f_m proceeds further (because this is the number of ways to set values on the preceding i members of this array). Each call of the function 'backwards' can generate 4^{i-1} new calls of this function, and each of these calls can give us 4^{i-2} new calls... So, basically, when setting the element of the array with the index i there can occur as many as $4^i * 4^{i-1} * 4^{i-2} * \cdots * 4 = 4^{i*(i+1)/2}$ calls of the function 'backwards'. So, if we suppose this is what happens when setting a value on each element of the array, we have $\Sigma_{i=1}^{i=M} 4^{i*(i+1)/2}$ calls of this function, M being the length of the backtracking array, that is the number of values to be set for a certain polymorphism. We can approximate the previous sum, from above, to $\mathbf{C * 4^{M*(M+1)/2}}$, C being some constant value (since $\Sigma_{i=0}^{i=\infty} 4^i = 4/3$). So the time complexity of this procedure would be $\mathbf{C * 4^{M*(M+1)/2}}$ time units (C a constant value, M the number of values to be

set for a certain polymorphism) , which is extremely high. There are a number of checks and loops accompanying these recursive calls, but they do not contribute significantly to the complexity obtained so we did not mention them for the sake of simplicity. The backtracking described worked for $M = 24$, a majority polymorphism needs 24 values defined, but could not work for $M = 36$ when searching for a wnu3 polymorphism. The breaking point was the length 35, or maybe even 36 since it worked on the length 34, as we have mentioned above.

At this point we could alter this complexity expression given above as to make it a function of the number of vertices – if a digraph had n vertices we would have to define $n * (n-1) * (n-2)$ values for a majority polymorphism, so the backtracking array would be that long, and this gives us the following complexity: $\mathbf{C * n^{n*(n-1)*(n-2)*(n*(n-1)*(n-2)+1)/2}}$.

5. When examining digraphs for a wnu2 polymorphism first we created a matrix of all idempotent commutative binary operations. Its dimensions are $n^{n*(n-1)/2} \times n * (n-1)/2$, where n is the number of vertices (in this case 4). For each digraph we went through this matrix checking compatibility, the check requiring a constant amount of time. Therefore time complexity of this section of code would be $C * n^{n*(n-1)/2} * N$, where C is some constant value, n is the number of vertices and N is the number of digraphs being examined (non–isomorphic digraphs not having a majority polymorphism).

6. The time complexity of the section of the code searching for a wnu3 polymorphism is the same as when searching for a majority polymorphism for we used the same functions to do the backtracking needed, $\mathbf{C * 4^{M*(M+1)/2}}$, where C is some constant value and M is the number of values to be set for a certain polymorphism (wnu3). We did shorten the backtracking part of the array from the length $M = 36$ (impossible) to the length at most 34 (possible) in the way described above, but generating the matrices needed and assigning values to the members at the beginning of the backtracking array do not effect overall complexity of this method – it is very high due to the number of recursive calls. Iterations mentioned above and needed to obtain results do not effect complexity either, they can just alter this constant value C in the expression.

7. The time complexity of the section of the code searching for polymorphism p and q is even greater than when searching for a majority polymorphism or a wnu3 polymorphism since the idea is to find a wnu3 polymorphism first, then to search for a p term for this particular wnu3, and in case it doesn't exist we search for another wnu3 polymorphism and then for a corresponding p term and so on. This basically means that the complexity would be (the upper bound) $\mathbf{C * 4^{M*(M+1)/2} * 4^{M_1*(M_1+1)/2}}$, where C is some constant value and M, M_1 are the numbers of values to be set for these polymorphisms (wnu3 and p respectively). The fact that in each case the term p was found for the very first wnu3 term being examined made things much easier for us, of course, but it does not effect the complexity – in the worst case it could be as given above.

6. Five–element digraphs

The five-element case is close to the boundary of currently available computing power. This is the reason why we used not only C-codes but also Paradox model builder when examining these digraphs.

For five–element digraphs exactly the same holds as for four–element ones regarding the strength of polymorphisms (figure 10 in [16]): if we exclude ones having a majority polymorphism and a wnu2 polymorphism the remaining digraphs are of 'bouded width' if and only if they have a wnu3 polymorphism. There should be 3475 of these digraphs, and exactly they would be the candidates for a counterexample.

It is here that we encountered a problem – namely we obtained results on a wnu2 polymorphism, but the source codes doing backtracking for 5–element digraphs were not fast enough, though ran in parallel mode, so we had to use Paradox model builder to examine these digraphs for majority, wnu3 and p and q polymorphisms.

6.1. The procedure used for a wnu2 polymorphism

Examining 5–element digraphs for a wnu2 polymorphism was done by two separate source codes (both can be found in [18]).

The first one works in sequential mode generating all 5–element digraphs and then detecting non–isomorphic ones. Since there are so many digraphs, the method used for this resembles the method for identifying all prime numbers known as 'the sieve of Eratosthen' – namely isomorphism flags for all digraphs are previously set to '0', so we take the first digraph , generate all its isomorphic copies and raise their flags to '1' in the array containing all digraphs. Then we take the next digraph having '0' on this flag, generate its copies and set their flags to '1', and so on... When raising the flags of copies to '1' we didn't search for the copies, through the array of all digraphs but rather calculate their positions in it – after generating each copy of a digraph we turned it from string format consisting of '0' and '1' into a binary number with the same digits, and then into a corresponding decimal number which is exactly the index of the copy in this array. In the end we write all the non–isomorphic digraphs into an output file. This way we came to 291968 non–isomorphic 5–element digraphs.

The second code is to be executed in parallel mode (it was originally executed on 13 processors in master–slave hierarchy) and it does the following:

- We read the output file made by the first code (containing all non–ismorphic digraphs) and store these digraphs into an array.
- We set the matrix containing all binary idempotent commutative operations on five nodes (because of idempotence and commutativity presumed each operation is represented by a 10–element row, this was already explained for 4–element digraphs).
- We set the matrix of subalgebras for all digraphs which would help us when checking the compatibility of binary operations with digraphs. In this case, again all one element subsets are subuniverses, so the matrix has 25 columns

for 25 potential nontrivial subalgebras (with 2, 3 or 4 elements) of a 5–element digraph.

- We test digraphs for a wnu2–term which now means checking compatibility of a digraph given with the operations from this matrix. This was done in exactly the same way as explained for a 4–element case. Once a compatible operation is found we proceed to the next digraph.

- Digraphs not having a compatible binary commutative operation (wnu2 term) are in the end written into a new output file.

The master–slave hierarchy mentioned above was implemented in the following way: each one of the slaves was given its own array of non–isomorphic digraphs, and also both a matrix of subalgebras and a matrix of idempotent binary commutative operations. The master processor was just distributing to the slaves the indexes of the digraphs to be examined and collecting results during the process (setting a wnu2–flag to '1' if need be in its own array of non–isomorphic digraphs). In the end this processor wrote the digraphs not having a wnu2 polymorphism into an output file.

The results obtained slightly differed from the ones given in [16] in the sense that we got 132509 *non–isomorphic digraphs having a wnu2 polymorphism, which is less by one from the number given in [16]. When examining the file provided by the authors containing the results of examination of all non–isomorphic 5–element digraphs for* 18 *different polymorphisms, we found out there were some redundant data at the end of it, so this was no more than a counting mistake on their behalf. However, it caused all the entries in the figure* 7 *in [16] to be greater by one then they actually are.*

The erratum is now on the website presenting results from [16].

6.2. Complexity

As in the previous section we shall estimate complexity for each part of the procedure:

- The time needed to generate all digraphs of size five is a linear function of the number of digraphs, so $C * N$, N being the number of digraphs, or $C * 2^{2^n}$, n being the number of vertices (here five).

- Detecting non–isomorphic digraphs in the way explained above takes $C * N$ time units, N being the number of all digraphs, because generating isomorphic copies for each digraph and then calculating their positions and setting corresponding flags to '1' takes a constant amount of time. This way we reduced the complexity by an order of magnitude, namely from $C * N^2$ (brute force) to $C * N$ ('the sieve of Eratosthen' method).

- As in the case of 4–element digraphs, setting the matrix of subalgebras takes $C * N$ time units, N being the number of non–isomorphic digraphs with five vertices.

- Setting the matrix of all idempotent commutative binary operations and examining non–isomorphic digraphs for this kind of operation (a wnu2 polymorphism) takes $C*n^{n*(n-1)/2}*N$ time units, where n is the number of vertices, here five, and N is the number of non–isomorphic digraphs of this size. The complexity expression is already explained in the 4–element case. Let us only notice that executing this section of code in parallel mode does not change its complexity but only the constant factor C, that is the actual amount of time needed.

6.3. Paradox

As mentioned above we used Paradox model builder to examine remaining digraphs for majority, wnu3 and p and q polymorphisms (as denoted in the system 1). Paradox was also executed in parallel mode under Linux shell. All input files were generated by C-programming language as text files (an input file depends on a digraph and a polymorphism). We shall present here an example of input file (this was used when searching for p and q, q being denoted by g):

```
cnf(mt, axiom, p(X,X,X)=X).
cnf(mt, axiom, p(X,X,Y)=p(X,Y,Y)).
cnf(pr, axiom, ~gr(X0,X1) | ~gr(X2,X3) |  ~gr(X4,X5)
    | gr(p(X0,X2,X4),p(X1,X3,X5))).
cnf(wnu, axiom, g(X,X,X)=X).
cnf(wnu, axiom, g(X,X,Y)=g(X,Y,X)).
cnf(wnu, axiom, g(X,X,Y)=g(Y,X,X)).
cnf(pr, axiom, ~gr(X0,X1) | ~gr(X2,X3) |  ~gr(X4,X5)
    | gr(g(X0,X2,X4),g(X1,X3,X5))).
cnf(mt, axiom, p(X,Y,X)=g(Y,X,X)).
cnf(graph, axiom, ~gr(n0,n0)).
cnf(graph, axiom, ~gr(n0,n1)).
cnf(graph, axiom, ~gr(n0,n2)).
cnf(graph, axiom, gr(n0,n3)).
cnf(graph, axiom, gr(n0,n4)).
cnf(graph, axiom, ~gr(n1,n0)).
cnf(graph, axiom, ~gr(n1,n1)).
cnf(graph, axiom, ~gr(n1,n2)).
cnf(graph, axiom, ~gr(n1,n3)).
cnf(graph, axiom, ~gr(n1,n4)).
cnf(graph, axiom, ~gr(n2,n0)).
cnf(graph, axiom, ~gr(n2,n1)).
cnf(graph, axiom, ~gr(n2,n2)).
cnf(graph, axiom, ~gr(n2,n3)).
cnf(graph, axiom, gr(n2,n4)).
cnf(graph, axiom, gr(n3,n0)).
cnf(graph, axiom, ~gr(n3,n1)).
cnf(graph, axiom, ~gr(n3,n2)).
cnf(graph, axiom, ~gr(n3,n3)).
cnf(graph, axiom, gr(n3,n4)).
cnf(graph, axiom, ~gr(n4,n0)).
```

```
cnf(graph, axiom, gr(n4,n1)).
cnf(graph, axiom, ~gr(n4,n2)).
cnf(graph, axiom, ~gr(n4,n3)).
cnf(graph, axiom, ~gr(n4,n4)).
cnf(elems, axiom, n0!=n1).
cnf(elems, axiom, n0!=n2).
cnf(elems, axiom, n0!=n3).
cnf(elems, axiom, n0!=n4).
cnf(elems, axiom, n1!=n2).
cnf(elems, axiom, n1!=n3).
cnf(elems, axiom, n1!=n4).
cnf(elems, axiom, n2!=n3).
cnf(elems, axiom, n2!=n4).
cnf(elems, axiom, n3!=n4).
cnf(elems, axiom, (X=n0 | X=n1 | X=n2 | X=n3 | X=n4)).
```

We managed to isolate 3475 5–element digraphs having a wnu3 polymorphism as a minimal one, and they proved to have p and q terms, that is to satisfy the system 1. This means we established there was no counterexample among 5–element digraphs, i.e. among corresponding algebras of polymorphisms.

7. The conjecture

Since we found no counterexample for the system 1 up to size 5, at this point we can state the conjecture:

A locally finite variety is congruence meet–semidistributive if and only if it satisfies the system 1:

$$\begin{cases} p(x,x,y) \approx p(x,y,y) \\ p(x,y,x) \approx q(x,x,y) \approx q(x,y,x) \approx q(y,x,x) \end{cases}$$

Acknowledgements :
I thank to both my phd. adviser Petar Marković and Tatjana Davidović for their help during this research.

References

[1] Baker, K., Finite equational bases for finite algebras in a congruence-distributive equational class. Adv. in Math. 24 (1977), 207-243.

[2] Barto, L., Kozik, M., Constraint satisfaction problems of bounded width. Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science, FOCS'09 (2009), 595-603.

[3] Bulatov, A., Valeriote, M., Results on the algebraic approach to the CSP, in Complexity of Constraints: An Overview of Current Research Themes. Springer-Verlag (2008), 68-92.

[4] Burris, S., Sankappanavar, H.P., A Course in Universal Algebra. Springer–Verlag New York Inc.(1981).

[5] Hobby,D., McKenzie, R., The Structure of Finite Algebras. Contemporary Mathematics, Vol.76 , American Mathematical Society, Providence,RI (1988, revised edition 1996).

[6] Jónsson, B., Algebras whose congruence lattices are distributive. Math. Scand. 21 (1967), 110-121.

[7] J. J.: Omitting unary and affine types. www.arXiv.org

[8] Kearnes, K., Willard, R., Residually finite, congruence meet-semidistributive varieties of finite type have a finite residual bound. Proc. Amer. Math. Soc. 127 (1999), 2841-2850.

[9] Kozik, M., Krokhin, A., Valeriote, M., Willard, R., On Maltsev Conditions associated with omitting certain types of local structure. Preprint.

[10] Malcev, A., On the general theory of algebraic systems. Mat. Sb. (77) 35 (1954), 3-20.

[11] Maróti, M., McKenzie, R.,Existence theorems for weakly symmetric operations. Algebra Universalis, 59(3-4) (2008), 463-489.

[12] Park, R., Equational classes of non-associative ordered algebras. Ph.D. dissertation, UCLA (1976).

[13] Quackenbush, R. W., Equational classes generated by finite algebras. Algebra Universalis 1(1971), 265-266.

[14] Valeriote, M., A subalgebra intersection property for congruence distributive varieties. The Canadian Journal of Mathematics, 61 (2009), no. 2, 451-464.

[15] Willard, R., A finite basis theorem for residually finite, congruence meet-semidistributive varieties. J. Symbolic Logic, 65 (2000), 187-200.

[16] Barto, L., Stanovsky, D., Polymorphisms of small digraphs. Novi Sad J. Math. 40/2 (2010), 95-109.

[17] J.J., On terms describing omitting unary and affine types. Filomat 27, no. 1 (2013), 183-199.

[18] J.J., On strong Mal'cev conditions for congruence meet–semidistributivity in a locally finite variety. www.ArXiv.org.

[19] Czedli, G., A characterization for congruence meet–semidistributivity. Proc. Conf. Universal Algebra and Lattice Theory, Puebla, Mexico (1982); Lecture Notes in Math.,vol.1004, pp. Springer, New York (1983), 104–110.

[20] Kearnes, K., Szendrei, Á., The relationship between two commutators. Internat.J.Algebra Comput.8 (1998), 497–531.

[21] Lipparini,P., A characterization of varieties with a difference term,II: Neutral = meet–semidistributive. Canad. Math. Bull. 41 (1998), 318–327.

Received by the editors September 11, 2014