TWO NEW VAN DER WAERDEN NUMBERS: w(2; 3, 17) AND w(2; 3, 18)

Tanbir Ahmed
ConCoCO Research Laboratory, Department of Computer Science and Software Engineering, Concordia University, Montréal, Canada
ta_ahmed@cs.concordia.ca

Received: 10/21/09, Revised: 3/4/10, Accepted: 3/28/10, Published: 8/10/10

Abstract

The *van der Waerden number* $w(r; k_1, k_2, \ldots, k_r)$ is the least integer m such that for every partition $P_1 \cup P_2 \cup \cdots \cup P_r$ of the set $\{1, 2, \ldots, m\}$, there is an index j in $\{1, 2, \ldots, r\}$ such that P_j contains an arithmetic progression of k_j terms. We have computed exact values of the previously unknown van der Waerden numbers $w(2; 3, 17)$ and $w(2; 3, 18)$.

1. Introduction

Van der Waerden's theorem [7] can be formulated (as in Chvátal [2]) as follows: Given any positive integer r and positive integers $k_1, k_2, \ldots, k_r$, there is an integer m such that given any partition

$$\{1, 2, \ldots, m\} = P_1 \cup P_2 \cup \cdots \cup P_r \tag{1}$$

there is always a class P_j containing an arithmetic progression of length k_j. Let us denote the least m with this property by $w(r; k_1, k_2, \ldots, k_r)$. By a *good partition*, we mean a partition of the form (1) such that no P_j contains an arithmetic progression of k_j terms.

We formulate an instance F of the satisfiability problem (described in the following paragraph) with n variables for the van der Waerden number $w(r; k_1, \ldots, k_r)$ such that F is satisfiable if and only if $n < w(r; k_1, \ldots, k_r)$.

To describe the satisfiability problem, we require a few other definitions. A *truth assignment* is a mapping f that assigns each variable in $\{x_1, x_2, \ldots, x_n\}$ a value in $\{0, 1\}$. The complement $\bar{x}_i$ of each variable x_i is defined by $f(\bar{x}_i) = 1 - f(x_i)$ for all truth assignments f. Both x_i and $\bar{x}_i$ are called *literals*: a *clause* is a set of (distinct) literals and a *formula* is a family of (not necessarily distinct) clauses. A truth assignment *satisfies a clause* if it maps at least one of its literals to 1; the assignment *satisfies a formula* if and only if it satisfies each of its clauses. A formula is called *satisfiable* if it is satisfied by at least one truth assignment; otherwise, it is called *unsatisfiable*. The problem of recognizing satisfiable formulas is known as the *satisfiability problem*, or SAT for short. These definitions are taken from Chvátal and Reed [3].

In this paper, our focus is on $w(2; k_1, k_2)$ only. Given a positive integer n, we construct an instance of the satisfiability problem with variables x_i for $1 \leqslant i \leqslant n$ and the following clauses:

(a) $\{\bar{x}_a, \bar{x}_{a+d}, \ldots, \bar{x}_{a+d(k_1-1)}\}$ with $a \geqslant 1, d \geqslant 1, a + d(k_1 - 1) \leqslant n$,

(b) $\{x_a, x_{a+d}, \ldots, x_{a+d(k_2-1)}\}$ with $a \geqslant 1, d \geqslant 1, a + d(k_2 - 1) \leqslant n$.

Here, $x_i = \texttt{1}$ encodes $i \in P_1$ and $x_i = \texttt{0}$ encodes $i \in P_2$ (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). Clauses (a) prohibit the existence of an arithmetic progression of length k_1 in P_1 and clauses (b) prohibit the existence of an arithmetic progression of length k_2 in P_2.

To test the satisfiability of the generated instance, we have coded an efficient implementation of the DPLL Algorithm [4], which we describe in brief before going to the next section. Given a formula F and a literal u in F, we let $F|u$ denote the *residual formula* arising from F when $f(u)$ is set to 1: explicitly, this formula is obtained from F by (i) removing all the clauses that contain u, (ii) deleting $\bar{u}$ from all the clauses that contain $\bar{u}$, (iii) removing both u and $\bar{u}$ from the list of literals. Each recursive call of DPLL may involve a choice of a literal u. Algorithms for making these choices are referred to as *branching rules*.

It is customary to represent each call of DPLL(F) by a node of a binary tree. By branching on a literal u, we mean calling DPLL($F|u$). If this call leads to a contradiction, then we call DPLL($F|\bar{u}$). Every node that is not a leaf has at least one child and may not have both children. This tree is referred to as *DPLL-tree* in the literature. Different branches of the DPLL-tree may be distributed to separate processors for improving the computation time.

Algorithm 1 RECURSIVE ALGORITHM DPLL(F)

```
1: function DPLL($F$)
2:     while $F$ includes a clause $C$ such that $|C| \leqslant 1$ do
3:         if $C = \emptyset$ then return UNSATISFIABLE
4:         else if $C = \{v\}$ then $F = F|v$
5:     end while
6:     if $F = \emptyset$ then return SATISFIABLE
7:     Choose an unassigned literal $u$ using a branching rule
8:     if DPLL($F|u$) = SATISFIABLE then return SATISFIABLE
9:     if DPLL($F|\overline{u}$) = SATISFIABLE then return SATISFIABLE
10:    return UNSATISFIABLE
11: end function
```

2. New Van Der Waerden Numbers

A list of known van der Waerden numbers was recently published in [1]. In Table 1, we present the exact values of $w(2; 3, 17)$ and $w(2; 3, 18)$. We provide examples of good partitions of the sets $\{1, 2, \ldots, 278\}$ and $\{1, 2, \ldots, 311\}$ for the numbers $w(2; 3, 17)$ and $w(2; 3, 18)$, respectively.

We denote partitions as strings; for example 11221122 means $P_1 = \{1, 2, 5, 6\}$ and $P_2 = \{3, 4, 7, 8\}$. We have used 2.2 GHz AMD Opteron 64-bit processors (80 of them) in the `cirrus` cluster of ConCoCO Research Laboratory at Concordia University to compute them. The CPU time is the total time taken by all the distributed processes each taking care of a separate branch of the DPLL-tree to prove that the SAT instance corresponding to $w(2; k_1, k_2)$ is unsatisfiable.

In the following section, we describe the reason behind choosing DPLL for computing van der Waerden numbers and provide a brief description of our implementation.

3. On Our Implementation of DPLL

For around fifty years, the DPLL backtrack-search algorithm has been immensely popular as a *complete* (that finds a satisfying assignment if one exists; otherwise, correctly says that no satisfying assignment exists) procedure to solve the satisfiability problem. Any *incomplete* algorithm (which is generally faster than algorithms like DPLL but may fail to deliver a satisfying assignment when there exists one) is not useful when we would like to correctly prove that an instance is unsatisfiable.

Our implementation is engineered to run faster on unsatisfiable van der Waerden instances. We make a little modification (as in Algorithm 2) in Algorithm 1 and describe why.

If in some yet-to-be-satisfied clause C, a literal u is unassigned and every other literal is assigned false, then u is called *unit.* We get an empty clause in $F|u$ only if $\bar{u}$ is already unit in F. So during unit-propagation (lines 2-5 in Algorithm 1), we can avoid computing $F|u$ and return UNSATISFIABLE, when both u and $\bar{u}$ are unit literals in F. During the search, we spend most of our time in unit-propagations leading to contradictions (generates an empty clause). We can save (as in lines 2-6 of Algorithm 2) one computation of $F|u$ in each of those contradictory unit-propagations. Assuming that the initial formula does not contain an empty clause, Algorithm 2 never generates an empty clause.

Now we describe how $F|u$ can be computed efficiently while solving instances for $w(2; 3, k)$ with $k \leqslant 32$. Let $L(u)$ denote the set of clauses that contain u as a literal. When computing $F|u$, each clause in $L(u)$ is deleted from F and each

Table 1: Two new van der Waerden numbers

$w(2; k_1, k_2)$	Example of a good partition	CPU-time	Run-time
$w(2; 3, 17) = 279$	22221222 22212222 21222122 22222222 12222221 22222122 22222211 22222222 22121222 22212222 22212122 22222212 22222222 22212122 22222222 22222212 22222222 11221122 22222221 22222222 22222222 12122222 22222221 22222222 12122222 22122222 21212222 22222211 22222222 12222212 22222122 22222222 12221222 22122222 212222	301 days	5 days
$w(2; 3, 18) = 312$	22222212 222222AB 22212222 22222222 22222112 22112122 22222222 12222222 22222222 21222222 21222222 22211211 22221222 22222212 22222212 22212222 22222222 22122222 22222122 22222222 21222222 22221222 22222222 22212222 12222222 12222222 22122221 12112222 22222122 22222122 22222222 22222212 22222222 21211222 11222222 22222222 2221222C 12222222 D222222 (where ABCD is arbitrary).	13.6 yrs	70 days

Algorithm 2 Our variant of the DPLL algorithm

```
1: function DPLL($F$)
2:     while TRUE do
3:         if $\{u\} \in F$ and $\{\bar{u}\} \in F$ then return UNSATISFIABLE
4:         else if there is a clause $\{v\}$ then $F = F|v$
5:         else break
6:     end while
7:     if $F = \emptyset$ then return SATISFIABLE
8:     Choose an unassigned literal $u$ using a branching rule
9:     if DPLL($F|u$) = SATISFIABLE then return SATISFIABLE
10:    if DPLL($F|\overline{u}$) = SATISFIABLE then return SATISFIABLE
11:    return UNSATISFIABLE
12: end function
```

clause in $L(\bar{u})$ shrinks in length by one. When the length of a clause becomes one, the general idea to find the unit literal is to scan the clause, which is expensive if there are many long clauses. SAT instances corresponding to $w(2;3,17)$ and $w(2;3,18)$ contain clauses of length 17 and 18, respectively. We encode each clause C as a 32 bit unsigned integer b, which is initialized to $2^t - 1$ (where $t = |C|$), a bitstring of t 1's. If $\bar{u}$ is the i-th ($i \in \{0, 1, \ldots, t-1\}$) literal in C, then we subtract 2^i from b. When $|C| = 1$, we know that b equals 2^s for some nonnegative integer s less than 32. We can compute, in at most 5 ($= \log_2 32$) steps, the location of the only TRUE bit, which corresponds to the unit literal, say v, in C. This works for instances corresponding to every $w(2;3,k)$ with $k \leqslant 32$.

The size of the DPLL-tree greatly varies with the choice of the branching rule. It is hard to find a branching rule that works good on every SAT instance. Branching rules can be described using a paradigm (proposed by Chvátal and introduced in Ouyang [6]), which associates a weight $w(F, u)$ with each literal u, and chooses a function Φ of two variables. The paradigm is as follows:

($\star$) Find a variable x that maximizes $\Phi(w(F, x), w(F, \bar{x}))$; choose x if $w(F, x) \geqslant w(F, \bar{x})$, otherwise, choose $\bar{x}$. If more than one variable maximizes Φ, then ties have to be broken by some rule.

Usually, $w(F, u)$ is defined in terms of $d_k(F, u)$, which is the number of clauses of length k in F that contain literal u. These branching rules (see Ouyang [6]) choose a literal analyzing the current state of the formula. These are heuristics each striving for a tradeoff between its own running time and the ability to reduce the size of the DPLL-tree. We use Two-sided Jeroslaw-Wang (2sJW), (by Hooker and Vinay [5]) in our implementation, which is ($\star$) with

$$w(F,u) = \sum_k 2^{-k} d_k(F,u),\ \ \Phi(x,y) = x+y.$$

Our implementation performs best with 2sJW (as branching rule) on unsatisfiable instances corresponding to van der Waerden numbers. All of the 30 numbers reported in [1] were computed using this branching rule. In addition, this branching rule helps us to recognize a pattern in the size of the subtrees when we split the DPLL-tree of $w(2;3,k)$ for distributed computation, as described in the following section.

Table 2: Splitting the DPLL-tree for $w(2;3,14)$

Process	(u_1, u_2, u_3)	CPU-time
P_0	(-93,-94,-92)	4635.50 sec
P_1	(-93,-94, 92)	1041.60 sec
P_2	(-93, 94,-96)	1068.80 sec
P_3	(-93, 94, 96)	178.51 sec
P_4	(93,-94,-91)	1074.53 sec
P_5	(93,-94, 91)	177.71 sec
P_6	(93, 94,-89)	224.78 sec
P_7	(93, 94, 89)	32.77 sec

Table 3: Splitting the DPLL-tree for $w(2;3,15)$

Process	(u_1, u_2, u_3, u_4)	CPU-time
P_0	(-109,-110,-108,-111)	40746.08 sec
P_1	(-109,-110,-108, 111)	9443.45 sec
P_2	(-109,-110, 108,-111)	9308.47 sec
P_3	(-109,-110, 108, 111)	1235.54 sec
P_4	(-109, 110,-112,-106)	9348.02 sec
P_5	(-109, 110,-112, 106)	1264.93 sec
P_6	(-109, 110, 112,-107)	1619.97 sec
P_7	(-109, 110, 112, 107)	191.86 sec
P_8	(109,-110,-107,-113)	9703.46 sec
P_9	(109,-110,-107, 113)	1265.80 sec
P_{10}	(109,-110, 107,-112)	1597.04 sec
P_{11}	(109,-110, 107, 112)	192.01 sec
P_{12}	(109, 110,-105,-114)	1909.96 sec
P_{13}	(109, 110,-105, 114)	251.87 sec
P_{14}	(109, 110, 105,-114)	255.68 sec
P_{15}	(109, 110, 105, 114)	18.01 sec

4. On Distributed Application of Our Implementation

Our implementation of DPLL works as a stand-alone SAT-solver. We choose to split the DPLL-tree and use the stand-alone solver for each subtree instead of implementing a parallel version of the solver. We completely avoid message-passing overheads between processes and other synchronization issues. In our case, the total CPU-time is the sum of CPU-times taken by each of the processes, and the total running time is the maximum of the CPU-times.

Given a branching rule, there are at most 2^t branching-sequences of length t from the root. Each branching-sequence $(u_1, u_2, \ldots, u_t)$ results in a subtree rooted at DPLL$(F|u_t)$. Each of these subtrees can be explored separately and independently by the stand-alone SAT-solver in a separate processor. Depending

Table 4: Further splitting the DPLL-tree for $w(2; 3, 15)$

Process	(u_1, u_2, u_3, u_4)	(u_5, u_6, u_7)	(u_8, u_9)	CPU-time
$P_{0,0,0}$	(-109,-110,-108,-111)	(-107,-112,-106)	(-113,-105)	10930.76 sec
$P_{0,0,1}$	(-109,-110,-108,-111)	(-107,-112,-106)	(-113, 105)	4152.11 sec
$P_{0,0,2}$	(-109,-110,-108,-111)	(-107,-112,-106)	(113,-105)	4157.58 sec
$P_{0,0,3}$	(-109,-110,-108,-111)	(-107,-112,-106)	(113, 105)	730.76 sec
$P_{0,1}$	(-109,-110,-108,-111)	(-107,-112, 106)		6097.47 sec
$P_{0,2}$	(-109,-110,-108,-111)	(-107, 112,-106)		5831.26 sec
$P_{0,3}$	(-109,-110,-108,-111)	(-107, 112, 106)		841.39 sec
$P_{0,4}$	(-109,-110,-108,-111)	(107,-112,-113)		5942.24 sec
$P_{0,5}$	(-109,-110,-108,-111)	(107,-112, 113)		840.16 sec
$P_{0,6}$	(-109,-110,-108,-111)	(107, 112,-105)		978.12 sec
$P_{0,7}$	(-109,-110,-108,-111)	(107, 112, 105)		262.63 sec
P_1	(-109,-110,-108, 111)			9443.45 sec
P_2	(-109,-110, 108,-111)			9308.47 sec
P_3	(-109,-110, 108, 111)			1235.54 sec
P_4	(-109, 110,-112,-106)			9348.02 sec
P_5	(-109, 110,-112, 106)			1264.93 sec
P_6	(-109, 110, 112,-107)			1619.97 sec
P_7	(-109, 110, 112, 107)			191.86 sec
P_8	(109,-110,-107,-113)			9703.46 sec
P_9	(109,-110,-107, 113)			1265.80 sec
P_{10}	(109,-110, 107,-112)			1597.04 sec
P_{11}	(109,-110, 107, 112)			192.01 sec
P_{12}	(109, 110,-105,-114)			1909.96 sec
P_{13}	(109, 110,-105, 114)			251.87 sec
P_{14}	(109, 110, 105,-114)			255.68 sec
P_{15}	(109, 110, 105, 114)			18.01 sec

on the number of processors available, we can set the value of t. As examples, we show (in Tables 2 and 3) the CPU-times of distributed processes for instances corresponding to $w(2;3,14)$ and $w(2;3,15)$, respectively. We write j and $-j$ to mean the literals x_j and $\bar{x}_j$, respectively.

For proving that the instance corresponding to $w(2;3,14)$ is unsatisfiable, we split the DPLL-tree into 8 subtrees, and run our SAT-solver on these subtrees separately and simultaneously in 8 different processors. We see, as in Table 2, that the subtrees corresponding to P_0, P_1, P_2, and P_4 are bigger than the other four subtrees. Similarly, we distribute the DPLL-tree for $w(2;3,15)$ into 16 processors. Here we see, as in Table 3, that the subtrees corresponding to P_0, P_1, P_2, P_4, and P_8 are bigger than the other eleven subtrees. In each of the cases, the subtree corresponding to P_0 is the biggest. In fact, these observations are true for all $w(2;3,k)$ with $k \leqslant 18$. From Table 3, we see that the total running time (the maximum of the CPU-times taken by the 16 processes) is 40746.08 seconds to prove that the instance corresponding to $w(2;3,15)$ is unsatisfiable. If we judiciously split the bigger processes considering the number of processors available, then we can control the total running time as shown in Table 4.

From Table 4, we see that the total running time (the maximum of the CPU-times taken by the 26 processes) has come down to 10930.76 seconds. The above techniques may be used to compute $w(2;3,k)$ for $k \geqslant 19$.

Acknowledgements The author is grateful to Vašek Chvátal for his continuous support, to the anonymous referee for his or her helpful comments and suggestions, and to Andalib Parvez for carefully reading the manuscript.

References

[1] Ahmed T., Some new van der Waerden numbers and some van der Waerden-type numbers, *Integers* **9** (2009), A06, 65–76, MR2506138.

[2] Chvátal V., Some unknown van der Waerden numbers, *Combinatorial Structures and Their Applications (R.Guy et al.,eds.)*, 31–33, Gordon and Breach, New York, 1970, MR0266891.

[3] Chvátal V., Reed B., Mick Gets Some (The Odds Are on His Side), *Proceedings of the 33rd Annual Symposium on FOCS*, 1992, 620–627.

[4] Davis M., Logemann G., Loveland D., A machine program for theorem-proving, *Comm. ACM* **5** (1962), 394–397, MR0149690.

[5] Hooker J. N., Vinay V., Branching rules for satisfiability, *J. Automat. Reason.* **15(3)** 1995, 359–383, MR1356629.

[6] Ouyang M., Implementation of the DPLL algorithm, *PhD Thesis*, Rutgers University, 1999.

[7] Van der Waerden, B. L., Beweis einer Baudetschen Vermutung, *Nieuw Archief voor Wiskunde* **15** (1927), 212–216.