Discrete Applied Mathematics 174 (2014) 27–51

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

On the van der Waerden numbers w(2; 3, t)

Tanbir Ahmed[a,*], Oliver Kullmann[b], Hunter Snevily[c,1]

[a] Department of Computer Science and Software Engineering, Concordia University, Montréal, Canada

[b] Computer Science Department, Swansea University, Swansea, UK

[c] Department of Mathematics, University of Idaho, Moscow, ID, USA

ARTICLE INFO

Article history:
Received 28 October 2011
Received in revised form 7 April 2014
Accepted 1 May 2014
Available online 16 May 2014

Keywords:
Arithmetic progression
Van der Waerden numbers
Ramsey theory
SAT solver

ABSTRACT

In this paper we present results and conjectures on the ordinary van der Waerden numbers $\mathrm{w}(2;3,t)$ and on the new *palindromic van der Waerden numbers* $\mathrm{pdw}(2;3,t)$. We have computed the exact value of the previously unknown number $\mathrm{w}(2;3,19) = 349$, and we provide new lower bounds for $20 \le t \le 39$, where for $20 \le t \le 30$ we conjecture these bounds to be exact. The lower bounds for $\mathrm{w}(2;3,t)$ with $24 \le t \le 30$ refute the conjecture that $\mathrm{w}(2;3,t) \le t^2$ as suggested in Brown et al. (2008). Based on the known values of $\mathrm{w}(2;3,t)$, we investigate regularities to better understand the lower bounds of $\mathrm{w}(2;3,t)$. Motivated by such regularities, we introduce palindromic van der Waerden numbers $\mathrm{pdw}(k;t_0,\ldots,t_{k-1})$, which are defined as the ordinary numbers $\mathrm{w}(k;t_0,\ldots,t_{k-1})$, but where only palindromic solutions are considered, reading the same from both ends. Different from the situation for ordinary van der Waerden numbers, these "numbers" need actually to be pairs of numbers. We compute $\mathrm{pdw}(2;3,t)$ for $3 \le t \le 27$, and we provide bounds for $t \le 39$, which we believe to be exact for $t \le 35$. All computations are based on SAT solving, and we discuss the various relations between SAT solving and Ramsey theory. Especially we introduce a novel (open-source) SAT solver, the `tawSolver`, which performs best on the SAT instances studied here, and which is actually the original DLL-solver by Davis et al. (1962), but with an efficient implementation and a modern heuristic typical for look-ahead solvers, applying the theory developed by the second author (Kullmann, 2009).

1. Introduction

We consider Ramsey theory and its connections to computer science (see [59] for a survey) by exploring a rather recent link, especially to algorithms and formal methods, namely to SAT solving. SAT is the problem of finding a satisfying assignment for a propositional formula. Since Ramsey problems can naturally be formulated as SAT problems, SAT solvers can be used to compute numbers from Ramsey theory. In the present article, we consider van der Waerden numbers [70], where SAT had its biggest success in Ramsey theory, namely the determination of $\mathrm{w}(2;6,6) = 1132$ in [44], the first new diagonal van der Waerden (short "vdW") number after almost 30 years.

Definition 1.1. We use $\mathbb{N} = \{x \in \mathbb{Z} : x \ge 1\}$, $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. An *arithmetic progression* of length $t \in \mathbb{N}$ is a subset $p \subset \mathbb{N}$ of length $|p| = t$ and of the form $p = \{a + i \cdot d : i \in \{0,\ldots,t-1\}\}$ for some $a, d \in \mathbb{N}$. A *block partition* of length $k \in \mathbb{N}$ of a

[*] Corresponding author.
E-mail addresses: ta_ahmed@cs.concordia.ca, tanbir@gmail.com (T. Ahmed), O.Kullmann@Swansea.ac.uk (O. Kullmann).

[1] Hunter Snevily passed away on November 11, 2013 after his long struggle with Parkinson's disease. He was an inspiring mathematician. We have lost a great friend and colleague. He will be heavily missed and fondly remembered.

http://dx.doi.org/10.1016/j.dam.2014.05.007
0166-218X/© 2014 Elsevier B.V. All rights reserved.

Table 1
Known values for w(2; 3, t).

t	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	**19**
w(2; 3, t)	9	18	22	32	46	58	77	97	114	135	160	186	218	238	279	312	**349**

set X is a tuple $(P_0, \dots, P_{k-1})$ of length k of subsets of X (possibly empty) which are pairwise disjoint ($P_i \cap P_j = \emptyset$ for $i \neq j$) and with $P_0 \cup \dots \cup P_{k-1} = X$. The *van der Waerden number* $\mathrm{w}(k; t_0, t_1, \dots, t_{k-1}) \in \mathbb{N}$ for $k, t_0, \dots, t_{k-1} \in \mathbb{N}$ is the smallest $n \in \mathbb{N}$ such that for any block partition $(P_0, \dots, P_{k-1})$ of length k of $\{1, \dots, n\}$ there exists a $j \in \{0, \dots, k-1\}$ such that P_j contains an arithmetic progression of length t_j.

That we have $\mathrm{w}(k; t_0, t_1, \dots, t_{k-1}) > n$ can be certified by an appropriate block partition of $\{1, \dots, n\}$; such partitions are the solutions of the SAT problems to be constructed, and we call them "good partitions":

Definition 1.2. A *good partition* of $\{1, \dots, n\}$ (where $n \in \mathbb{N}_0$) w.r.t. parameters $t_0, t_1, \dots, t_{k-1}$ is a block partition $(P_0, \dots, P_{k-1})$ of $\{1, \dots, n\}$ containing no block P_j with an arithmetic progression of length t_j (for any j).

So there exists a good partition of $\{1, \dots, n\}$ if and only if $n < \mathrm{w}(k; t_0, t_1, \dots, t_{k-1})$. For every $k, t_0, \dots, t_{k-1} \in \mathbb{N}$ the only block partition of $\{1, \dots, 0\} = \emptyset$ is $(\emptyset, \dots, \emptyset)$, and this is a good partition. In this paper, we are interested in the specific van der Waerden numbers $\mathrm{w}(2; 3, t)$, $t \geq 3$. Specialising the general definition we obtain:

w(2; 3, t) is the smallest $n \in \mathbb{N}$, such that
for all $P_0, P_1 \subseteq \{1, \dots, n\}$ with $P_0 \cap P_1 = \emptyset$ and $P_0 \cup P_1 = \{1, \dots, n\}$
either P_0 has an arithmetic progression of size 3 or P_1 has an arithmetic progression of size t, or both.

The known exact values of w(2; 3, t) are shown in Table 1 (with our contribution in bold).

As references and for relevant information on the above numbers, see Chvátal [16], Brown [14], Beeler and O'Neil [8], Kouril [44], Landman, Robertson and Culver [54], and Ahmed [2–5].[2] Recently, Kullmann [50][3] reported the following lower bounds

$$\mathrm{w}(2; 3, 19) \geq 349, \qquad \mathrm{w}(2; 3, 20) \geq 389, \qquad \mathrm{w}(2; 3, 21) \geq 416.$$

We confirm the exact value of $\mathrm{w}(2; 3, 19) = 349$, and we extend the list of lower bounds up to $t = 39$. Brown, Landman, and Robertson [15], showed the lower bound $\mathrm{w}(2; 3, t) > t^{2-1/\log\log t}$ for $t \geq 4 \cdot 10^{316}$, and observed that $\mathrm{w}(2; 3, t) \leq t^2$ for $5 \leq t \leq 16$, suggesting that this might hold for all t. Our lower bounds in Section 3.2 however prove that there are t with $\mathrm{w}(2; 3, t) > t^2$. We provide an improved upper bound $1.675t^2$ in Section 3.3 (satisfying all known values and lower bounds of w(2; 3, t)).

We also present a new type of van-der-Waerden-like numbers, namely *palindromic number-pairs*, obtained by the constraint on good partitions that they must be symmetric under reflection at the mid-point of the interval $\{1, \dots, n\}$. Perceived originally only as a heuristic tool for studying ordinary van der Waerden numbers, it turned out that these numbers are interesting objects on their own. An interesting phenomenon is that we no longer have the standard behaviour of the SAT instances with increasing n, where

- first all instances are satisfiable (for $n < \mathrm{w}(k; t_0, \dots, t_{k-1})$), and from a certain point on (the van der Waerden number) all instances are unsatisfiable (for $n \geq \mathrm{w}(k; t_0, \dots, t_{k-1})$),
- but now first again all instances are satisfiable (for $n \leq p$), then we have a region with strict alternation between unsatisfiability and satisfiability, and only from a second point on all instances are unsatisfiable (for $n \geq q$).

These two turning points constitute the palindromic "number" $\mathrm{pdw}(2; 3, t) = (p, q)$ as pairs of natural numbers. We were able to compute pdw(2; 3, t) for $t \leq 27$. We also provide (conjectured) values for $t \leq 39$.[4] The full definition is in Section 5, while the special case experimentally studied in this paper is defined as follows:

In $\mathrm{pdw}(2; 3, t) = (p, q)$,
the number q is the smallest number such that for all $n \geq q$ and
for all $P_0, P_1 \subseteq \{1, \dots, n\}$ with $P_0 \cap P_1 = \emptyset$ and $P_0 \cup P_1 = \{1, \dots, n\}$ with the property,
that for all $v \in \{1, \dots, n\}$ we have $v \in P_0 \Leftrightarrow n + 1 - v \in P_0$ and $v \in P_1 \Leftrightarrow n + 1 - v \in P_1$,
either P_0 has an arithmetic progression of size 3 or P_1 has an arithmetic progression of size t, or both.
While p is the largest number such that for all $n \leq p$ and for all such (P_0, P_1)
neither P_0 has an arithmetic progression of size 3 nor P_1 has an arithmetic progression of size t.

[2] This sequence is http://oeis.org/A007783 in the "On-Line Encyclopedia of Integer Sequences".
[3] The conference article [51] contains only material related to Green–Tao numbers and SAT.
[4] The sequence pdw(2; 3, t) is http://oeis.org/A198684, http://oeis.org/A198685 in the "On-Line Encyclopedia of Integer Sequences" (the first and second components).

In the ordinary case of plain partitions (without the additional symmetry condition) we have $p + 1 = q$, and thus one uses just one number (instead of a pair), however here we can have a "palindromic span", that is, $p + 1 < q$ can happen for the palindromic case. The reason is that from a good partition of $\{1, \dots, n\}$ we obtain a good partition of $\{1, \dots, n-1\}$ by simple removing n, however for "good palindromic partitions" besides removing n we also need to remove the corresponding vertex 1 (due to the palindromicity condition).

Apparently the most advanced special algorithm (and implementation) for computing (mixed) van der Waerden numbers is the algorithm/implementation developed in [64].[5] For computing $\mathrm{w}(2; 3, 17) = 279$, with this special algorithm a run-time of 552 days is reported; the machine used should be at most 30% slower than the machine used in our experiments, and so this should translate into at least 400 days on our machine. As we can see in Table 9, the `tawSolver`-2.6 used is 85-times faster, while Table 10 shows, that `Cube & Conquer` is around 40-times faster. These algorithms know nothing about the specific problem, and are just given the generic SAT formulation of the underlying hypergraph colouring problem. So it seems that SAT solving does a good job here.[6]

1.1. *Using SAT solvers*

As explored in Dransfield et al. [20], Herwig et al. [28], Kouril [44,43], Ahmed [2,3], and Kullmann [50,51], we can generate an instance $F(t_0, \dots, t_{k-1}; n)$ of the satisfiability problem (for definition, see any of the above references) corresponding to $\mathrm{w}(k; t_0, t_1, \dots, t_{k-1})$ and integer n, such that $F(t_0, \dots, t_{k-1}; n)$ is satisfiable if and only if $n < \mathrm{w}(k; t_0, t_1, \dots, t_{k-1})$. In particular, the instance $F(3, t; n)$ corresponding to $\mathrm{w}(2; 3, t)$ with n variables consists of the following clauses:

(a) $\{x_a, x_{a+d}, x_{a+2d}\}$ with $a \geq 1, d \geq 1, a + 2d \leq n$, and
(b) $\{\overline{x}_a, \overline{x}_{a+d}, \dots, \overline{x}_{a+d(t-1)}\}$ with $a \geq 1, d \geq 1, a + d(t-1) \leq n$,

where an assignment $x_i = \varepsilon$ encodes $i \in P_\varepsilon$ for $\varepsilon \in \{0, 1\}$ (if x_i is not assigned but the formula is satisfied, then i can be arbitrarily placed in either of the blocks of the partition). The ("positive") clauses (a) (consisting only of variables), constructed from all arithmetic progressions of length 3 in $\{1, \dots, n\}$, prohibit the existence of an arithmetic progression of length 3 in P_0. And the ("negative") clauses (b) (consisting only of negated variables), constructed from all arithmetic progressions of length t in $\{1, \dots, n\}$, prohibit the existence of an arithmetic progression of length t in P_1. To check the satisfiability of the generated instance, we need to use a "SAT solver". A complete SAT solver finds a satisfying assignment if one exists, and otherwise correctly says that no satisfying assignment exists and the formula is unsatisfiable. One of the earliest complete algorithms is the DLL algorithm [19], and our algorithm for computing $\mathrm{w}(2; 3, 19) \leq 349$, discussed in Section 2, actually implements this very basic scheme, using modern heuristics.

SAT solving has progressed much beyond this simple algorithm, and the handbook [13] gives an overview (where [71] discusses some applications of SAT to combinatorics). There in [18] we find a general overview on complete SAT algorithms, while [41] gives an overview on incomplete algorithms. For complete algorithms especially the algorithms derived from the DLL algorithm are of importance, and there are two families, namely the (earlier) "look-ahead solvers" outlined in [31], and the (later) "conflict-driven solvers" (or "CDCL" like "conflict-driven clause-learning") outlined in [56]. In Section 6 we will discuss how general SAT solvers perform on the problems from this article. The motivation for our choice of the most basic DLL algorithm for tackling the unsatisfiability of the instance $F(3, 19; 349)$, already employed in [3] and discussed in Section 3.1, is, that on these special problems classes this basic algorithm together with a modern heuristic is very competitive—best on ordinary problem instances, and beaten on palindromic instances only by the `Cube & Conquer` method.[7] And then it is also instructive to use such an algorithm, which due to its simplicity might enable greater insight. Another advantage of its simplicity is, that it can also count and enumerate the solutions, but in this article we focus mostly on mere SAT solving; see [23] for an overview on counting solutions.

Local-search based incomplete algorithms (see Ubcsat-suite [68]) are generally faster than a DLL-like algorithm in finding a satisfying assignment (on such combinatorial problems), and this is also the case for the instances of this article. However they may fail to deliver a satisfying assignment when there exists one, and they cannot prove unsatisfiability. If they succeed on our instances, then they deliver a good partition, and thus a lower bound for a certain van der Waerden number. So such incomplete algorithms are used for obtaining good partitions and improving lower bounds of van der Waerden numbers. When they fail to improve the lower bound any further, we need to turn to a complete algorithm.

1.1.1. *Informed versus uninformed SAT solving*

We use general SAT solvers, and the new solvers developed by us are also general SAT solvers, which can run without modification on any SAT problem; these solvers just run on the naked and natural SAT formulation of the problem, without giving them further information. More specifically, to show unsatisfiability we have developed the `tawSolver` (Section 2)

5 http://www.mpi-inf.mpg.de/~pascal/software/VanderWaerdenR1160.tar.

6 As discussed in Section 2.1, for enumerating all solutions for $n = \mathrm{w}(2; 3, 17) - 1 = 278$ with `tawSolver`-2.6 we need at most the time needed for determining unsatisfiability; in this special case we have actually precisely one solution.

7 The `Cube & Conquer` method, developed originally on the instances of this article, combines a look-ahead solver with a conflict-driven solver, and is faster by a factor of two on palindromic instances.

and the Cube & Conquer-method (Section 6.1.1), while to find satisfying assignments we have selected local-search algorithms (Section 6.2).

On the other end of the spectrum is [44,43], which uses a highly specialised method, which involves a variety of specialised SAT solvers on specialised hardware, in combination with some special insights into the problem domain. For finding satisfying assignments we have the methods developed in [28,34,33]. For more examples on informed search to compute van der Waerden numbers, see also Section 2 of [5].

Our ''uninformed approach'' has stronger bearings on general SAT solving, while the informed approach can be more efficient for producing numerical results (however it seems to need a lot of effort to beat general SAT solvers (by specialised SAT solvers); as we have already reported, our general methods are at least on the instances of this paper much faster than the dedicated (non-SAT-based) method in [64]).

1.1.2. Parallel/distributed SAT solving

The problems we consider are computationally hard, and for the hardest of them in this paper, computation of $w(2; 3, 19) = 349$, a single processor, even when run for a long time, is not enough. Hence some form of parallelisation or distribution of the work is needed. Four levels of parallelisation have been considered for general-purpose SAT solving (in a variety of schemes):

(i) Processor-level parallelisation: This helps only for very special algorithms, and can only achieve some relatively small speed-up; see [32] for an example which exploits parallel bit-operations. It seems to play no role for the problems we are considering.

(ii) Computer-level parallelisation: Here it is exploited that currently a single (standard) computer can contain up to, say, 16 relatively independent processing units, working on shared memory. So threads (or processes) can run in parallel, using one (or more) of the following general forms of collaboration:
  (a) Partitioning the work via partitioning the instance (see below); [72,40] are ''classical'' examples.
  (b) Using the same algorithm running in various nodes on the same problem, exploiting randomisation and/or sharing of learned results; see [37,25] for recent examples.
  (c) Using some portfolio approach, running different algorithms on the same problem, exploiting that various algorithms can behave very differently and unpredictably; see [24] for the first example.

  Often these approaches are combined in various ways; see [63,22,38,39] for recent examples. Approaches (b) and (c) do not seem to be of much use for the well-specified problem domain of hard instances from Ramsey theory. Only (a) is relevant, but in a more extreme form (see below). In the context of (ii), still only relatively ''easy'' problems (compared to the hard problems from Ramsey theory) are tackled.

(iii) Parallelisation on a cluster of computers: Here up to, say, 100 computers are considered, with restricted communication (though typically still non-trivial). In this case, the approach (ii)(a) becomes more dominant, but other considerations of (ii) are still relevant. For hard problems this form of computation is a common approach.

(iv) Internet computation, with completely independent computers, and only very basic communication between the centre and the machines: In principle, the number of computers is unbounded. Since progress must be guaranteed, and the instances for which Internet computation is applied would be very hard, at the global level only (ii)(a) is applicable (while at a local level all the other schemes can in principle be applied). Yet there is no real example for a SAT computation at this level.

We remark that the classical area of ''high performance computing'' seems to be of no relevance for SAT solving, since the basic SAT algorithms like unit-clause propagation are, different from typical forms of numerical computation, inherently sequential (compare also our remarks to (i)). However using dedicated hardware with specialised algorithms has been utilised in [44,43], yielding the currently most efficient machinery for computing van der Waerden numbers.

A major advantage of the DLL solver architecture (which has been further developed into so-called ''look-ahead'' SAT solvers) is that the computation is easily parallelisable and distributable: Just compute the tree only up to a certain depth d, and solve the (up to) 2^d sub-problems at level d. Only minimal interaction is required: The sub-problems are solved independently, and in case one sub-problem has been found satisfiable, then the whole search can be aborted (for the purpose of mere SAT-solving; for counting all solutions of course the search needs to be completed). And the sub-problems are accessible via the partial assignment constituting the path from the root to the corresponding leaf, and thus also require only small storage space. This is the core of method (ii)(a) from above, and will be further considered in Section 3.1 (for our special example class).

In the subsequent subsection we will discuss the general merits of applying SAT solving to (hard) Ramsey problems. One spin-off of this combination lies in pushing the frontier of large computations. As a first example we have developed in [30,69], motivated by the considerations of the present article, an improved method for (ii)(a) called ''Cube & Conquer'', which is also relevant for industrial problems (typically from the verification area). One aspect exploited here is that for extremely hard problems, splitting into millions of sub-instances is needed. In the literature until now (see above for examples) only splitting as required, by at most hundreds of processors, has been performed, while it turned out that the above ''extreme splitting'', when combined with ''modern'' (CDCL) SAT solvers, is even beneficial when considered as a (hybrid) solver on a single-processor, and this for a large range of problem instances.

1.1.3. Synergies between Ramsey theory and SAT

For Ramsey-numbers (see [58] for an overview on exact results), relatively precise asymptotic bounds exist, and due to the inherent symmetry, relatively specialised methods for solving concrete instances have an advantage. Van-der-Waerden-like numbers seem harder to tackle, both asymptotically and exactly, and perhaps the only way ever to know the precise values is by computation (and perhaps this is also true for Ramsey-numbers, only more structures are to be exploited). SAT solvers are especially suited for the task, since the computational problems are hypergraph-colouring problems, which, at least for two colours, have canonical translations into SAT problems (as only considered in this paper). For more colours, see the approach started in [51], while for a general theory of multi-valued SAT close to hypergraph-colouring, see [52,53].

Through applying and improving SAT solvers (as in the present article), Ramsey theory itself acquires an applied side. Perhaps unknown to many mathematicians is the fact, that whenever for example a recent microchip is employed, this likely involves SAT solving, playing an important (though typically hidden) role in its development, by providing the underlying "engines" for its verification; see the recent handbook [13] to get some impression of this astounding development. Now we believe that problem instances from Ramsey theory are good benchmarks, serving to improve SAT solvers on hard instances:

- Unlike with random instances (see [1] for an overview), instances from Ramsey theory are "structured" in various ways. One special structure which one finds in all these instances is that they are layered by the number of vertices (the same structural pattern is repeated again and again, on growing scales).
- A major advantage of random instances is their scalability, that is, we can create relatively easily instances of the same "structure" and different sizes. With instances from Ramsey theory we can also vary the parameters, however due to the possibly large and unknown growth of Ramsey-like numbers, controlling satisfiability and hardness is more complicated here. This possible disadvantage can be overcome through computational studies like in this paper, which serve to calibrate the scale via precise numerical data, so that the field of SAT instances from Ramsey-theory becomes accessible (one knows for initial parameter values the satisfiability status and (apparent) solving complexity, and gets a feeling what happens beyond that).
- In this paper, we consider two instance classes: instances related to ordinary van der Waerden numbers $\mathrm{w}(2; 3, t)$ and instances related to the palindromic forms $\mathrm{pdw}(2; 3, t)$. Now already with these two classes, the two main types of complete SAT solvers, "look-ahead" (see [31]) and "conflict-driven" (see [56]), are covered in the sense that they dominate on one class each (and are (relatively) efficient); see Section 6 for further details. On the other hand, for random instances only look-ahead solvers are efficient (for complete solvers).
- Especially for local-search methods (see [41] for an overview), these problems are hard, but not overwhelmingly so (for the ranges considered), and thus all the given lower bounds can trigger further progress (and insight) into the solution process in a relatively simple engineering-like manner (by studying which algorithms work best where).
- On the other hand, for upper bounds we need to show unsatisfiability, which is much harder (we can only solve much smaller instances). All applications of SAT solving in hardware verification are "unsatisfiability-driven" (see [10,45] for introductions). So future progress in solving hard Ramsey instances might trigger a breakthrough in tackling unsatisfiability, and should then also improve these industrial applications.

We believe that for better SAT solving, established hard problem instances are needed in a great variety, and we believe that Ramsey theory offers this potential. To begin the process of applying Ramsey theory in this direction, problem instances from this paper (as well as related to [51]) have been used in the SAT 2011 competition (http://www.satcompetition.org/2011/). As already mentioned in the previous subsection, the first fruits of the collaboration between SAT and Ramsey theory appeared in [30,69], yielding a method for tackling hard problems with strong scalability.

Finally, the interaction between Ramsey theory and SAT should yield new insights for Ramsey theory itself:

1. The numerical data can yield conjectures on growth rates; see Section 3.3.
2. The good partitions found can yield conjectures on patterns; see Section 4.
3. New forms of Ramsey problems can be found through algorithmic considerations; see Section 5.
4. The SAT solving process, considered *in detail*, acts like a microscope, enabling insights into the structure of the problem instances which are out of sight for Ramsey theory yet. For approaches towards structures in SAT instances, which we hope to study in the future, see [62,42].

1.2. The results of this paper

In Section 2, we present the new SAT solver, `tawSolver`-2.6, with superior performance on the instances considered in this paper (only for palindromic instances the new hybrid method `Cube & Conquer` is superior). Section 3 contains our results on the numbers $\mathrm{w}(2; 3, t)$. We discuss the computation of the one new van der Waerden number, and present further conjectures regarding precise values[8] and the growth rate. In Section 4, we investigate some patterns we found in the good partitions (establishing the lower bounds). In Section 5, we introduce palindromic problems and the corresponding palindromic number-pairs. Finally in Section 6, we discuss the observations on the use of the various SAT solvers involved.

[8] To establish these conjectures will require major advances in SAT solving.

The certificates of the lower bounds and how to access all the numbers and many methods in the `OKlibrary` (an open-source research platform for hard problems around the SAT problem) can be found in the underlying report [6].

In this paper, we represent partitions of $\mathrm{w}(2; 3, t)$ as bitstrings. For example, the partition $P_0 = \{1, 4, 5, 8\}$ and $P_1 = \{2, 3, 6, 7\}$, which is an example of a good partition of $\{1, 2, \dots, 8\}$, where $8 = \mathrm{w}(2; 3, 3) - 1$, is represented as 01100110, or more compactly as $01^2 0^2 1^2 0$, using exponentiation to denote repetition of bits.

2. The `tawSolver`

We now discuss the `tawSolver`, an open-source SAT solver, created by the first author with a special focus on van der Waerden problems (version 1.0), and improved by the second author through an improved branching heuristic (version 2.6).[9] Algorithm 1 shows that the basic algorithm of the `tawSolver` is the simplest possible (reasonable) DLL-scheme, just branching on a variable plus unit-clause propagation. As we can see in Section 6, it is the strongest SAT solver on the instances considered in this paper, only beaten on palindromic problems by the new hybrid scheme `Cube & Conquer`, which came out as a result on research on the instances of this paper.

2.1. The basic structure

Algorithm 1 specifies the `tawSolver`, which for input F (a formula or "clause-set") decides satisfiability:

1. Lines 3–5 is "unit-clause propagation" (UCP), denoted by the function r_1, which sets literals x in the current F to true while there are unit-clauses $\{x\} \in F$.
   (a) Setting a literal x to true in a clause-set F is performed by first removing all clauses from F containing x, and removing the element $\overline{x}$ from the remaining clauses.
   (b) r_1 finds a contradiction (Line 4) by finding two unit-clauses $\{v\}$ and $\{\overline{v}\}$ (i.e., $v \wedge \neg v$).
   (c) While r_1 finds a satisfying assignment (Line 5) if all clauses vanished (have been satisfied).
2. Lines 6–7 give the branching heuristic, which yields the branching literal x, first set to true, then to false, in the recursive call of the `tawSolver`.
   (a) $p(a, b) \in \mathbb{R}_{>0}$ for $a, b \in \mathbb{R}_{>0}$ in Line 6 is the "projection", and we consider three choices p_+, p_*, p_τ.
   (b) $w_F(x)$ for literal x is a heuristical value, measuring in a sense the "progress achieved" when setting x to FALSE ("progress" in the sense of the instance becoming more constrained, so that more unit-clause propagations are to be expected).
   (c) The details are specified in Sections 2.3 and 2.5.
3. The implementation is discussed in Section 2.4.
4. The tree of recursive calls made by the solver is called the *DLL-tree* of F.

Besides the choice of the heuristic, this is the basic SAT solver as published in [19]. The implementation is optimised for the needs of the branching heuristic, which requires to know from each (original) clause in the input F whether it has been satisfied meanwhile, and if not, what is its current length.

Algorithm 1 `tawSolver`

```
1:  Global variable $F$, initialised by the input.
2:  **function** DLL( ) : returns SAT or UNSAT for the current $F$
3:      Update $F$ to $r_1(F)$
4:      If contradiction found via $r_1$, then goto line 12
5:      If satisfying assignment found via $r_1$, then return SAT
6:      Choose variable $v$ with maximal $p(w_F(v), w_F(\overline{v}))$
7:      If $w_F(v) \geq w_F(\overline{v})$, then $x := v$, else $x := \overline{v}$
8:      Set $x$ to TRUE in $F$; if DLL( ) = SAT, then return SAT
9:      Undo assignment of $x$
10:     Set $\overline{x}$ to TRUE in $F$; if DLL( ) = SAT, then return SAT
11:     Undo assignment of $\overline{x}$
12:     Undo assignments made by $r_1$
13:     Return UNSAT
14: **end function**
```

With a small modification, namely just continuing when a satisfying assignment was found, the `tawSolver` can also count all satisfying assignments, or output them; this is available as a compile-time option for the solver. In Section 4, we will discuss some patterns which we found in satisfying assignments for $F(3, t; n)$ with $n < \mathrm{w}(2; 3, t)$. We do not report run-times for determining (or counting) all solutions in Section 6, but for $n = \mathrm{w}(2; 3, t) - 1$ (empirically) the run-time is at most the run-time needed to determine unsatisfiability for $n = \mathrm{w}(2; 3, t)$; for numerical values of solution-counts see [43].

[9] http://sourceforge.net/projects/tawsolver/, and in the `OKlibrary`: https://github.com/OKullmann/oklibrary/blob/master/Satisfiability/Solvers/TawSolver/tawSolver.cpp.

2.2. Look-ahead solvers

It is useful for the general picture to consider the general r_k-operations, as introduced in [46] and further studied in [26,27]. These operations transform a clause-set F into a satisfiability-equivalent clause-set via application of some forced assignments (i.e., where the opposite assignments would yield an unsatisfiable clause-set). Let $\bot$ be the empty clause, which stands for a trivial contradiction. r_0 just maps F to $\{\bot\}$ in case of $\bot \in F$, while otherwise F is left unchanged. Now we can recognise r_1 as an operation which is applied recursively to the result of F with literal x set to true if setting $\overline{x}$ to true yields $\{\bot\}$ via r_0. This scheme yields also the general r_k for $k \in \mathbb{N}$: as long as there is a literal x such that F with $\overline{x}$ set to true yields $\{\bot\}$ via r_{k-1}, set x to true and iterate. The final result, denoted by $r_k(F)$, is uniquely determined. Besides the ubiquitous unit-clause propagation r_1 also r_2, called "failed literal elimination", is popular for SAT solving, and even r_3, typically called "double look-ahead", is used in some solvers (always partially, testing the reductions only for selected variables).

The general scheme for a look-ahead solver (as stipulated in [49]) now generalises the DLL-procedure from Algorithm 1, by replacing the reduction $F \rightsquigarrow r_1(F)$ in Line 3 by the general $F \rightsquigarrow r_k(F)$ for some $k \geq 1$. Furthermore, for the inspection of a branching variable and the computation of the heuristical values $w(v)$ and $w(\overline{v})$, now the effects of setting v resp. $\overline{v}$ to true and performing r_{k-1} reduction are considered. This explains also the notion of "look-ahead": the r_k-reduction can be partially achieved at the time when running through all variables v, setting v resp. $\overline{v}$ to true and applying r_{k-1}—if this yields $\{\bot\}$, then performing the opposite assignment is justified. Since r_1 is the standard for reduction of a branch, (partial) r_2 is the default for the reduction at a node.[10]

We see that `tawSolver` uses $k = 1$ (so the "look-ahead" uses $k = 0$, and in this sense `tawSolver` is a "look-ahead solver with zero look-ahead"). The prototypical solver for using $k = 2$ is the `OKsolver` [47]. In a rather precise sense the `tawSolver` can be considered at the level-1-version of the `OKsolver` (or the latter as the level-2-version of the `tawSolver`). Also for the branching heuristic, which is discussed in the following subsection, `tawSolver` uses the same scheme as the `OKsolver`, appropriately simplified to the lower level. Both `tawSolver` and `OKsolver` are solvers with a "mathematical meaning", precisely implementing an algorithm to full extent, with the only magical numbers the clause-weights used in the branching heuristic.

The general scheme for the branching heuristic of a look-ahead solver, as developed in [49, Section 7.7.2], is as follows: For a clause-set F and its direct successor F' on a branch (applying the branching assignment and further reductions), a "distance measure" $d(F, F') \in \mathbb{R}_{>0}$ is chosen, with the meaning the bigger this distance, the larger the decrease in complexity. The branching heuristic considers for each variable v its two successors F', F'' and computes the distances $d(F, F'), d(F, F'')$. Then via a "projection" $p : \mathbb{R}^2_{>0} \to \mathbb{R}_{>0}$ one heuristical value $h_v := p(d(F, F'), d(F, F''))$ is obtained. Finally some v with maximal h_v is chosen. Choosing which of v or $\overline{v}$ to be processed first (important for satisfiable instances) is done via a second heuristic, estimating the satisfiability-probabilities of F', F'' in some way.

2.3. From `tawSolver`-1.0 to `tawSolver`-2.6

We are now turning to the discussion of the branching heuristic in `tawSolver`-2.6 (lines 6, 7 in Algorithm 1), the version developed for this article. For `tawSolver`-1.0 (used in [2,3]) the "Two-sided Jeroslaw-Wang" (2sJW) rule by Hooker and Vinay [35] was used, which chooses v such that the weighted sum of the number of clauses of F containing v is maximal, where the weight of a clause of length k is 2^{-k}.[11] As discussed in [49], the ideas from [35] are actually rather misleading, and this is demonstrated here again by obtaining a large speed-up through the replacement of the branching heuristic, as can be seen by the data in Section 6 (comparing `tawSolver`-1.0 with `tawSolver`-2.6).

For a literal x, a clause-set F and $k \in \mathbb{N}$ let $\mathrm{ld}^k_F(x) := |\{C \in F : x \in C \wedge |C| = k\}|$ be the "literal degree" of x in the k-clauses of F. The 2sJW-rule consists of three components:

1. The weight $w_F(x)$ of literal x is set as $w_F(x) := \sum_k 2^{-k} \cdot \mathrm{ld}^k_F(x)$.
2. A variable v with maximal $p_+(w_F(v), w_F(\overline{v}))$ for $p_+(a, b) := a + b$ is chosen.
3. The literal $x \in \{v, \overline{v}\}$ to be set first to true is given by the condition $w_F(x) \geq w_F(\overline{x})$.

This approach has the following fundamental flaws:

1. The choice of the first branch (v or $\overline{v}$) is mixed up with the choice of v itself, but very different heuristics are needed:
   (a) For the choice of the first branch, some form of approximated *satisfiability*-probability must be maximised,
   (b) while the branching-variable must minimise some approximated tree-size for the worst case, the *unsatisfiable* case.
   In 2sJW the weights 2^{-k} are only motivated by satisfiability-probabilities, but are used for the choice of v itself.

[10] The look-ahead solvers `satz` and `march_pl` run through the variables once (actually also only considering "interesting" variables by some criterion), and so they do not compute r_2, but only an approximation. The only solver to completely compute r_2 is the `OKsolver` (while `satz` and `march_pl` search also for some r_3 reductions on selected variables).

[11] We do not care much here about the order of branching, since the algorithm is only effective on unsatisfiable problems, where the order does not matter (while on satisfiable problems local search is much faster).

2. Once two weights $w_F(v)$, $w_F(\overline{v})$ have been determined, one number (the projection) must be computed from this (to be maximised). 2sJW uses the sum, which, as demonstrated in [49], corresponds to minimising a *lower bound* on the DLL-tree-size—much better is the product $p_*(a, b) := a \cdot b$, which corresponds to minimising an *upper bound* on the tree-size.

So the improved heuristic (which nowadays, when extended appropriately to take the look-ahead into account, is the basis for all look-ahead solvers) chooses clause-weights $w_2, w_3, \ldots \in \mathbb{R}_{>0}$, from which the total weight

$$w_F(x) := \sum_k w_k \cdot \mathrm{ld}_F^k(x)$$

is determined, and chooses a variable v with maximal

$$p_*(w_F(v), w_F(\overline{v})) = w_F(v) \cdot w_F(\overline{v}).$$

The meaning of these weights is completely different from the argumentation in [35]: as mentioned, satisfiability-probabilities have no place here. The underlying distance measure is $\sum_k w_k \cdot \nu^k(F')$, where F' is the resulting clause-set after performing the branch-assignment and the subsequent r_k-reduction, while $\nu^k(F')$ is the number of *new* k-clauses in F'. When setting literal x to true, $\mathrm{ld}_F^k(\overline{x})$ is an "approximation" of the number of new clauses of length $k - 1$ (since in the clauses containing $\overline{x}$ this literal is removed).

The weights w_k^{OK} for the `OKsolver` have been experimentally determined as roughly 5^{-k}. Since the value of the first weight is arbitrary, the weights are rescaled to $w_2^{\mathrm{OK}} = 1$, obtaining then each new weight by multiplication with 1/5. Now w_2 for the `tawSolver` is a stand-in for the number of new 1-clauses, which are handled in the `OKsolver` by the look-ahead; accordingly it seems plausible that now w_2 needs a relatively higher weight. We rescale here the weights to $w_3 = 1$ (note that for the `tawSolver` the weight w_k concerns new clauses of length $k - 1$). Empirically we determined $w_2 = 4.85$, $w_4 = 0.354$, $w_5 = 0.11$, $w_6 = 0.0694$, and thereafter a factor of $\frac{1}{1.46}$; thus starting with w_2 the next weights are obtained by multiplying with (rounded) $1/4.85$, $1/2.82$, $1/3.22$, $1/1.59$, $1/1.46, \ldots$.

For the choice of the first branch there are two main schemes, as discussed in [49, Section 7.9]. Roughly, the target now is to get rid off (satisfy) as many short clauses as possible (since shorter clauses are bigger obstructions for satisfiability).[12] Both schemes amount to choose literal $x \in \{v, \overline{v}\}$ with $w'_F(x) \geq w'_F(\overline{x})$ for some weights w'_k. For the Franco-estimator we have $w'_k = -\log(1 - 2^{-k})$, while for the Johnson-estimator we have $w'_k = 2^{-k}$. In the `OKsolver` the Franco-estimator is used. But for the `tawSolver` with its emphasis on unsatisfiable instances, while the computation of the heuristic is very time-sensitive (much more so than for the `OKsolver`), actually just the same weights $w'_k = w_k$ are used.

As one can see from the data in Section 6, on ordinary van der Waerden problems the new heuristic yields a reduction in the size of the DLL-tree by a factor increasing from 2 to 5 for $t = 12, \ldots, 16$ (comparing `tawSolver`-2.6 with `tawSolver`-1.0), and for palindromic problems by a factor increasing from 5 to 20 for $t = 17, \ldots, 23$.[13] We do not present the data, but most of the reduction in node-count is due to the replacement of the sum as projection by the product (the optimised clause-weights only further improve the node reduction by at most 50% for the biggest instances, compared with a simple but reasonable scheme like 2^{-k}).

2.4. The implementation

The `tawSolver` is written in modern C++ (C++11, to be precise), with around 1000 lines of code, with complete input- and output-facilities, error handling and various compile-time options for implementations. The code is highly optimised for run-time speed, but at the same time expressing the concepts via appropriate abstractions, relying on the expressiveness of C++ both at the abstraction- and the implementation-level, so that the compiler can do a good job producing efficient code.

Look-ahead solvers are often "eager", that is, they represent the clause-set at each node of the DLL-tree in such a way, that the current ("residual") clause-set is visible to the solver, and precisely the current clauses can be accessed. On the other hand, conflict-driven solvers are all "lazy", that is, the initial clause-set is not updated, and the state of the current clause-set has to be inferred via the current assignment to the variables. The representation of the input clause-set F by the `tawSolver` now is "mostly lazy":

1. Assignments to variables are entered into a global array.
2. Via the usual occurrence lists, for each literal x one obtains access to all the clauses $C \in F$ with $x \in C$.
3. This representation of F is static (is not updated), and in this sense we have a lazy datastructure.
4. But the status of clauses, which is either inactive (when satisfied) or active, and their length (in the active case) is handled eagerly, by storing status and length for each clause and updating this information appropriately. So at each node, when running through the occurrence lists (still as in the input), for each clause we can see directly whether the clause is active and in this case its current length.

12 While for a good branching variable we want to *create* as many short clauses as possible (via setting literals to false)!

13 `tawSolver`-2.6 additionally has the implementation improved, so that nodes are processed now twice as fast as with `tawSolver`-1.0.

5. When doing an assignment, the clause-lengths are updated: if a literal is falsified in a clause, the length is decreased by one, and if a literal is satisfied, the status of the clause is set to inactive.
6. For each active clause containing a variable which is assigned, there is exactly one change (either decrease in length or going from active to inactive). This change is entered into a change-list.
7. When backtracking, the assignment is simply undone by going through the change-list in reverse order, and undoing the changes to the clauses.

No counters are maintained for the literal degrees $\mathrm{ld}^k(x)$. Instead, the heuristic is computed by running through all literal occurrences in the original input for the unassigned literals, and adding the contributions of the clauses which are still active (this is the use of maintaining the length of a clause).

When doing unit-clause propagation, the basic choice is whether performing a BFS search, by using a first-in-first-out strategy for the processing of derived unit-clauses, of a DFS search, using a last-in-first-out strategy. BFS is slightly easier to implement, but on the palindromic vdW-instances needs roughly 10% more unit-clauses to propagate,[14] while on ordinary vdW-instances it uses less propagations, though the difference is less than 2%, and thus DFS is the default. This can also be motivated by the consideration that newly derived unit-clauses can be considered to be "more expensive", and thus should be treated as soon as possible.

Look-ahead solvers in general rely on the distance for branch-evaluation to be positive, while a zero value should indicate that a special reduction can be performed. And indeed, when counting new clauses, the weighted sum being zero means that an autarky has been found, a partial assignment not creating new clauses, which means that all touched clauses are satisfied; see [42].[15] Thus starting with the `OKsolver`, look-ahead solvers looked out for such autarkies, and applied them when found [31,49]. Now for a zero-look-ahead solver like the `tawSolver`, these autarkies are just pure literals (only occurring in one sign, not in the other). Their elimination causes a slight run-time increase, without changing much anything else, and so by default they are not eliminated but not chosen for branching (if there are still non-pure literals left).[16]

2.5. The optimal projection: the τ-function

In [49] it is shown that the τ-function is the best generic projection in the following sense:

- The τ-function is defined for arbitrary tuples $a \in \mathbb{R}^n$, $n \in \mathbb{N}$, namely $\tau(a) \in \mathbb{R}_{>0}$ is the unique $x \geq 1$ such that $\sum_{i=1}^{n} x^{-a_i} = 1$.
- This projection induces a linear order on the set of all such "branching tuples" a (of arbitrary length) by defining $a \leq b$ if $\tau(a) \leq \tau(b)$; here "$a \leq b$" means that a is better than b.
- Theorem 7.5.3 in [49] shows that when imposing some general consistency-constraints on the comparison of branching tuples (where it is of importance that branching tuples can have arbitrary length), there is precisely one such linear order on the set of branching tuples, namely the one induced by τ.

Now specific solvers might have a special built-in bias, and, more importantly, the theorem is not applicable when considering only branching tuples of length 2 (as it is the case for ordinary Boolean SAT solving). But nevertheless, considering the τ-function as projection (more precisely, since we maximised projection values, $1/\tau$ is used) is an interesting option, and leads to the τ`awSolver`-2.6 (with "τ" in place of "`t`"):

$$p_\tau(w_F(v), w_F(\overline{v})) := 1/\tau(w_F(v), w_F(\overline{v})).$$

In Section 6 we see that τ`awSolver`-2.6 is faster than `tawSolver`-2.6 on large palindromic problems due to a much reduced node-count, but on ordinary problems the node-count stays basically the same, and then the overhead for computing p_τ makes the τ`awSolver`-2.6 slower.

The weights for τ`awSolver`-2.6 have been empirically determined as $w_2 = 7$, $w_4 = 0.31$, $w_5 = 0.19$, and then a factor of $\frac{1}{1.7}$; so starting with w_2 the next weights are obtained by multiplying with $1/7, 1/3.22, 1/1.63, 1/1.7, \ldots$.[17]

3. Computational results on $\mathrm{w}(2; 3, t)$

This section is concerned with the numbers $\mathrm{w}(2; 3, t)$. The discussion of the computation of $\mathrm{w}(2; 3, 19)$ is the subject of Section 3.1. Conjectures on the values of $\mathrm{w}(2; 3, t)$ for $20 \leq t \leq 30$ are presented in Section 3.2, and also further lower bounds for $31 \leq t \leq 39$ are given there. Finally in Section 3.3, we update the conjecture on the (quadratic) growth of $\mathrm{w}(2; 3, t)$.

[14] The final result is uniquely determined, but in general there are many ways to get there.

[15] The point about autarkies is that they can be applied satisfiability—equivalently.

[16] Elimination of pure literals is thus optional, but not possible if counting or enumerating all solutions. If enabled, elimination of pure literals does not happen iteratively, but only those literals with currently $w_F(x) = 0$ will be eliminated (by setting $\overline{x}$ to true), when running (once) through all the variables to determine the branching variable.

[17] We consider the values for the weights as reasonable all-round values. A deeper understanding, based on the theory developed in [49], is left for future investigations.

Table 2
Conjectured precise lower bounds for $\mathrm{w}(2; 3, t)$.

t	20	21	22	23	24	25	26	27	28	29	30
$\mathrm{w}(2; 3, t) \geq$	389	416	464	516	593	656	727	770	827	868	903

3.1. $\mathrm{w}(2; 3, 19) = 349$

The lower bound $\mathrm{w}(2; 3, 19) \geq 349$ was obtained by Kullmann [50] using local search algorithms and it could not be improved any further using these incomplete algorithms (because, as we now know, the bound is tight). An example of a good partition of the set $\{1, 2, \ldots, 348\}$ is as follows:

$$1^4 0 1^6 0 1^{18} 0 1^3 0 1^4 0 1^5 0 1^4 0 1^{11} 0 1^9 0 1^3 0 1^6 0 1^7 0 1^5 0 1^{14} 0 1^{16} 0 1 0 1^2 0^2 1^2 0 1^{15} 0 1^4 0 1^{12} 0$$
$$1^{15} 0 1^2 0 1^5 0 1^7 0 1^{10} 0 1^{13} 0 1^2 0 1^{15} 0 1^{12} 0 1^4 0 1^{15} 0 1^2 0^2 1^2 0 1 0 1^9 0 1^6 0 1^{14} 0 1^5 0 1^{14} 0 1^2.$$

To finish the search, i.e., to decide that a current lower bound of a certain van der Waerden number is exact, one might require many years of CPU-time. Discovering a new van der Waerden number has always been a challenge, as it requires to explore the search space completely, which has a size exponential in the number of variables in the corresponding satisfiability instance. To prove that an instance with n variables is unsatisfiable, the DLL algorithm has to implicitly enumerate all the 2^n cases. So the algorithm systematically explores all possible cases, however without actually explicitly evaluating all of them—herein lies the strength (and the challenge) for SAT solving.

In Section 1.1.2, we gave an overview on the area of distributing hard SAT problems from a general SAT perspective, and we are concerned here with method (ii)(a), applied to `tawSolver`. We find the simplest division of the computation of the search into parts, that have no inter-process communication among themselves, together with the observation of some patterns, very successful. Namely a level (depth) $L \in \mathbb{N}_0$ of the DLL-tree is chosen, where the level considers only the decisions (ignoring the variables inferred via unit-clause propagation), and the 2^L subtrees rooted at that level are distributed among the processors.

To show the unsatisfiability of $F(3, 19; 349)$, we have used `tawSolver`-1.0 and 2.2 GHz AMD Opteron 64-bit processors (200 of them) from the `cirrus` cluster at Concordia University for running the distributed branches of the DLL-tree. The value $L = 8$ was chosen, splitting the search space into $2^8 = 256$ independent parts (subtrees) $P_0, \ldots, P_{255}$. The total CPU-time of all processors together was roughly 196 years (the first part P_0 alone has taken roughly 60 years of CPU-time).[18] For the prediction of run-times for the sub-tasks, the following observation made in Ahmed [3] was used. Recall that for `tawSolver`-1.0 (Algorithm 1) the branching rule was to select a variable with maximal $w_F(v) + w_F(\overline{v}) = \sum_k (\mathrm{ld}_F(v) + \mathrm{ld}_F(\overline{v})) \cdot 2^{-k}$, where for the first branch $x \in \{v, \overline{v}\}$ with $\sum_k \mathrm{ld}_F(v) \cdot 2^{-k} \geq \sum_k \mathrm{ld}_F(\overline{v})$ is chosen. Now the observation is that the parts (sub-trees of the DLL-tree) $P_0, P_1, P_2, P_4, P_8, P_{16}, P_{32}, P_{64}, P_{128}$ are bigger than the other parts, and P_0 is the biggest.

Meanwhile our result $\mathrm{w}(2; 3, 19) = 349$ has been reproduced in [43], via an alternative SAT solving approach (see Section 1.1.1). At least at this time there seems to be no competitive alternative to SAT solving. See Section 6 for further remarks on SAT solving for these instances in general. It would be highly desirable to be able to substantially compress the resolution proofs obtained from the solver runs, so that a proof object would be obtained which could be verified by certified software (and hardware); see [17] for some recent literature.

3.2. Some new conjectures

In Table 2, we provide conjectured values of $\mathrm{w}(2; 3, t)$ for $t = 20, 21, \ldots, 30$. We have used the `Ubcsat` suite [68] of local-search based satisfiability algorithms for generating good partitions, which provide a proof of these lower bounds; see [6] for the certificates. In Section 6.2 we provide details of the algorithms used to find the good partitions. The characteristics of the searches were such that we believe these values to be optimal, namely with the right settings, these bounds can be found rather quickly, and in the past, all such conjectures turned out to be true (though, as discussed below, the situation gets weaker for $t = 29, 30$). However, since local search based algorithms are incomplete (they may fail to deliver a satisfying assignment, and hence a good partition when there exists one), it remains to prove exactness of the numbers using a complete satisfiability solver or some complete colouring algorithm.

We observe that for $t = 24, 25, \ldots, 30$ we have $\mathrm{w}(2; 3, t) > t^2$, which refutes the possibility that $\forall\, t : \mathrm{w}(2; 3, t) \leq t^2$, as suggested in [15], based on the exact values for $5 \leq t \leq 16$ known by then. Further (strict) lower bounds we found are in Table 3 (where now we think it is likely that these bounds can be improved; see [6] for the certificates).

[18] Comparing `tawSolver`-1.0 with `tawSolver`-2.6, as we can see in Table 9, the series of quotients q_i = old-time / new-time, for $t = 12, \ldots, 16$ is (rounded) 4.3, 5.6, 6.8, 9.4, 12.8. This can be approximated well by the law $q_{i+1} = 1.3 \cdot q_i$, which would yield for $t = 19$ the factor $12.8 \cdot 1.3^3 \approx 28.1$. So we would expect with `tawSolver`-2.6 at least a speed-up by a factor 20, which would reduce the 200 years to 10 years. Another approximation is obtained by considering Table 9: we see that for each step from t to $t + 1$ the run-time always increases by less than a factor of 10, while for $t = 17$ we use less than five days, which would yield at most 500 days for $t = 19$ with `tawSolver`-2.6.

Table 3
Further lower bounds for $\mathrm{w}(2;3,t)$.

t	31	32	33	34	35	36	37	38	39
$\mathrm{w}(2;3,t) >$	930	1006	1063	1143	1204	1257	1338	1378	1418

That we conjecture the data of Table 2 to be true, that is, that the used local-search algorithm is strong enough, while for the data of Table 3 that algorithm seems too weak to reach the solution, has the following background in the data: As we report in Section 6.2, in the range $24 \le t \le 33$ the local-search algorithm RoTS from the `Ubcsat` suite was found best-performing. This algorithm is used in an incremental fashion, initialising the search by known solutions for smaller n. This approach for $t = 28$, with a cut-off $5 \cdot 10^6$ rounds, found a solution for $n = 826$, and in 1000 (non-incremental) independent runs two solutions were found. But for $n = 827$ with cut-off 10^7 in 1000 runs and with cut-off $2 \cdot 10^7$ in 500 runs no solutions was found. From our experience this seems "pretty safe" for a conjecture. We are entering now a transition period. For $t = 29$ the iterative approach with cut-off $5 \cdot 10^6$ found the solution for $n = 867$, while cut-off 10^7 found no solution for $n = 868$ in 1000 runs. For $t = 30$ the iterative approach managed to find a solution for $n = 897$; restarting it with cut-off 10^8 found a solution for $n = 902$, while for $n = 903$ no solution with that cut-off was found in 300 runs. So we see that already $t = 30$ is stretching it. However for $t = 31$ the iterative approach with cut-off 10^8 only reached $n = 919$ (despite restarts), while we happen to have a palindromic solution for $n = 930$ (these are much easier to find; see Section 5.3). So here now we believe we definitely over-stretched the abilities of the algorithm.

3.3. A conjecture on the upper bound

An important theoretical question is the growth-rate of $t \mapsto \mathrm{w}(2;3,t)$. Although the precise relation "$\mathrm{w}(2;r,t) \le t^2$" has been invalidated by our results, quadratic growth still seems appropriate (see [50] for a more general conjecture on polynomial growth for van der Waerden numbers in certain directions of the parameter space; indeed in some directions linear growth is proven there):

Conjecture 3.1. *There exists a constant $c > 1$ such that $\mathrm{w}(2;3,t) \le ct^2$.*

See Conjecture 4.4 for a strengthening. To determine the current best guess for c, and to give some heuristic justification for Conjecture 3.1, we observe the known exact values and lower bounds, and we arrive at the following possible recursion:

$$\mathrm{w}(2;3,t) \le \mathrm{w}(2;3,t-1) + d(t-1),$$

for $4 \le t \le 39$ and some $d > 0$, with $\mathrm{w}(2;3,3) = 9$. So we make the Ansatz $\mathrm{w}(2;3,t) \le w_t := 9 + \sum_{i=3}^{t-1} d \cdot i$, for $t \ge 3$, where $d := \max_{t=4}^{39} \frac{\mathrm{w}(2;3,t)-\mathrm{w}(2;3,t-1)}{t-1}$; in case $\mathrm{w}(2;3,t)$ is not known, we use the lower bounds from Tables 2, 3. From our data we obtain $d = \frac{593-516}{23} = \frac{77}{23}$ (see the underlying report [6]). We have (geometric sum) $w_t = \frac{d}{2}t^2 - \frac{3}{2}dt + 9 - 2d < \frac{d}{2}t^2$, and so we obtain

$$\mathrm{w}(2;3,t) \le \frac{d}{2}t^2 = \frac{77}{46}t^2 < 1.675t^2,$$

which satisfies all data regarding $\mathrm{w}(2;3,t)$ presented so far.

4. Patterns in the good partitions

In this section, we investigate the set of all good partitions corresponding to certain van der Waerden numbers $\mathrm{w}(2;3,t)$ for patterns. As described in Section 1.1.1, the motivation behind this section is to obtain more problem-specific information on the solution-patterns, which may help to design heuristics to reduce search-space while computing specific van der Waerden numbers.

Let $S(t)$ denote the set of all binary strings each of which represents a good partition of the set $\{1, 2, \ldots, \mathrm{w}(2;3,t) - 1\}$. Generating $S(t)$ involves traversing the respective search space completely. Let $n_0(B)$, $n_1(B)$, and $n_{00}(B)$ be the number of zeros, ones, and double-zeros, respectively, in a bitstring B (note that three consecutive zeros are not possible in any $B \in S(t)$). Let EP1S(B) denote the sequence of powers of 1 in a bitstring B. Let $n_\mathrm{p}(B)$ and $n_\mathrm{v}(B)$ denote the number of peaks (local maxima) and valleys (local minima), respectively, in EP1S(B) (not necessarily strict). For example, for the compact bitstring $1^8001^601^301^101^3001^501^801^5001^301^101^301^6001^8$ (with $n_0 = 16, n_1 = 60$ and $n_{00} = 4$), we have the following EP1S, with p and v, marking peaks and valleys, respectively, corresponding to changes in magnitudes.

$$\begin{array}{ccccccccccccc} 8 & 6 & 3 & 1 & 3 & 5 & 8 & 5 & 3 & 1 & 3 & 6 & 8 \\ \mathrm{p} & & & \mathrm{v} & & & \mathrm{p} & & & \mathrm{v} & & & \mathrm{p} \end{array}.$$

And for $B = 1^101^101^201^201^301^3$ we have $n_0(B) = 5$, $n_1(B) = 12$, $n_{00}(B) = 0$, while there is one valley followed by one peak, and thus $n_\mathrm{v}(B) = n_\mathrm{p}(B) = 1$.

Table 4
Zeros in good partitions of $\{1, 2, \ldots, \mathrm{w}(2; 3, t) - 1\}$.

$\mathrm{w}(2; 3, t)$	$(\min\{n_0(B) : B \in S(t)\}, \max\{n_0(B) : B \in S(t)\})$	$\max\{n_{00}(B) : B \in S(t)\}$
$\mathrm{w}(2; 3, 3) = 9$	$(4, 4)$	2
$\mathrm{w}(2; 3, 4) = 18$	$(6, 6)$	2
$\mathrm{w}(2; 3, 5) = 22$	$(7, 9)$	2
$\mathrm{w}(2; 3, 6) = 32$	$(8, 10)$	4
$\mathrm{w}(2; 3, 7) = 46$	$(11, 12)$	3
$\mathrm{w}(2; 3, 8) = 58$	$(14, 14)$	1
$\mathrm{w}(2; 3, 9) = 77$	$(16, 16)$	4
$\mathrm{w}(2; 3, 10) = 97$	$(19, 21)$	5
$\mathrm{w}(2; 3, 11) = 114$	$(19, 22)$	5
$\mathrm{w}(2; 3, 12) = 135$	$(22, 22)$	1
$\mathrm{w}(2; 3, 13) = 160$	$(25, 29)$	5
$\mathrm{w}(2; 3, 14) = 186$	$(29, 29)$	4

4.1. Number of 0's and 00's

In this section, we determine the minimal and maximal numbers of zeros in good partitions ($\min\{n_0(B) : B \in S(t)\}$ resp. $\max\{n_0(B) : B \in S(t)\}$), and the maximal numbers $\max\{n_{00}(B) : B \in S(t)\}$ of double-zeros, for $3 \le t \le 14$. Note that the zeros mark the elements of the first block of the good partitions, which have to avoid arithmetic progressions of size 3, and thus there are far fewer zeros than ones. Observations in Table 4 lead us to Conjectures 4.1 and 4.2.

It seems that there is little variation concerning the total number of zeros:

Conjecture 4.1. *There exists a constant $c > 0$ such that $|n_0(B) - n_0(B')| \le ct$, $\forall B, B' \in S(t)$ with $t \ge 3$.*

And there seem to be very few consecutive zeros:

Conjecture 4.2. *There exists a constant $c > 0$ such that $n_{00}(B) < ct$, $\forall B \in S(t)$ with $t \ge 3$.*

4.2. Number of 1's

In this section, we determine $T = \min\{n_{\mathrm{p}}(\mathrm{EP1S}(B)) + n_{\mathrm{v}}(\mathrm{EP1S}(B)) : B \in S(t)\}$, as well as minimum and maximum values of $n_1(B)$ over all $B \in S(t)$. The observations in Table 5 lead us to Conjectures 4.3, 4.4, and Questions 4.1 and 4.2.

Again, there seems little variation concerning the total number of ones:

Conjecture 4.3. *There exists a constant $c > 0$ such that $|n_1(B) - n_1(B')| \le ct$, $\forall B, B' \in S(t)$ with $t \ge 3$.*

Stronger than Conjecture 4.3, the number of ones seems very close to the vdW-number for the previous t:

Conjecture 4.4. *There exists a constant $c > 0$ such that $|\mathrm{w}(2; 3, t - 1) - n_1(B)| < ct$, $\forall B \in S(t)$.*

This conjecture also implies the earlier conjecture on the quadratic growth of $\mathrm{w}(2; 3, t)$:

Lemma 4.1. *Conjecture 4.4 implies Conjecture 4.3 and Conjecture 3.1.*

Proof. Conjecture 4.3 follows by the triangle inequality. Conjecture 3.1 follows, if for t large enough we can show $n_0(B) \le n_1(B)$ for all $B \in S(T)$, and this is a special case of Szemerédi's Theorem [67], which for arithmetic progressions of size 3 was already proven in [60],[19] namely that the relative size of maximum independent subsets of the hypergraph of arithmetic progressions of size 3 in the numbers $1, \ldots, t$ goes to 0 with $t \to \infty$. $\square$

We turn to the growth of the number of peaks and valleys:

Question 4.1. For each positive constant c does there exist a t' such that for all $t \ge t'$, $n_{\mathrm{p}}(\mathrm{EP1S}(B)) + n_{\mathrm{v}}(\mathrm{EP1S}(B)) \ge ct$, $(t \ge 3)$ $\forall B \in S(t)$? (We conjecture yes).

We conclude with the observation, that for $t > 3$ there do not seem to be long plateaus for the numbers of ones:

Question 4.2. Is there a good partition $B \in S(t)$, $(t \ge 4)$ with 3 consecutive numbers equal in $\mathrm{EP1S}(B)$? (Note that, for $t = 3$, the partition $1^1 0 1^1 0 0 1^1 0 1^1$ has four consecutive exponents, which are the same.)

[19] See http://rothstheorem.wikidot.com/on-certain-sets-of-integers.

Table 5
Selected good-partitions of $\{1, 2, \dots, \mathrm{w}(2; 3, t) - 1\}$.

$\mathrm{w}(2; 3, t)$	A good partition B corresponding to T	T	$\min\{n_1(B) : B \in S(t)\}, \max\{n_1(B) : B \in S(t)\}$
$\mathrm{w}(2; 3, 3) = 9$	1^2001^200 `(2 2)`	1	$(4, 4)$
$\mathrm{w}(2; 3, 4) = 18$	$1^3001^101^3001^101^3$ `(3 1 3 1 3)`	5	$(11, 11)$
$\mathrm{w}(2; 3, 5) = 22$	$001^3001^101^4001^401^1$ `(3 1 4 4 1)`	4	$(12, 14)$
$\mathrm{w}(2; 3, 6) = 32$	$01^5001^501^3001^5001^5$ `(5 5 3 5 5)`	3	$(21, 23)$
$\mathrm{w}(2; 3, 7) = 46$	$1^101^101^401^201^501^401^1001^301^501^201^50$ `(1 1 4 2 5 4 1 3 5 2 5)`	8	$(33, 34)$
$\mathrm{w}(2; 3, 8) = 58$	$1^401^201^401^101^401^301^5001^501^301^401^101^401^201^1$ `(4 2 4 1 4 3 5 5 3 4 1 4 2 1)`	12	$(43, 43)$
$\mathrm{w}(2; 3, 9) = 77$	$1^8001^601^301^101^3001^501^801^5001^301^101^301^6001^8$ `(8 6 3 1 3 5 8 5 3 1 3 6 8)`	5	$(60, 60)$
$\mathrm{w}(2; 3, 10) = 97$	$1^701^401^201^5001^2001^701^401^801^101^801^4001^6001^2001^801^9$ `(7 4 2 5 2 7 4 8 1 8 4 6 2 8 9)`	13	$(75, 77)$
$\mathrm{w}(2; 3, 11) = 114$	$01^{10}01^4001^601^{10}01^2001^901^601^101^9001^1001^{10}01^6001^{10}01^{10}$ `(10 4 6 10 2 9 6 1 9 1 10 6 10 10)`	11	$(91, 94)$
$\mathrm{w}(2; 3, 12) = 135$	$1^901^801^901^201^301^101^701^20101^301^{11}0^2$ $1^{11}01^30101^201^701^101^301^201^901^801^9$ `(9 8 9 2 3 1 7 2 1 3 11 11 3 1 2 7 1 3 2 9 8 9)`	17	$(112, 112)$
$\mathrm{w}(2; 3, 13) = 160$	$1^101^601^{12}01^4001^{11}001^601^{10}01^201^401^{11}01^10$ $1^601^901^201^301^701^{10}01^1001^501^{12}01^501^401^2$ `(1 6 12 4 11 6 10 2 4 11 1 6 9 2 3 7 10 1 5 12 5 4 2)`	15	$(130, 134)$

4.3. *How can it help for SAT solving?*

If one of the above conjectures (or some other conjecture) turns out to be true, and if moreover the numerical constants have good estimates, then they can be used to restrict the search space. When using a general purpose SAT solver, this can be achieved by adding further constraints. It seems however that these constraints do not help with the search, even if we assume that they are true, since they are too difficult to handle for the solver. It seems the problem is that these constraints do not mix well with the original problem formulation, and a deeper integration is needed. Such an integration was achieved in the case of the palindromic constraint, which is the subject of the following section—here an organic new problem formulation could be established, where the additional restriction does not appear as an "add-on", but establishes a natural new problem class.

5. Palindromes

Recall Definitions 1.1, 1.2:

1. for given $k \in \mathbb{N}$ (the number of "colours"),
2. $t_0, \dots, t_{k-1}$ (the lengths of arithmetic progressions),
3. and $n \in \mathbb{N}$ (the number of vertices)

we consider block partitions $(P_0, \dots, P_{k-1})$ of $\{1, \dots, n\}$ such that no P_i contains an arithmetic progression of length t_i—these are the "good partitions", and $\mathrm{w}(k; t_0, \dots, t_{k-1}) \in \mathbb{N}$ is the smallest n such that no good partition exists. If $(P_0, \dots, P_{k-1})$ is a good partition of $\{1, \dots, n\}$ w.r.t. $t_0, \dots, t_{k-1}$, then for $1 \le n' \le n$ we obtain a good partition of $\{1, \dots, n'\}$ w.r.t. $t_0, \dots, t_{k-1}$ by just removing vertices $n' + 1, \dots, n$ from their blocks. Thus $\mathrm{w}(k; t_0, \dots, t_{k-1})$ completely determines for which $n \in \mathbb{N}$ good partitions exist, namely exactly for $n < \mathrm{w}(k; t_0, \dots, t_{k-1})$.

Definition 5.1. For $n \in \mathbb{N}$ let $\mathrm{m}_n : \{1, \dots, n\} \to \{1, \dots, n\}$ (with "m" like "mirror") defined by $\mathrm{m}_n(v) := n + 1 - v$. This map is extended to $S \subseteq \{1, \dots, n\}$ as usual: $\mathrm{m}_n(S) := \{\mathrm{m}_n(v) : v \in S\}$.

Now if $(P_0, \dots, P_{k-1})$ is a good partition w.r.t. n, then also $(\mathrm{m}_n(P_0), \dots, \mathrm{m}_n(P_{k-1}))$ is a good partition w.r.t. n. So it is of interest to consider self-symmetric partitions (with $\mathrm{m}_n(P_i) = P_i$ for all i):

Definition 5.2. A *good palindromic partition* of $\{1, \dots, n\}$ w.r.t. parameters $t_0, \dots, t_{k-1}$, where $n, t_0, \dots, t_{k-1} \in \mathbb{N}$, is a good partition of $\{1, \dots, n\}$ w.r.t. $t_0, \dots, t_{k-1}$ such that for all $j \in \{0, \dots, k-1\}$ holds $\mathrm{m}_n(P_j) = P_j$.

We call these special good partitions "palindromic", since a block partition can be represented as a string of numbers over $\{0, \dots, k-1\}$, and then the block partition is palindromic iff the string is a palindrome (reads the same forwards and backwards). For example, the string 01^2001^20 represents a good palindromic partition for $k = 2, t_0 = t_1 = 3$ and $n = 8$, namely $(\{1, 4, 5, 8\}, \{2, 3, 6, 7\})$, and so does $(\{1, 3, 6, 8\}, \{2, 4, 5, 7\})$, represented by 0101^2010, while $(\{1, 2, 5, 6\}, \{3, 4, 7, 8\})$, represented by 001^2001^2, is a good partition which is not palindromic.

For given k and $t_0, \dots, t_{k-1}$ again we want to completely determine (in theory) for which n do good palindromic partitions exist and for which not. The key is the following observation (which follows also from Lemmas 5.2, 5.3).

Lemma 5.1. *Consider fixed $k, t_0, \dots, t_{k-1}$, and $n \geq 3$. From a good palindromic partition $(P_0, \dots, P_{k-1})$ of $\{1, \dots, n\}$ we obtain a good palindromic partition $(P'_0, \dots, P'_{k-1})$ of $\{1, \dots, n-2\}$ by removing vertices $1, n$ and replacing the remaining vertices v by $v - 1$, that is, $P'_i := \{v - 1 : v \in P_i \setminus \{1, n\}\}$.*

Proof. The notion of a good partition of $\{1, \dots, n\}$ w.r.t. $\mathrm{w}(k; t_0, \dots, t_{k-1})$, as defined in Definition 1.2, can be generalised to good partitions of arbitrary $T \subseteq \mathbb{Z}$ by demanding that for every block partition $(P_0, \dots, P_{k-1})$ of T into k parts no part P_j contains an arithmetic progression of size t_j. In the remainder of the proof we omit the "w.r.t. $t_0, \dots, t_{k-1}$".

If T has a good partition, then also every subset has a good partition, by restricting the blocks accordingly, and for every $d \in \mathbb{Z}$ also $d + T = \{d + x : x \in T\}$ has a good partition, by shifting the blocks as well.

We can also generalise the notion of a good palindromic partition to intervals $T = \{a, a+1, \dots, b\} \subset \mathbb{Z}$ for $a < b$, defining now the mirror-map $m_{a,b} : T \to T$ via $v \in T \mapsto b + a - v$ (m_n in Definition 5.1 is the special case $m_n = m_{1,n}$).

Again, if T has a good palindromic partition, then $d + T$ for $d \in \mathbb{Z}$ has as well. But for subsets of T we can only consider sub-intervals $T' = \{a', \dots, b'\}$, where from both sides we have taken away equal amounts. That is, for $a \leq a' < b' \leq b$ with $a' - a = b - b'$ we have, that from a good palindromic partition for T we can obtain a good palindromic partition for T' (by just restricting the blocks).

So from a good palindromic partition of $\{1, \dots, n\}$ we obtain a good palindromic partition of $\{1, \dots, n-2\}$ by first restricting to $\{2, \dots, n-1\}$ and then shifting by -1. □

Corollary 5.1.1. *If there is no good palindromic partition of $\{1, \dots, n\}$, then there is no good palindromic partition of $\{1, \dots, n + 2 \cdot i\}$ for all $i \in \mathbb{N}_0$.*

Proof. If there would be a good palindromic partition of $\{1, \dots, n + 2 \cdot i\}$, then by repeated applications of Lemma 5.1 we would obtain a good palindromic partition of $\{1, \dots, n\}$. □

Since by van der Waerden's theorem we know there always exists some n such that for all $n' \geq n$ no good palindromic partition exists, we get that the existence of good palindromic partitions w.r.t. fixed $t_0, \dots, t_{k-1}$ is determined by two numbers, the endpoint p of "always exists" resp. q of "never exists", with alternating behaviour in the interval in-between:

Corollary 5.1.2. *Consider the maximal $p \in \mathbb{N}_0$ such that for all $n \leq p$ good palindromic partitions exist, and the minimal $q \in \mathbb{N}$ such that for all $n \geq q$ no good palindromic partitions exist. Then $q - p$ is an odd natural number, where no good palindromic partition exists for $p + 1$, but $p + 2$ again has a good palindromic partition, and so on alternately, until from q on no good palindromic partition exists anymore.*

Proof. By Corollary 5.1.1 there is no good palindromic partition for $p + 1 + 2i$ and all $i \in \mathbb{N}_0$. Now for the first $i \in \mathbb{N}_0$, such that $p+2+2i$ has no good palindromic partition, we let $q' := (p+2+2i)-1$. We have a good palindromic partition for $q-1$ by definition of i (as the smallest such i) resp. in case of $i = 0$ by definition of p. We have $q' + 2j = (p + 2 + 2i) - 1 + 2j = p+1+2(i+j)$ for $j \in \mathbb{N}_0$, and thus there is no good palindromic partition for $q'+2j$. And if there would be a good palindromic partition for $q' + 1 + 2j = p+2+2i+2j$, then by Corollary 5.1.1 there would be a good palindromic partition for $p+2+2i$. So we have $q' = q$. □

Definition 5.3. The *palindromic van-der-Waerden number* $\mathrm{pdw}(k; t_0, \dots, t_{k-1}) \in \mathbb{N}_0^2$ is defined as the pair (p, q) such that p is the largest $p \in \mathbb{N}_0$ with the property, that for all $1 \leq n \leq p$ there exists a good palindromic partition of $\{1, \dots, n\}$, while q is the smallest $q \in \mathbb{N}$ such that for no $n \geq q$ there exists a good palindromic partition of $\{1, \dots, n\}$. We use $\mathrm{pdw}(k; t_0, \dots, t_{k-1})_1 = p$ and $\mathrm{pdw}(k; t_0, \dots, t_{k-1})_2 = q$. So $0 \leq \mathrm{pdw}(k; t_0, \dots, t_{k-1})_1 < \mathrm{pdw}(k; t_0, \dots, t_{k-1})_2 \leq \mathrm{w}(k; t_0, \dots, t_{k-1})$.

The *palindromic gap* is

$$\mathrm{pdg}(k; t_0, \ldots, t_{k-1}) := \mathrm{w}(k; t_0, \ldots, t_{k-1}) - \mathrm{pdw}(k; t_0, \ldots, t_{k-1})_2 \in \mathbb{N}_0,$$

while the *palindromic span* is defined as

$$\mathrm{pds}(k; t_0, \ldots, t_{k-1}) := \mathrm{pdw}(k; t_0, \ldots, t_{k-1})_2 - \mathrm{pdw}(k; t_0, \ldots, t_{k-1})_1 \in \mathbb{N}.$$

To certify that $\mathrm{w}(k; t_0, \ldots, t_{k-1}) = n$ holds means to show that there exists a good partition of $\{1, \ldots, n-1\}$ and that there is no good partition of n. For palindromic number-pairs we need to double the effort:

Theorem 5.1. *To certify that* $\mathrm{pdw}(k; t_0, \ldots, t_{k-1}) = (p, q)$ *holds, exactly the following needs to be shown for (arbitrary)* $p \in \mathbb{N}_0$, $q \in \mathbb{N}$ *with* $p < q$:

(i) *there are good palindromic partitions of* $\{1, \ldots, p-1\}$ *and* $\{1, \ldots, q-1\}$ *w.r.t.* $t_0, \ldots, t_{k-1}$;
(ii) *there are no good palindromic partitions of* $\{1, \ldots, p+1\}$ *and* $\{1, \ldots, q+1\}$ *w.r.t.* $t_0, \ldots, t_{k-1}$.

Proof. The given conditions are necessary for $\mathrm{pdw}(k; t_0, \ldots, t_{k-1}) = (p, q)$ by the defining properties of p and q. We show that they are sufficient to establish $\mathrm{pdw}(k; t_0, \ldots, t_{k-1}) = (p, q)$. First we have by Corollary 5.1.1 that $q - p$ is odd, since otherwise $p + 1$ having no good palindromic partitions would yield that $q - 1$ would have no good palindromic partition. Then, again by Corollary 5.1.1, all $n \geq q + 1$ have no good palindromic partition, while all $n \leq p - 1$ have good palindromic partitions. By Corollary 5.1.2 we must now have $\mathrm{pdw}(k; t_0, \ldots, t_{k-1}) = (p, q)$. □

5.1. Palindromic vdW-hypergraphs

Recall that a finite hypergraph G is a pair $G = (V, E)$, where V is a finite set (of "vertices") and E is a set of subsets of V (the "hyperedges"); one writes $V(G) := V$ and $E(G) := E$. The essence of the (finite) van der Waerden problem (which we will now often abbreviate as "vdW-problem") is given by the hypergraphs $\mathrm{ap}(t, n)$ of arithmetic progressions with progression length $t \in \mathbb{N}$ and the number $n \in \mathbb{N}_0$ of vertices:

- $V(\mathrm{ap}(t, n)) := \{1, \ldots, n\}$
- $E(\mathrm{ap}(t, n)) := \{p \subseteq \{1, \ldots, n\} : p \text{ arithmetic progression of length } t\}$.

For example $\mathrm{ap}(3, 5) = (\{1, 2, 3, 4, 5\}, \{\{1, 2, 3\}, \{2, 3, 4\}, \{1, 3, 5\}, \{3, 4, 5\}\})$. Considering hypergraphs, the reader might wonder how determination of vdW-numbers fits with hypergraph colouring. While the determination of diagonal vdW-numbers is an ordinary hypergraph colouring problem, for general vdW-numbers a more general concept of hypergraph colouring is to be used, involving the simultaneous colouring of several hypergraphs in the following sense: The diagonal vdW-number $\mathrm{w}(k; t, \ldots, t)$ for $k, t \in \mathbb{N}$ is the smallest $n \in \mathbb{N}$ such that the hypergraph $\mathrm{ap}(t, n)$ is not k-colourable, where in general a k-colouring of a hypergraph G is a map $f : V(G) \to \{1, \ldots, k\}$ such that no hyperedge is "monochromatic", that is, every hyperedge gets at least two different values by f. For the general vdW-number $\mathrm{w}(k; t_0, \ldots, t_{k-1})$ we now consider for each colour $i \in \{0, \ldots, k-1\}$ the hypergraph $\mathrm{ap}(t_i, n)$, and we forbid (to formulate "good partition") for each i that there is a hyperedge in $\mathrm{ap}(t_i, n)$ monocoloured with colour i (while we do not care about the other colours here). Accordingly the SAT-encoding of "$\mathrm{w}(2; 3, t) > n$?", as discussed in Section 1.1, exactly consists of the two hypergraphs $\mathrm{ap}(3, n)$ and $\mathrm{ap}(t, n)$ represented by positive resp. negative clauses.

The task now is to define the palindromic version $\mathrm{pdap}(t, n)$ of the hypergraph of arithmetic progressions, so that for diagonal palindromic vdW-numbers $\mathrm{pdw}(k; t, \ldots, t) = (p, q)$ we have, that q is minimal for the condition that for all $n \geq q$ the hypergraph $\mathrm{pdap}(t, n)$ is not k-colourable, while p is maximal for the condition that for all $n \leq p$ the hypergraph is k-colourable. Furthermore we should have that for two-coloured problems (i.e., $k = 2$) the SAT-encoding of "$\mathrm{pdw}(2; t_0, t_1) > n$?" (satisfiable iff the answer is yes) consists exactly of the two hypergraphs $\mathrm{pdap}(t_0, n)$, $\mathrm{pdap}(t_1, n)$ represented by positive resp. negative clauses (while for more than two colours generalised clause-sets can be used; see [51]).

Consider fixed $t \in \mathbb{N}$ and $n \in \mathbb{N}_0$. Obviously $\mathrm{pdap}(t, 0) := \mathrm{ap}(t, 0) = (\{\}, \{\})$, and so assume $n \geq 1$. Recall the permutation $m = m_n$ of $\{1, \ldots, n\}$ from Definition 5.1. As every permutation, m induces an equivalence relation $\sim$ on $\{1, \ldots, n\}$ by considering the cycles, which here, since m is an involution (self-inverse), just has the equivalence classes $\{1, \ldots, n\}/\sim = \{\{v, f(v)\}\}_{v \in \{1,\ldots,n\}}$ of size 1 or 2 comprising the elements and their images. Note that m has a fixed point (an equivalence class of size 1) iff n is odd, in which case the unique fixed point is $\frac{n+1}{2}$. The idea now is to define $m' : \{1, \ldots, n\} \to \{1, \ldots, n\}$, which chooses from each equivalence class one representative (so $m'(v) \in \{v, m(v)\}$ and $v \sim w \Leftrightarrow m'(v) = m'(w)$), and to let $\mathrm{pdap}(t, n)$ be the image of $\mathrm{ap}(t, n)$ under m', that is, $(m'(V(\mathrm{ap}(t, n))), \{m'(H)\}_{H \in E(\mathrm{ap}(t,n))})$. Naturally we choose $m'(v)$ to be the smaller of v and $m(v)$. Now it occurs that images of arithmetic progressions under m' can subsume each other, i.e., for $H_1, H_2 \in E(\mathrm{ap}(t, n))$ with $H_1 \neq H_2$ we can have $m'(H_1) \subset m'(H_2)$, and so we define $\mathrm{pdap}(t, n)$ as the image of $\mathrm{ap}(t, n)$ under m', where also all subsumed hyperedges are removed (so we only keep the minimal hyperedges under the subset-relation).

Definition 5.4. For $t \in \mathbb{N}$ and $n \in \mathbb{N}_0$ the hypergraph $\mathrm{pdap}(t, n)$ is defined as follows:

- $V(\mathrm{pdap}(t, n)) := \{1, \dots, \lceil \frac{n}{2} \rceil\}$
- $E(\mathrm{pdap}(t, n))$ is the set of minimal elements w.r.t. $\subseteq$ of the set of $m'_n(H)$ for $H \in E(\mathrm{ap}(t, n))$, where $m'_n : \{1, \dots, n\} \to V(\mathrm{pdap}(t, n))$ is defined by $m'_n(v) := v$ for $v \le \lceil \frac{n}{2} \rceil$ and $m'_n(v) := n + 1 - v$ for $v > \lceil \frac{n}{2} \rceil$.

Using $\mathrm{ap}(3, 5) = (\{1, 2, 3, 4, 5\}, \{\{1, 2, 3\}, \{2, 3, 4\}, \{1, 3, 5\}, \{3, 4, 5\}\})$ as above, we have $m'(\{1, 2, 3\}) = \{1, 2, 3\}$, $m'(\{2, 3, 4\}) = \{2, 3\}$, $m'(\{1, 3, 5\}) = \{1, 3\}$ and $m'(\{3, 4, 5\}) = \{1, 2, 3\}$, whence $\mathrm{pdap}(3, 5) = (\{1, 2, 3\}, \{\{1, 3\}, \{2, 3\}\})$.

Lemma 5.2. *Consider $t \in \mathbb{N}$ and $n \in \mathbb{N}_0$. The hypergraph $\mathrm{pdap}(t, n)$ is embedded into the hypergraph $\mathrm{pdap}(t, n + 2)$ via the map $e : V(\mathrm{pdap}(t, n)) \to V(\mathrm{pdap}(t, n + 2))$ given by $v \mapsto v + 1$.*

Proof. First we note that $|V(\mathrm{pdap}(t, n+2))| = |V(\mathrm{pdap}(t, n))| + 1$, and so the range of e is $V(\mathrm{pdap}(t, n+2)) \setminus \{1\}$. Let G be the hypergraph with vertex set $V(\mathrm{pdap}(t, n + 2)) \setminus \{1\}$, whose hyperedges are all those hyperedges $H \in E(\mathrm{pdap}(t, n + 2))$ with $1 \notin H$. We show that e is an (hypergraph-)isomorphism from $\mathrm{pdap}(t, n)$ to G, which proves the assertion.

Now obviously the underlying hypergraph $\mathrm{ap}(t, n)$ is embedded into the underlying $\mathrm{ap}(t, n + 2)$ via the underlying map $v \in V(\mathrm{ap}(t, n)) \mapsto v + 1 \in V(\mathrm{ap}(t, n + 2))$, where the image of this embedding is given by the hypergraph with vertex set $V(\mathrm{ap}(t, n + 2)) \setminus \{1, n + 2\}$, and where the hyperedges are those $H \in E(\mathrm{ap}(t, n + 2))$ with $1, n + 2 \notin H$. Since $m'_{n+2}(n + 2) = 1$ and $m'_n(v) = m'_{n+2}(v + 1) - 1$ for $v \in \{1, \dots, n\}$, the assertion follows from the fact that there are no hyperedges $H, H' \in E(\mathrm{ap}(t, n + 2))$ with $H \cap \{1, n + 2\} \neq \emptyset$, $H' \cap \{1, n + 2\} = \emptyset$ and $m'_{n+2}(H) \subset m'_{n+2}(H')$ (thus $m'_{n+2}(H')$ can only be removed from $\mathrm{pdap}(t, n + 2)$ by subsumptions already at work in $\mathrm{pdap}(t, n)$), and this is trivial since $1 \in m'_{n+2}(H)$ but $1 \notin m'_{n+2}(H')$. □

The SAT-translation of "Is there a good palindromic partition of $\{1, \dots, n\}$ w.r.t. t_0, t_1?" is accomplished similar to the translation of "$\mathrm{w}(2; t_0, t_1) > n$?", now using $\mathrm{pdap}(t_0, n)$, $\mathrm{pdap}(t_1, n)$ instead of $\mathrm{ap}(t_0, n)$, $\mathrm{ap}(t_1, n)$:

Lemma 5.3. *Consider $t_0, t_1 \in \mathbb{N}$, $t_0 \le t_1$, and $n \in \mathbb{N}_0$. Let the Boolean clause-set $F^{\mathrm{pd}}(t_0, t_1, n)$ be defined as follows:*

- *the variable-set is $\{1, \dots, \lceil \frac{n}{2} \rceil\}$ $(= V(\mathrm{pdap}(t_0, n)) = V(\mathrm{pdap}(t_1, n)))$;*
- *the hyperedges of $\mathrm{pdap}(t_0, n)$ are directly used as positive clauses;*
- *the hyperedges H of $\mathrm{pdap}(t_1, n)$ yield negative clauses $\{\overline{v}\}_{v \in H}$.*

Then there exists a good palindromic partition if and only if $F^{\mathrm{pd}}(t_0, t_1, n)$ is satisfiable, where the satisfying assignments are in one-to-one correspondence to the good palindromic partitions of $\{1, \dots, n\}$ w.r.t. (t_0, t_1). □

For more than two colours, Lemma 5.3 can be generalised by using generalised clause-sets, as in [51], and there one also finds the "generic translation", a general scheme to translate generalised clause-sets (with non-Boolean variables) into Boolean clause-sets (see also [52,53]).

5.2. Precise values

See Section 6.1 for details of the computation (see Table 6).

5.3. Conjectured values and bounds

For $28 \le t \le 39$ we have reasonable values on $\mathrm{pdw}(2; 3, t)$, which are given in Table 7, and which we believe to be exact for $t \le 35$. These values have been computed by local-search methods (see Section 6.2), and thus for sure we can only say that they present lower bounds. We obtain conjectured values for the palindromic span (which might however be too large or too small) and conjectured values for the palindromic gap (which additionally depend on the conjectured values from Section 3.2, while for $t \ge 31$ we only have the lower bounds from Section 3.2).

For the certificates for these lower bounds see [6].

5.4. Open problems

The relation between ordinary and palindromic vdW-numbers are of special interest:

- It seems the palindromic span can become arbitrarily large—also in relative terms? Perhaps the span shows a periodic behaviour, oscillating between small and large?
- Similar questions are to be asked for the gap. Does it attain value 0 infinitely often?

Do the hypergraphs $\mathrm{pdap}(t, n)$ have interesting properties (more basic than their chromatic numbers)? A basic exercise would be to estimate the number of hyperedges and their sizes. In the subsequent Section 6.1 we find data that SAT solvers behave rather different on palindromic vdW-problems (compared to ordinary problems). It seems that palindromic problems are more "structured" than ordinary problems—can this be made more precise? Perhaps the hypergraphs $\mathrm{pdap}(t, n)$ show characteristic differences to the hypergraphs $\mathrm{ap}(t, n)$, which could explain the behaviour of SAT solvers?

Table 6
Palindromic vdW-numbers pdw(2; 3, t).

t	pdw(2; 3, t)	pds(2; 3, t)	pdg(2; 3, t)
3	(6, 9)	3	0
4	(15, 16)	1	2
5	(16, 21)	5	1
6	(30, 31)	1	1
7	(41, 44)	3	2
8	(52, 57)	5	1
9	(62, 77)	15	0
10	(93, 94)	1	3
11	(110, 113)	3	1
12	(126, 135)	9	0
13	(142, 155)	13	5
14	(174, 183)	9	3
15	(200, 205)	5	13
16	(232, 237)	5	1
17	(256, 279)	23	0
18	(299, 312)	13	0
19	(338, 347)	9	2
20	(380, 389)	9	≥ 0
21	(400, 405)	5	≥ 11
22	(444, 463)	19	≥ 1
23	(506, 507)	1	≥ 9
24	(568, 593)	25	≥ 0
25	(586, 607)	21	≥ 49
26	(634, 643)	9	≥ 84
27	(664, 699)	35	≥ 71

Table 7
Conjectured palindromic vdW-numbers pdw(2; 3, t).

t	pdw(2; 3, t) $\geq$	pds(2; 3, t) $\sim$	pdg(2; 3, t) $\sim$
28	(728, 743)	15	84
29	(810, 821)	11	47
30	(844, 855)	11	48
31	(916, 931)	15	0
32	(958, 963)	5	44
33	(996, 1005)	9	59
34	(1054, 1081)	27	63
35	(1114, 1155)	41	50
36	(1186, 1213)	27	45
37	(1272, 1295)	23	44
38	(1336, 1369)	33	10
39	(1406, 1411)	5	8

5.5. Remarks on the use of symmetries

The heuristic use of symmetries for finding good partitions has been studied in [28,34,33] (while for symmetries in the context of general SAT solving see [61]). Especially we find there an emphasis on "internal symmetries", which are not found in the problem, but are imposed on the solutions.

The good palindromic partitions introduced in this section are more restricted in the sense, that they are based on the symmetries m from Section 5.1 of the clause-sets F expressing "$\mathrm{w}(k; t_0, t_1, \ldots, t_{k-1}) > n$?" (i.e., we have $m(F) = F$; recall Section 1.1), which then is imposed as an internal symmetry on the potential solution by demanding that the solutions be self-symmetric. In [28] "reflection symmetric" certificates are mentioned, which for even n are the same as good palindromic partitions, however for odd n they ignore vertex 1, not the mid-point $\lceil \frac{n}{2} \rceil$ as we do. This definition in [28] serves to maintain monotonicity (i.e., a solution for $n + 1$ yields a solution for n, while we obtain one only for $n - 1$ (Lemma 5.1)). We believe that palindromicity is a more natural notion, but further studies are needed here to compare these two notions.

Other internal symmetries used in [28,34,33] are obtained by modular additions and multiplications (these are central to the approaches there), based on the method from [57] for constructing lower bounds for diagonal vdW-numbers. No generalisations are known for the mixed problems we are considering.

Finally we wish to emphasise that we do not consider palindromicity as a mere heuristic for finding lower bounds, but we get an interesting variation of the vdW-problem in its own right, which hopefully will help to develop a better understanding of the vdW-problem itself in the future.

6. Experiments with SAT solvers

We conclude by summarising the experimental results and insights gained by running SAT solvers on the instances considered in this paper. All the solvers (plus build environments), generators and the data are available in the OKlibrary [48]; see [6] for more information.

For determining unsatisfiability we consider complete SAT solvers in Section 6.1. In general, for (ordinary) vdW-problems look-ahead solvers seem to perform better than conflict-driven solvers, while for palindromic problems it seems to be the opposite. However tawSolver-2.6 is the best (single) solver for both classes.

The hybrid approach, Cube & Conquer, was developed precisely on the instances of this paper, as discussed in [30] (further developments one finds in [69]). This approach is third-best on vdW-problems (after tawSolver-2.6 and τawSolver-2.6), and best on palindromic vdW-problems (before τawSolver-2.6 and tawSolver-2.6).

We conclude this section in Section 6.2 by remarks on incomplete SAT solvers, used to obtain lower bounds (determine satisfiability).

For the experiments we used a 64-bit workstation with 32 GB RAM and Intel i5-2320 CPUs (6144 kB cache) running with 3 GHz, where we only employed a single CPU.

6.1. Complete solvers

Complete SAT solvers exist in mainly two forms, "look-ahead solvers" and "conflict-driven solvers"; see [56,31] for general overviews on these solver paradigms. Besides the tawSolver (see Section 2), for our experimentation we use the following (publicly available) complete solvers, which give a good coverage of state of the art SAT solving and of the winners of recent SAT competitions and SAT races[20]:

- Look-ahead solvers:
  - OKsolver [47], a solver with well-defined behaviour, no ad-hoc heuristics, and which applies complete r_2 (at every node). This solver won gold at the SAT 2002 competition.
  - satz [55], a solver which applies partial r_2 and r_3. In the OKlibrary we maintain version 215, with improved/corrected in/output and coding standard.
  - march_pl [29], a solver applying partial r_2, r_3, and resolution- and equivalence-preprocessing. march_pl contains the same underlying technology as its sibling solvers march_{rw,hi,ks,dl,eq}, which won gold, silver and bronze at the 2004 to 2011 SAT competitions and SAT races. We use the pl (*p*artial *l*ookahead) version.
- Conflict-driven solvers:
  - MiniSat family:
    - MiniSat [21], version 2.0 and 2.2, the latest version of this well-established solver, used as starting point for many new conflict-driven solvers. Previous versions won gold at the SAT Race 2006 and 2008, as well as numerous bronze and silver awards at the SAT competition 2007.
    - CryptoMiniSat [65], a MiniSat derivative designed specifically to tackle hard cryptographic problems. This solver won gold at SAT Race 2010 and gold and silver at the SAT competition 2011. We use version 2.9.6.
    - Glucose [7], a MiniSat derivative utilising a new clause scoring scheme and aggressive learnt-clause deletion. This solver won gold in both SAT 2011 competition and SAT Challenge 2012. We use versions 2.0 and 2.2.
  - Lingeling family:
    - PicoSAT [9], a conflict-driven solver using an aggressive restart strategy, compact data-structures, and offering proof-trace options to allow for unsatisfiability checking. This solver won gold and silver at the SAT competition 2007. We use the latest version 913.
    - PrecoSAT [11], integrates the SATeLite preprocessor into PicoSAT, applying various reductions including partial r_2 at certain nodes in the search tree. This solver won gold and silver at the SAT 2009 competition. We use the latest version 570.
    - Lingeling [12], based on PrecoSAT, focuses further on integrating preprocessing and search, introducing new algorithms and data-structures to speed up these techniques and reduce memory footprint. As with PrecoSAT, this solver applies partial r_2 at specially chosen nodes in the search tree. This solver won bronze at the SAT 2011 competition and silver at the SAT Race 2010. We use the latest version ala-b02aa1a-121013.

6.1.1. Cube-and-Conquer

The Cube & Conquer method uses a look-ahead solver as the "cube-solver", splitting the instance into subinstances small enough such that the "conquer-solver", a conflict-driven solver, can solve almost all sub-instances in at most a few

[20] The (parent) SAT competition homepage is at http://www.satcompetition.org with links to each individual competition.

Table 8
Instance data for $F(3, t; n)$, where n is the number of vertices as well as the number of variables, $c = c_3 + c_t$ is the number of clauses, c_i the number of clauses of length i, and $\ell = 3c_3 + tc_t$ is the number of literal occurrences.

t	n	c	c_3	c_t	ℓ
12	135	5,251	4,489	762	22,611
13	160	7,308	6,320	988	31,804
14	186	9,795	8,556	1239	43,014
15	218	13,362	11,772	1590	59,166
16	238	15,812	14,042	1770	70,446
17	279	21,616	19,321	2295	96,978
18	312	26,889	24,180	2709	121,302
19	349	33,487	30,276	3211	151,837

seconds. We use the `OKsolver` as the cube-solver and `MiniSat` as the conquer-solver. The main (and single) parameter is $D \in \mathbb{N}_0$, the cut-off depth for the `OKsolver`: the DLL-tree created by the `OKsolver` is cut off when the number of assignments reaches D, where it is important that this includes *all assignments* on the path, not just the decisions, but also the forced assignments found by r_1 and r_2—only in this way a relatively balanced load is guaranteed. The data reported in Tables 10, 14 shows first data on the cube-phase, namely

- D (cut-off depth),
- the number of nodes in the (truncated) DLL-tree of the `OKsolver`,
- the time needed (this includes writing the partial assignments representing the sub-instances to files),
- and the number N of sub-instances.

For the conquer-phase we have:

- the median and maximum time for solving the sub-instances by `MiniSat`,
- the sum of conflicts over all sub-instances,
- and the total time used by `MiniSat`.

Finally the overall total time is reported, which does not include the time used by the processing-script, which applies the partial assignments to the original instance and produces so the sub-instances: this adds an overhead of nearly 20% for the smallest problem, but this proportion becomes smaller for larger problems, and is less than 1% for the largest problems.

6.1.2. VdW-problems

We consider the (unsatisfiable) instances to determine the upper bounds for $\mathrm{w}(2; 3, t)$ with $12 \le t \le 17$; in Table 8 we give basic data for these instances (plus $t = 18, 19$).

In Table 9, we see the running times and number of nodes/conflicts for the SAT solvers. We see that in general look-ahead solvers here have the upper hand over conflict-driven solvers, with the `tawSolver`-2.6 with a large margin the fastest solver. Regarding conflict-driven solvers, we see that version 2.2 for `MiniSat` is superior over version 2.0, while for `Glucose` it is the opposite. The low node-count for `march_pl` seems due to the preprocessing phase, which adds a large number of resolvents to the original instance: this reduces the node-count, but increases the run-time. Compared to the other look-ahead solvers, the strength of `tawSolver`-2.6 is that the number of nodes is just larger by a factor of most 3, while processing of each node happens much faster. Compared with the strongest conflict-driven solver, `MiniSat`-2.2, we see that the node-count of `tawSolver`-2.6 is considerably less than the number of conflicts used by `MiniSat`, and that one node is processed somewhat faster than one conflict.

One aspect important here for the superiority of look-ahead solver is the "tightness" of the problem formulation. Consider for example $t = 12$, not with $n = 135$ as in Tables 8, 9, but with $n = 1000$; this yields $c = 294{,}455$, $c_3 = 249{,}500$, $c_{12} = 44{,}955$, and $\ell = 1{,}287{,}960$, which is now a highly redundant problem instance. For `tawSolver`-2.6 we obtain 1,311,511 nodes and 2868 s, and for τ`awSolver`-2.6 we get 935,475 nodes and 2452 s, while for `MiniSat`-2.2 we get 1,140,616 conflicts and 159 s. We see that `MiniSat`-2.2 was able to utilise the additional clauses to determine unsatisfiability with fewer conflicts, and with a run-time not much affected by the large increase in problem size, while for `tawSolver`-2.6 the run-time (naturally) explodes, and the number of nodes stayed the same.[21] If we consider typical branching-heuristics for look-ahead solvers (as discussed in Section 2.3), then we see that locality of the search process is not taken into consideration, and thus for non-tight problem formulations the solver can "switch attention" again and again. This is very different from heuristics for conflict-driven solvers, which via "clause-activity" have a strong focus on locality of reasoning. Furthermore, look-ahead solvers consider much more of the whole input, for example the `tawSolver` considers always all remaining variables and their occurrences for the branching heuristic, while conflict-driven solvers do not use such global heuristics.

[21] The `OKsolver` yields a more extreme example: the run was aborted after 657,648 s and 603,177 nodes, where yet only 49.2% of the search space was visited (so that the solver was still working on completing the first branch at the root of the tree, making very slow progress towards 50%). This shows the big overhead caused by the r_2-reduction, and the danger of a heuristic which (numerically) sees opportunities "all over the place", and thus cannot focus on one relevant part of the input.

Table 9
Complete solvers on unsatisfiable instances $F(3,t;n)$ for computing $\mathrm{w}(2;3,t)$ (with $t = 12,\dots,16$ and $n = 135, 160, 186, 218, 238$). The first line is run-time in seconds, the second line is the number of nodes for look-ahead solvers resp. number of conflicts for conflict-driven solvers.

$t =$	12	13	14	15	16	17
`tawSolver-2.6`	11 961,949	83 5,638,667	673 35,085,795	5,010 194,035,915	42,356 1,462,429,351	401,940 10,258,378,909
`τawSolver-2.6`	19 953,179	143 5,869,055	1068 35,668,687	7,607 200,208,507	59,585 1,479,620,647	
`tawSolver-1.0`	47 1,790,733	463 13,722,975	4577 102,268,511	47,006 774,872,707	532,416 8,120,609,615	
`satz`	77 262,304	711 1,698,185	6233 10,822,316	54,913 66,595,028	562,161 599,520,428	
`march_pl`	185 47,963	1,849 279,061	17,018 1,975,338	175,614 11,959,263		
`OKsolver`	216 281,381	3,806 2,970,723	47,598 22,470,241			
`MiniSat-2.2`	107 5,963,349	1,716 63,901,998	16,836 463,984,635	190,211 3,205,639,994		
`MiniSat-2.0`	273 1,454,696	3,022 9,298,288	33,391 60,091,581	274,457 314,678,660		
`PrecoSAT`	211 2,425,722	2,777 16,978,254	47,624 140,816,236			
`PicoSAT`	259 9,643,671	4,258 82,811,468	48,372 576,692,221			
`Glucose-2.0`	58 1,263,087	781 8,377,487	84,334 163,500,051			
`Lingeling`	519 1,659,607	7,651 24,124,525	107,243 176,909,499			
`CryptoMiniSat`	212 2,109,106	4,630 18,137,202	141,636 205,583,043			
`Glucose-2.2`	94 1,444,017	1,412 10,447,051	>940,040 Aborted			

Table 10
`Cube & Conquer`, via the `OKsolver` as the cube-solver, and `MiniSat-2.2` as the conquer-solver. Times are in seconds. "Factor" is run-time of `MiniSat-2.2`, divided by total time of `Cube & Conquer`. The run-times of the `OKsolver` include writing all data-files (the partial assignments), the run-times of `MiniSat` include reading the files. 10^6 s are roughly 11.6 days.

$t =$	13	14	15	16	17
D	20	30	35	40	50
nds	3197	27,053	64,663	209,593	1,399,505
t	10	146	821	3248	23,546
N	1599	13,527	32,331	104,797	699,751
t: med, max	0.06, 0.49	0.06, 0.68	0.16, 3.9	0.46, 29.6	0.8, 199
Σ cfs	8,479,987	59,402,586	361,511,501	3,723,995,162	35,931,491,146
Σt	120	961	6888	80,056	1,006,718
Total t	130	1107	7709	83,304	1,030,264
Factor	13.2	15.2	24.7	NA	NA

Finally we consider `Cube & Conquer`, with the `OKsolver` as Cube-solver and `MiniSat-2.2` as Conquer-solver, in Table 10. We see that the combination is vastly superior to each of the two solvers involved, and approaches in performance the best solver, the `tawSolver-2.6` (but still slower by a factor of two).

6.1.3. *Palindromic vdW-problems*

The data for the palindromic problems we considered is shown in Table 11. Recall that for palindromic problems, that is, the determination of $\mathrm{pdw}(2;3,t) = (n_1, n_2)$, we have to determine two numbers: the n_1 such that all $F^{\mathrm{pd}}(3,t,n)$ with $n \le n_1$ are satisfiable, while $F^{\mathrm{pd}}(3,t,n_1+1)$ is unsatisfiable, and n_2 for which $F^{\mathrm{pd}}(3,t,n)$ is unsatisfiable for all $n \ge n_2$,

Table 11
Instance data for $F^{\text{pd}}(3, t, n)$, where v is the number of variables, $c = c_2 + c_3 + c_{\lceil t/2 \rceil} + c_{\lceil t/2 \rceil+1} + c_t$ is the number of clauses, c_i the number of clauses of length i, and ℓ is the number of literal occurrences.

t	n	v	c	ℓ	c_2	c_3	$c_{\lceil t/2 \rceil}$	$c_{\lceil t/2 \rceil+1}$	c_t
17	279	140	10,536	45,139	185	9,357	25	0	969
18	312	156	13,277	58,763	52	11,954	9	0	1262
19	347	174	16,208	70,414	230	14,586	28	0	1364
20	389	195	20,327	88,944	258	18,393	10	19	1647
21	405	203	21,950	96,305	269	19,958	29	0	1694
22	463	232	28,650	126,560	308	26,171	11	21	2139
23	507	254	34,289	152,236	337	31,448	34	0	2470
24	593	297	46,881	209,792	394	43,156	12	24	3295
25	607	304	48,979	219,525	404	45,237	37	0	3301
26	643	322	54,843	246,503	428	50,813	12	24	3566
27	699	350	64,719	292,102	465	60,133	38	0	4083

Table 12
Look-ahead solvers on unsatisfiable instances $F^{\text{pd}}(3, t; n)$ for computing $\text{w}(2; 3, t)$ (with $t = 17, \dots, 25$ and $n = 279, \dots, 607$). The first line is run-time in seconds, the second line is the number of nodes.

t	τawSolver-2.6	tawSolver-2.6	satz	tawSolver-1.0	march_pl	OKsolver
17	1 32,855	0.8 32,697	12 16,466	7 143,319	35 1,448	18 5,023
18	11 276,249	8 279,309	182 208,873	60 1,063,979	269 12,289	335 100,803
19	13 283,229	10 285,037	143 123,199	134 2,009,635	500 12,423	322 62,009
20	48 894,777	39 897,529	701 459,899	738 9,076,261	1,980 39,681	1,419 206,617
21	115 2,144,743	101 2,239,371	2,592 1,567,736	2,541 30,470,349	5,053 99,493	3,536 490,841
22	564 8,427,503	525 8,683,035	9,418 4,393,139	18,306 170,414,771	25,841 376,285	47,593 3,197,173
23	1,547 19,858,971	1695 21,565,129	35,633 12,587,868	86,869 573,190,251	77,763 876,315	132,150 7,461,907
24	8,558 79,790,419	26,724 198,685,857				
25	22,841 219,575,127					

while $F^{\text{pd}}(3, t, n_2 - 1)$ is satisfiable. In order to do so, as shown in Theorem 5.1, the main unsatisfiable instances are for n_1 and $n_2 + 1$. To reduce the amount of data, we do not show the data for these two critical points, but for n_2, which is easier than $n_2 + 1$ (in our range by a factor of around five; possible due to the fact that except of one case n_2 happens to be odd here, as discussed in the next paragraph), and harder than n_1.

For $F^{\text{pd}}(3, t, n)$ with odd n we can determine that the middle vertex $\frac{n+1}{2}$ cannot be element of the first block of the partition (belonging to progression-size 3), since then no other vertex could be in the first block (due to the palindromic property and the symmetric position of the middle vertex), and then we would have an arithmetic progression of size t in the second block. Due to this (and there might be other reasons), palindromic problems for odd n are easier (running times can go up by a factor of 10 for even n).[22]

First we consider the look-ahead solvers in Table 12. Comparing tawSolver with the other solvers, we see a similar behaviour as with (ordinary) vdW-problems, but more extreme so. The node-count of tawSolver-2.6 and τawSolver-2.6 is not much worse than the "real" look-ahead solvers, with exception of march_pl (where again a large number of inferred clauses is added by the solver). The weak performance of the OKsolver is (likely) explained by the instances not having many r_2-reductions (recall that OKsolver is *completely* eliminating failed literals, as the only solver), and so the overhead is prohibitive (the savings in node-count do not pay off). satz only investigates 10% of the most promising variables for r_2-reductions, and additionally looks for some r_3-reductions. This strategy here works far better than OKsolver's "strategy" (but the OKsolver deliberately does not employ a "strategy" here, since the aim is to have a stable and "mathematical meaningful" solver); nevertheless still the overhead is too large.

[22] We remark that while for example PrecoSAT determines this forced variable right at the beginning, this is not the case for the MiniSat versions, which infer that fact rather late, and they are helped by adding the corresponding unit-clause to the instance.

Table 13
Conflict-driven solvers on unsatisfiable instances $F^{pd}(3, t; n)$ for computing $w(2; 3, t)$ (with $t = 17, \ldots, 24$ and $n = 279, \ldots, 593$). The first line is run-time in seconds, the second line is the number of conflicts.

t	`MiniSat`	`Glucose`	`PrecoSAT`	`Lingeling`	`CryptoMiniSat`
17	0.8 34,426	0.8 34,826	1.2 41,961	3.7 57,306	3.6 59,443
18	19 607,908	14 340,568	25 506,793	59 919,123	78 871,916
19	19 568,924	15 336,861	24 485,357	61 915,107	72 765,301
20	118 2,852,150	66 1,132,012	131 1,799,145	355 3,633,502	384 3,071,462
21	423 9,179,642	228 2,903,573	445 4,687,589	1060 8,672,073	1418 8,458,496
22	3151 51,582,064	1631 13,397,451	2825 22,283,651	8428 41,696,062	14,321 49,716,762
23	8191 108,028,217	6817 36,314,064	9280 54,951,563	28,543 104,007,799	55,544 141,249,316
24	54,678 476,716,936	>992,540 >1,100,664,795 Aborted	82,750 261,084,988	152,076 285,546,948	

Table 14
`Cube & Conquer`, via the `OKsolver` as the cube-solver, and `MiniSat`-2.2 as the conquer-solver. Times are in seconds. "Factor" is run-time of best solver, i.e., τ`awSolver`-2.6, divided by total time of `Cube & Conquer`. 10^5 s are roughly 1.2 days.

$t =$	23	24	25	26	27
D	25	35	45	55	65
nds	1717	5559	17,633	77,161	220,069
t	106	500	1752	7889	25,478
N	859	2780	8817	38,581	110,032
t: med, max	0.95, 17.6	1.2, 27	0.81, 47	0.95, 58	0.82, 125
Σ cfs	27,308,572	93,831,664	258,829,555	1,231,383,588	3,423,841,749
Σt	1095	4466	11,822	55,306	172,033
Total t	1201	4966	13,574	63,195	197,511
Factor	1.3	1.7	1.7	NA	NA

An interesting aspect is that for larger t the more complex heuristic (i.e., projection) of τ`awSolver`-2.6 compared to `tawSolver`-2.6 pays off. This is different from ordinary vdW-problems. And as the comparison with `tawSolver`-1.0 shows, the heuristic (mostly the projection) is of great importance here (more pronounced than for ordinary vdW-problems).

The conflict-driven solvers are shown in Table 13. We see that they are not competitive with `tawSolver`-2.6 or τ`awSolver`-2.6, however now most of them are better than the "real" look-ahead solvers. Here `MiniSat`-2.2 is better than `MiniSat`-2.0, and `Glucose`-2.2 is better than `Glucose`-2.0, so we show only data for the newest versions. With `Glucose` we see a pattern which we observed also at other (hard) instance classes: for smaller instances `Glucose` is better than `MiniSat`, but from a certain point on the performance of `Glucose` becomes very bad. This is likely due to the more aggressive restart strategy, which pays off for smaller instances, but from a certain point on the solver becomes essentially incomplete.

Finally we consider `Cube & Conquer` in Table 14. We see that this is now the fastest solver overall. `Glucose`-2.2 is 10% faster, but since this is only a small amount, for consistency we stick with `MiniSat`-2.2.

6.2. Incomplete solvers (stochastic local search)

In the `OKlibrary` we use the `Ubcsat` suite (see [68]) of local-search algorithms in version 1-2-0. The considered algorithms are GSAT, GWSAT, GSAT-TABU, HSAT, HWSAT, WALKSAT, WALKSAT-TABU, WALKSAT-TABU-NoNull, Novelty, Novelty+, Novelty++, Novelty+p, Adaptive Novelty+, RNovelty, RNovelty+, SAPS, RSAPS, SAPS/NR, PAWS, DDFW, G2WSAT, Adaptive G2WSat, VW1, VW2, RoTS, IRoTS, and SAMD. The performance of local-search algorithms is very much instance-dependent, and so a good choice of algorithms is essential. Our experiments yield the following selection criteria:

- For standard problems (Section 3) the best advice seems to use GSAT-TABU for $t \leq 23$, to use RoTS for $t > 23$, and to use Adaptive G2WSat for $t > 33$ (also trying DDFW then).
- For the palindromic problems (Section 5) GSAT-TABU is the best algorithm.

For a given t in principle we let these algorithms run for $n = t + 1, t + 2, \ldots,$, until the search seems unable to find a solution. But running these algorithms from scratch on these vdW-problems is much less effective than using an incremental approach, based on a solution found for $n - 1$, respectively for palindromic vdW-problems on a solution found for $n - 2$ (according to Lemma 5.1), as initial guess, and repeating this process for the next n: this helps to go much quicker through the easier part of the search space (of possible n), and also seems to help for the harder problems. Finally, we recall that in Section 3.2 we explained how we made the distinction between lower bounds we conjecture to be exact and sheer lower bounds.

7. Conclusion

This article presented the following contributions to the fields of Ramsey theory and SAT solving:

- Study of $\mathrm{w}(2; 3, t)$:
  1. determination of $\mathrm{w}(2; 3, 19) = 349$;
  2. lower bounds for $\mathrm{w}(2; 3, t)$ with $20 \le t \le 30$, conjectured to be exact;
  3. further lower bounds for $31 \le t \le 39$;
  4. improved conjecture on the growth rate of $\mathrm{w}(2; 3, t)$;
  5. various observations on structural properties of good partitions.
- Introduction and study of $\mathrm{pdw}(2; 3, t)$:
  1. basic definitions and properties;
  2. determination of $\mathrm{pdw}(2; 3, t)$ for $t \le 27$;
  3. lower bounds for $\mathrm{pdw}(2; 3, t)$ with $28 \le t \le 35$, conjectured to be exact;
  4. further lower bounds for $36 \le t \le 39$.
- SAT solving:
  1. introduced the new SAT-solver `tawSolver`, with the basic implementation given by `tawSolver`-1.0, and the versions with improved heuristic by `tawSolver`-2.6 and τ`awSolver`-2.6;
  2. experimental comparison with current look-ahead and conflict-driven solvers;
  3. comparison and data for the new `Cube` & `Conquer` method;
  4. experimental determination of good local-search algorithms for lower bounds.

We hope that these investigations contribute to a better understanding of the connections between Ramsey theory and SAT solving. The following seem relevant research directions for future investigations:

- Showing $\mathrm{w}(2; 3, 20) = 389$ (recall Section 3.2) should be in reach with `tawSolver`-2.6, while showing $\mathrm{w}(2; 3, 21) = 416$ seems to require new (algorithmic) insight (when using similar computational resources).
- Conjecture 3.1 states that the lower bound from [15] for $\mathrm{w}(2; 3, t)$ is tight up to a small factor.
- In Section 4 four conjectures on patterns in good partitions are presented (one implying Conjecture 3.1).
- In Section 5.4 various open problems on palindromic van der Waerden numbers are stated.
- Considering SAT solving:
  1. Understand the differences between ordinary and palindromic problems:
     – Why is the projection relatively more important for the palindromic problems? (So that the difference between `tawSolver`-2.6 and `tawSolver`-1.0 is more pronounced, and τ`awSolver`-2.6 becomes faster than `tawSolver`-2.6 on bigger instances.)
     – Why do we have different behaviour of look-ahead versus conflict-driven solvers?
  2. Can the branching heuristic of `tawSolver` for the instances of this paper be much further improved? Especially can we gain some understanding of the weights?
  3. How to understand the success of `Cube` & `Conquer`? Does its success indicate that there are important dag-like structures in good resolution refutations of the instances of these classes, which are dispersed locally, so that ordinary conflict-driven solvers have problems exploiting them?

Acknowledgements

The authors would like to thank Donald Knuth, the Editor and the anonymous referees for their valuable suggestions and helpful comments.

References

[1] Dimitris Achlioptas, Random satisfiability, in: Biere et al. [13], pp. 245–270 (Chapter 8).
[2] Tanbir Ahmed, Some new van der Waerden numbers and some van der Waerden-type numbers, Integers 9 (2009) 65–76. #A6.

[3] Tanbir Ahmed, Two new van der Waerden numbers: $w(2; 3, 17)$ and $w(2; 3, 18)$, Integers 10 (2010) 369–377. #A32.
[4] Tanbir Ahmed, On computation of exact van der Waerden numbers, Integers 12 (3) (2012) 417–425. #A71.
[5] Tanbir Ahmed, Some more van der Waerden numbers, J. Integer Seq. 16 (4) (2013) #13.4.4.
[6] Tanbir Ahmed, Oliver Kullmann, Hunter Snevily, On the van der Waerden Numbers $w(2; 3, t)$, Technical Report, 2014, arXiv arXiv:1102.5433v4 [math.CO].
[7] Gilles Audemard, Laurent Simon, Predicting learnt clauses quality in modern SAT solvers, in: IJCAI'09 Proceedings of the 21st International Joint Conference on Artificial Intelligence, AAAI, 2009, pp. 399–404.
[8] Michael D. Beeler, Patrick E. O'Neil, Some new van der Waerden numbers, Discrete Math. 28 (2) (1979) 135–146.
[9] Armin Biere, Picosat essentials, J. Satisf. Boolean Model. Comput. 4 (2008) 75–97.
[10] Armin Biere, Bounded model checking, in: Biere et al. [13], pp. 455–481 (Chapter 14).
[11] Armin Biere, P{re, i}coSAT@SC'09, 2009. http://fmv.jku.at/precosat/preicosat-sc09.pdf.
[12] Armin Biere, Lingeling and friends entering the SAT Challenge 2012, in: Adrian Balint, Anton Belov, Daniel Diepold, Simon Gerber, Matti Järvisalo, Carsten Sinz (Eds.), Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions, in: Department of Computer Science Series of Publications B, vol. B-2012-2, University of Helsinki, 2012, pp. 33–34. https://helda.helsinki.fi/bitstream/handle/10138/34218/sc2012_proceedings.pdf.
[13] Armin Biere, Marijn J.H. Heule, Hans van Maaren, Toby Walsh (Eds.), Handbook of Satisfiability, in: Frontiers in Artificial Intelligence and Applications, vol. 185, IOS Press, 2009.
[14] T.C. Brown, Some new van der Waerden numbers (preliminary report), Notices Amer. Math. Soc. 21 (1974) 432.
[15] Tom Brown, Bruce M. Landman, Aaron Robertson, Bounds on some van der Waerden numbers, J. Combin. Theory Ser. A 115 (2008) 1304–1309.
[16] V. Chvátal, Some unknown van der Waerden numbers, in: R.K. Guy (Ed.), Combinatorial Structures and their Applications, Gordon and Breach, New York, 1970, pp. 31–33.
[17] Scott Cotton, Two techniques for minimizing resolution proofs, in: Strichman and Szeider [66], pp. 306–312. ISBN-13 978-3-642-14185-0.
[18] Adnan Darwiche, Knot Pipatsrisawat, Complete algorithms, in: Biere et al. [13], pp. 99–130 (Chapter 3).
[19] Martin Davis, George Logemann, Donald Loveland, A machine program for theorem-proving, Commun. ACM 5 (1962) 394–397.
[20] Michael R. Dransfield, Lengning Liu, Victor W. Marek, Miroslaw Truszczyński, Satisfiability and computing van der Waerden numbers, Electron. J. Combin. 11 (#R41) (2004).
[21] Niklas Eén, Niklas Sörensson, An extensible SAT-solver, in: Enrico Giunchiglia, Armando Tacchella (Eds.), Theory and Applications of Satisfiability Testing 2003, in: Lecture Notes in Computer Science, vol. 2919, Springer, Berlin, ISBN: 3-540-20851-8, 2004, pp. 502–518.
[22] Luís Gil, Paulo Flores, Luís Miguel Silveira, PMSat: a parallel version of MiniSAT, J. Satisf. Boolean Model. Comput. 6 (2009) 71–98.
[23] Carla P. Gomes, Ashish Sabharwal, Bart Selman, Model counting, in: Biere et al. [13], pp. 633–654 (Chapter 20).
[24] Jun Gu, The multi-SAT algorithm, Discrete Appl. Math. 96–97 (1999) 111–126.
[25] Long Guo, Youssef Hamadi, Said Jabbour, Lakhdar Sais, Diversification and intensification in parallel SAT solving, in: CP'10 Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming, in: Lecture Notes in Computer Science, vol. 6308, Springer-Verlag, 2010, pp. 252–265.
[26] Matthew Gwynne, Oliver Kullmann, Generalising and unifying SLUR and unit-refutation completeness, in: Peter van Emde Boas, Frans C.A. Groen, Giuseppe F. Italiano, Jerzy Nawrocki, Harald Sack (Eds.), SOFSEM 2013: Theory and Practice of Computer Science, in: Lecture Notes in Computer Science (LNCS), vol. 7741, Springer, 2013, pp. 220–232.
[27] Matthew Gwynne, Oliver Kullmann, Generalising unit-refutation completeness and SLUR via nested input resolution, J. Automat. Reason. 52 (1) (2014) 31–65.
[28] P.R. Herwig, M.J.H. Heule, P.M. van Lambalgen, H. van Maaren, A new method to construct lower bounds for van der Waerden numbers, Electron. J. Combin. 14 (#R6) (2007).
[29] Marijn Heule, Mark Dufour, Joris van Zwieten, Hans van Maaren, March_eq: implementing additional reasoning into an efficient look-ahead SAT solver, in: Hoos and Mitchell [36], pp. 345–359. ISBN 3-540-27829-X.
[30] Marijn J.H. Heule, Oliver Kullmann, Siert Wieringa, Armin Biere, Cube and conquer: guiding CDCL SAT solvers by lookaheads, in: Kerstin Eder, João Lourenço, Onn Shehory (Eds.), Hardware and Software: Verification and Testing, HVC 2011, in: Lecture Notes in Computer Science (LNCS), vol. 7261, Springer, 2012, pp. 50–65. http://cs.swan.ac.uk/~csoliver/papers.html#CuCo2011.
[31] Marijn J.H. Heule, Hans van Maaren, Look-ahead based SAT solvers, in: Biere et al. [13], pp. 155–184 (Chapter 5).
[32] Marijn J.H. Heule, Hans van Maaren, Parallel SAT solving using bit-level operations, J. Satisf. Boolean Model. Comput. 4 (2008) 99–116.
[33] Marijn Heule, Toby Walsh, Internal symmetry, in: Pierre Flener, Justin Pearson (Eds.), The 10th International Workshop on Symmetry in Constraint Satisfaction Problems, SymCon'10, 2010, pp. 19–33.
[34] Marijn Heule, Toby Walsh, Symmetry within solutions, in: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI-10, 2010, pp. 77–82.
[35] John N. Hooker, V. Vinay, Branching rules for satisfiability, J. Automat. Reason. 15 (1995) 359–383.
[36] Holger H. Hoos, David G. Mitchell (Eds.), Theory and Applications of Satisfiability Testing 2004, in: Lecture Notes in Computer Science, vol. 3542, Springer, Berlin, ISBN: 3-540-27829-X, 2005.
[37] Antti E. Hyvärinen, Tommi Junttila, Ilkka Niemelä, Incorporating clause learning in grid-based randomized SAT solving, J. Satisf. Boolean Model. Comput. 6 (2009) 223–244.
[38] Antti E. Hyvärinen, Tommi Junttila, Ilkka Niemelä, Partitioning search spaces of a randomized search, in: AI*IA 2009: Proceedings of the XIth International Conference of the Italian Association for Artificial Intelligence Reggio Emilia on Emergent Perspectives in Artificial Intelligence, in: Lecture Notes in Computer Science, vol. 5883, Springer-Verlag, 2009, pp. 243–252.
[39] Antti E. Hyvärinen, Tommi Junttila, Ilkka Niemelä, Partitioning SAT instances for distributed solving, in: LPAR'10 Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, in: Lecture Notes in Computer Science, vol. 6397, Springer-Verlag, 2010, pp. 372–386.
[40] Bernard Jurkowiak, Chu Min Li, Gil Utard, A parallelization scheme based on work stealing for a class of SAT solvers, J. Automat. Reason. 34 (1) (2005) 73–101.
[41] Henry A. Kautz, Ashish Sabharwal, Bart Selman, Incomplete algorithms, in: Biere et al. [13], pp. 185–203 (Chapter 6).
[42] Hans Kleine Büning, Oliver Kullmann, Minimal unsatisfiability and autarkies, in: Biere et al. [13], pp. 339–401 (Chapter 11).
[43] Michal Kouril, Computing the van der Waerden number $w(3, 4) = 293$, INTEGERS: Electron. J. Comb. Number Theory 12 (A46) (2012) 1–13.
[44] Michal Kouril, Jerome L. Paul, The van der Waerden number $W(2, 6)$ is 1132, Experiment. Math. 17 (1) (2008) 53–61.
[45] Daniel Kroening, Software verification, in: Biere et al. [13], pp. 505–532 (Chapter 16).
[46] Oliver Kullmann, Investigating a General Hierarchy of Polynomially Decidable Classes of CNF's Based on Short Tree-like Resolution Proofs, Technical Report TR99-041, Electronic Colloquium on Computational Complexity (ECCC), 1999.
[47] Oliver Kullmann, Investigating the Behaviour of a SAT Solver on Random Formulas, Technical Report CSR 23-2002, Swansea University, Computer Science Report Series, 2002, p. 119. Available from http://www-compsci.swan.ac.uk/reports/2002.html.
[48] Oliver Kullmann, The `OKlibrary`: Introducing a "holistic" research platform for (generalised) SAT solving, Stud. Log. 2 (1) (2009) 20–53.
[49] Oliver Kullmann, Fundaments of branching heuristics, in: Biere et al. [13], pp. 205–244 (Chapter 7).
[50] Oliver Kullmann, Exact Ramsey Theory: Green-Tao Numbers and SAT, Technical Report, 2010, arXiv arXiv:1004.0653v2 [cs.DM].
[51] Oliver Kullmann, Green-Tao numbers and SAT, in: Strichman and Szeider [66], pp. 352–362. ISBN-13 978-3-642-14185-0.
[52] Oliver Kullmann, Constraint satisfaction problems in clausal form I: autarkies and deficiency, Fund. Inform. 109 (1) (2011) 27–81.
[53] Oliver Kullmann, Constraint satisfaction problems in clausal form II: minimal unsatisfiability and conflict structure, Fund. Inform. 109 (1) (2011) 83–119.

[54] Bruce Landman, Aaron Robertson, Clay Culver, Some new exact van der Waerden numbers, INTEGERS: Electron. J. Comb. Number Theory 5 (2) (2005) 1–11. #A10.
[55] Chu Min Li, Anbulagan, Heuristics based on unit propagation for satisfiability problems, in: Proceedings of 15th International Joint Conference on Artificial Intelligence, IJCAI'97, Morgan Kaufmann Publishers, 1997, pp. 366–371.
[56] Joao P. Marques-Silva, Ines Lynce, Sharad Malik, Conflict-driven clause learning SAT solvers, in: Biere et al. [13], pp. 131–153 (Chapter 4).
[57] John R. Rabung, Some progression-free partitions constructed using Folkman's method, Canad. Math. Bull. 22 (1) (1979) 87–91.
[58] Stanisław P. Radziszowski, Small Ramsey numbers, Electron. J. Combin. (2009) Dynamic Surveys DS1, Revision 12; see http://www.combinatorics.org/Surveys.
[59] Vera Rosta, Ramsey theory applications, Electron. J. Combin. (2004) Dynamic Surveys DS13, Revision 1; see http://www.combinatorics.org/Surveys.
[60] K.F. Roth, On certain sets of integers, J. Lond. Math. Soc. 28 (1953) 245–252.
[61] Karem A. Sakallah, Symmetry and satisfiability, in: Biere et al. [13], pp. 289–338 (Chapter 10).
[62] Marko Samer, Stefan Szeider, Fixed-parameter tractability, in: Biere et al. [13], pp. 425–454 (Chapter 13).
[63] Tobias Schubert, Matthew Lewis, Bernd Becker, PaMiraXT: parallel SAT solving with threads and message passing, J. Satisf. Boolean Model. Comput. 6 (2009) 203–222.
[64] Pascal Schweitzer, Problems of Unknown Complexity: Graph Isomorphism and Ramsey Theoretic Numbers (Ph.D. thesis), Universität des Saarlandes, Saarbrücken, 2009, Revised version, April 2012; http://www.mpi-inf.mpg.de/~pascal/docs/thesis_pascal_schweitzer.pdf.
[65] Mate Soos, Karsten Nohl, Claude Castelluccia, Extending SAT solvers to cryptographic problems, in: Oliver Kullmann (Ed.), Theory and Applications of Satisfiability Testing—SAT 2009, in: Lecture Notes in Computer Science, vol. 5584, Springer, 2009, pp. 244–257. http://planete.inrialpes.fr/~soos/publications/Extending_SAT_2009.pdf.
[66] Ofer Strichman, Stefan Szeider (Eds.), Theory and Applications of Satisfiability Testing—SAT 2010, in: Lecture Notes in Computer Science, LNCS, vol. 6175, Springer, 2010, ISBN: 13 978-3-642-14185-0.
[67] E. Szemerédi, On sets of integers containing no *k* elements in arithmetic progression, Acta Arith. 27 (1975) 299–345.
[68] Dave A.D. Tompkins, Holger H. Hoos, UBCSAT: an implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT, in: Hoos and Mitchell [36], pp. 306–320. ISBN 3-540-27829-X.
[69] Peter van der Tak, Marijn J.H. Heule, Armin Biere, Concurrent Cube-and -Conquer, in: Alessandro Cimatti, Roberto Sebastiani (Eds.), Theory and Applications of Satisfiability Testing—SAT 2012, in: Lecture Notes in Computer Science (LNCS), vol. 7317, Springer, 2012, pp. 475–476.
[70] B.L. van der Waerden, Beweis einer Baudetschen Vermutung, Nieuw Arch. Wiskd. 15 (1927) 212–216.
[71] Hantao Zhang, Combinatorial designs by SAT solvers, in: Biere et al. [13], pp. 533–568 (Chapter 17).
[72] Hantao Zhang, Maria Paola Bonacina, Jie Hsiang, PSATO: a distributed propositional prover and its application to quasigroup problems, J. Symbolic. Comput. 11 (1996) 1–18.