jet INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2011, VOL. 57, NO. 3, PP. 233–249
Manuscript received June 21, 2011; revised September 2011. DOI: 10.2478/v10177-011-0032-9

The Solution of SAT Problems Using Ternary Vectors and Parallel Processing

Christian Posthoff and Bernd Steinbach

***Abstract*—This paper will show a new approach to the solution of SAT-problems. It has been based on the isomorphism between the Boolean algebras of finite sets and the Boolean algebras of logic functions depending on a finite number of binary variables. *Ternary vectors* are the main data structure representing sets of Boolean vectors. The respective set operations (mainly the *complement* and the *intersection*) can be executed in a *bit-parallel* way (64 bits at present), but additionally also on different processors working in parallel. Even a hierarchy of processors, a small set of processor cores of a single CPU, and the huge number of cores of the GPU has been taken into consideration. There is no need for any search algorithms. The approach always finds *all* solutions of the problem without consideration of special cases (such us *no solution*, *one solution*, *all solutions*). It also allows to include problem-relevant knowledge into the problem-solving process at an early point of time. Very often it is possible to use ternary vectors directly for the modeling of a problem. Some examples are used to illustrate the efficiency of this approach (Sudoku, Queen's problems on the chessboard, node bases in graphs, graph-coloring problems, Hamiltonian and Eulerian paths etc.).**

***Keywords*—SAT-solver, ternary vector, parallel processing, XBOOLE.**

I. Introduction

FOR many years our research is centered on the solution of logic equations in a very general sense. Two logic functions $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n)$ can be used to build a general logic equation:

$$f(x_1, \ldots, x_n) = g(x_1, \ldots, x_n).$$

Each vector $\mathbf{x} = (x_1, \ldots, x_n)$ resulting in $f(\mathbf{x}) = g(\mathbf{x}) = 0$ or in $f(\mathbf{x}) = g(\mathbf{x}) = 1$ is considered to be a solution of this equation. In this sense the solution of the SAT-problem is a special case of our general intentions: the function $f(\mathbf{x})$ is now given as a conjunction of disjunctions of non-negated or negated variables, and the right side is the constant function $1(\mathbf{x})$. This research resulted, in addition to several publications, in the software package XBOOLE [1] that can be found at

http://www.informatik.tu-freiberg.de/XBOOLE.

The books [2], [3], and [4] can be used to get a full understanding of the results that have been achieved. Our main target has been the development of "numerical methods" for the use of logics in circuit design, but over the past years we also solved many problems of Discrete Mathematics, constraint problems and similar applications.

C. Posthoff is with the Department of Computing and Information Technology, Faculty of Science and Agriculture, St. Augustine Campus, The University of The West Indies, Trinidad & Tobago (e-mail: christian.posthoff@sta.uwi.edu).

B. Steinbach is with the Department of Computer Science, Faculty of Mathematics and Computer Sciences, Freiberg University of Mining and Technology, D-09596 Freiberg, Germany (e-mail: steinb@informatik.tu-freiberg.de).

II. Basic Concepts

In this section we summarize the basic concepts. Naturally they are well known to everybody who is working in this area. We include these concepts only in order to have a consistent representation for our explanations.

Let $B = \{0, 1\}$ be the set of two values 0 and 1. The propositional logic identifies the value 0 very often with **false** or **f**, the value **true** or **t** is used instead of 1. We will use constantly the values 0 and 1. With this set B, we get

$$B^n = \underbrace{B \times \ldots \times B}_{\text{n times}} = \{\mathbf{x} \,|\, \mathbf{x} = (x_1, \ldots, x_n),\ x_i \in B,\ i = 1, \ldots, n\}$$

as the set of all binary vectors of length n (vectors with n components). It can easily be seen (by induction) that this set has 2^n elements. In Computer Science these vectors are well known as, for instance, *dual numbers* representing the natural numbers from 0 to $2^n - 1$. Any unique mapping

$$f\ :\ B^n \Rightarrow B$$

is a *(logic) function* of n variables. The most important function for $n = 1$ is the *negation* indicated, for a given function f, by $\overline{f}$. It is defined by Table I and converts the value 0 into 1 and 1 into 0. Two applications of the negation result in the original value.

TABLE I
The Negation of a Variable x_1

x_1	$\overline{x_1}$	$\overline{\overline{x_1}}$
0	1	0
1	0	1

All the other important elementary functions will be defined for $n = 2$. Their definition and the operators to be used are given in Table II. For $n \geq 3$ they can be defined by induction: $x_1 \vee x_2 \vee x_3 = (x_1 \vee x_2) \vee x_3$ etc.

Based on these definitions, logic functions can be described or defined by means of *expressions* (*formulas*). We start with variables x_1, x_2, ... In a first step these variables can be negated, and we get $\overline{x}_1$, $\overline{x}_2$ etc. Variables and negated variables together are very often called *literals*. Each literal can take the values 0 or 1, resp. Now these literals can be combined by the given operations *conjunction*, *disjunction*, *antivalence* and

TABLE II
THE BASIC FUNCTIONS OF TWO VARIABLES

x_1	x_2	$x_1 \wedge x_2$ *conjunction*	$x_1 \vee x_2$ *disjunction*	$x_1 \oplus x_2$ *antivalence*	$x_1 \sim x_2$ *equivalence*
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	1

equivalence, resp., according to Table II. In this table we can see the following remarkable properties:

- The conjunction of two values is equal to 1 if and only if both values are equal to 1.
- The disjunction of two values is equal to 0 if and only if both values are equal to 0.
- The antivalence is equal to 0 if and only if both values are equal to each other.
- The equivalence is equal to 1 if and only if both values are equal to each other.
- The antivalence is equal to the negated equivalence (and vice versa).

Other important rules are

- Negation rules:
  $x_1 \vee \overline{x}_1 = 1$, $x_1 \wedge \overline{x}_1 = 0$,
  $\overline{x}_1 = x_1 \oplus 1 = x_1 \sim 0$.
- De Morgan's Laws:
  $\overline{x_1 \wedge x_2} = \overline{x}_1 \vee \overline{x}_2$, $\quad \overline{x_1 \vee x_2} = \overline{x}_1 \wedge \overline{x}_2$.
- Commutativity:
  $x_1 \vee x_2 = x_2 \vee x_1$, $\quad x_1 \wedge x_2 = x_2 \wedge x_1$,
  $x_1 \oplus x_2 = x_2 \oplus x_1$, $\quad x_1 \sim x_2 = x_2 \sim x_1$.
- Associativity:
  $(x_1 \vee x_2) \vee x_3 = x_1 \vee (x_2 \vee x_3)$,
  $(x_1 \wedge x_2) \wedge x_3 = x_1 \wedge (x_2 \wedge x_3)$,
  $(x_1 \oplus x_2) \oplus x_3 = x_1 \oplus (x_2 \oplus x_3)$,
  $(x_1 \sim x_2) \sim x_3 = x_1 \sim (x_2 \sim x_3)$.
- Distributivity:
  $x_1 \wedge (x_2 \vee x_3) = (x_1 \wedge x_2) \vee (x_1 \wedge x_3)$,
  $x_1 \vee (x_2 \wedge x_3) = (x_1 \vee x_2) \wedge (x_1 \vee x_3)$,
  $x_1 \wedge (x_2 \oplus x_3) = (x_1 \wedge x_2) \oplus (x_1 \wedge x_3)$,
  $x_1 \vee (x_2 \sim x_3) = (x_1 \vee x_2) \sim (x_1 \vee x_3)$.
- The use of 0 and 1:
  $x_1 \wedge 1 = x_1$, $\quad x_1 \vee 1 = 1$,
  $x_1 \wedge 0 = 0$, $\quad x_1 \vee 0 = x_1$,
  $x_1 \oplus 1 = \overline{x}_1$, $\quad x_1 \sim 1 = x_1$,
  $x_1 \oplus 0 = x_1$ $\quad x_1 \sim 0 = \overline{x}_1$.
- The elimination of $\oplus$ and $\sim$:
  $x_1 \oplus x_2 = x_1\overline{x}_2 \vee \overline{x}_1 x_2$,
  $x_1 \sim x_2 = x_1 x_2 \vee \overline{x}_1\overline{x}_2$.

Mostly the $\wedge$ between the brackets will be omitted (like the multiplication sign in arithmetic expressions). For two given functions $f(\mathbf{x})$ and $g(\mathbf{x})$ we consider the expression $f(\mathbf{x}) = g(\mathbf{x})$ as a Boolean equation, and each vector $\mathbf{x}$ with $f(\mathbf{x}) = g(\mathbf{x}) = 0$ or with $f(\mathbf{x}) = g(\mathbf{x}) = 1$ is a solution of the given equation. We can assume that f and g always depend on the same set of variables. If, for instance, x_n is missing in the definition of a function f, then we duplicate the definition of f for $x_n = 0$ and $x_n = 1$:

$$f(x_1, \ldots, x_n) = f(x_1, \ldots, x_{n-1})(x_n \vee \overline{x}_n)$$
$$= x_n f(x_1, \ldots, x_{n-1}) \vee \overline{x}_n f(x_1, \ldots, x_{n-1}) \ .$$

In this way missing variables can be added on both sides of the equation until both sides depend on the same set of variables.

Finally, if the function on the left side has been given as a *conjunction of disjunctions of literals* (**clauses**) and the function on the right side is constantly equal to 1, then we consider this special equation as an example of a SAT-problem.

Example.

$$(x_1 \vee \overline{x}_2 \vee x_4) \wedge (x_2 \vee x_3 \vee \overline{x}_5) = 1,$$

or shorter

$$(x_1 \vee \overline{x}_2 \vee x_4)(x_2 \vee x_3 \vee \overline{x}_5) = 1.$$

This problem is considered to be difficult because (in principle) each binary vector of B^n has to be checked whether each bracket has the value 1 for the values of this vector – the conjunction of the brackets is equal to 1 if and only if each bracket itself is equal to 1. This means that the number of vectors that have to be checked is given by an *exponential function*: if n is the number of variables to be considered then 2^n binary vectors have to be checked. The solution algorithm will have *exponential complexity*.

In this example it would be sufficient to set $x_1 = 1$ and $x_2 = 1$, and this would be already a solution of the problem, independent on the values of x_3, x_4 and x_5. Many other possibilities will exist.

Hint: We always include all the variables appearing in at least one of the clauses into our considerations. The given example considers the first bracket as well as the second bracket as a function of x_1, x_2, x_3, x_4, x_5.

The *dual problem* can be defined in the following way: in a given SAT-problem

- we exchange each literal by the negated literal;
- we exchange each $\wedge$ by $\vee$ and vice versa;
- we exchange the role of 0 and 1.

Example.

$$(x_1 \vee \overline{x}_2 \vee x_4) \wedge (x_2 \vee x_3 \vee \overline{x}_5) = 1$$

will be transformed into

$$\overline{x}_1 x_2 \overline{x}_4 \vee \overline{x}_2 \overline{x}_3 x_5 = 0.$$

The two equations have the same solution set and the same solution algorithms. It is now common practice to use the conjunctive format.

III. TERNARY VECTORS AS THE MAIN DATA STRUCTURE

As a first step we introduce the data structure of a *ternary vector*. Let

$$\mathbf{x} = (x_1, \cdots, x_n), x_i \in \{0, 1, -\}, i = 1, \ldots, n.$$

Then $\mathbf{x}$ is called a **ternary vector**. The components of this vector can take one of three values. Such a vector has the big

advantage that it can be used or understood as an abbreviation or description of a set of binary vectors. When we replace one value $-$ by 0 or 1, then we get two new ternary vectors **generated** by this ternary vector. These two vectors have the same values in the other components, in the considered component they have the value 0 and 1, resp. In this way the vector $(0-1-)$ represents four binary vectors $(0\underline{0}1\underline{0})$, $(0\underline{0}1\underline{1})$, $(0\underline{1}1\underline{0})$ and $(0\underline{1}1\underline{1})$. A **list** (**matrix**) of ternary vectors can be understood as the **union** of the corresponding sets of binary vectors. A single ternary vector that includes d dash elements $(-)$ represents 2^d binary vectors. Hence, ternary vectors allow to reduce the required memory exponentially.

There is a direct relation of ternary vectors with conjunctions of Boolean variables. When there is given a conjunction C with variables $x_1, \ldots, x_k$, then we can build a ternary vector t with the components $t_1, \ldots, t_k$ according to the following coding:

$$x_i : t_i = 1, \qquad \overline{x}_i : t_i = 0, \qquad x_i \text{ missing} : t_i = -.$$

This coding expresses directly on one side the respective conjunction, on the other side the set of all binary vectors satisfying $C = 1$. The value $-$ indicates that the value of the respective variable has not yet been defined or determined.

Example: Let be given $x_1\overline{x}_2x_3\overline{x}_5 = 1$, then we have $\mathbf{t} = (t_1, t_2, t_3, t_4, t_5) = (101-0)$ which expresses the two binary vectors (10100) and (10110).

Hint: It has been assumed that the relevant Boolean space includes the variables x_1, x_2, x_3, x_4, x_5. The problem space to be considered always comprises the union of the variables that appear in the overall problem. If, for instance, x_6 is also a variable to be considered, then we would write

$$\mathbf{t} = \begin{pmatrix} 1 & 0 & 1 & - & 0 & - \end{pmatrix}.$$

IV. Set-theoretic Considerations

Let be given two ternary vectors **x** and **y**. The **intersection** of these two vectors (i.e. the intersection of the respective two sets of binary vectors) will be computed according to Table III which has to be applied in each component of the two vectors. The symbol $\emptyset$ indicates that the intersection of the two sets is empty.

TABLE III
Intersection of Ternary Values

x_i	0	0	0	1	1	1	$-$	$-$	$-$
y_i	0	1	$-$	0	1	$-$	0	1	$-$
$x_i \cap y_i$	0	$\emptyset$	0	$\emptyset$	1	1	0	1	$-$

A sophisticated coding of the three values 0, 1 and $-$ allows the introduction of binary vector operations that can be executed on the level of registers (32, 64 or even 128 bits in parallel). We use the coding of Table IV. *The first bit indicates that for a component i there is a value 0 or 1 in this component of the ternary vector, the second bit indicates the value itself.*

When the three-valued operations for the intersection are transferred to these binary vectors, then the intersection is empty if and only if

$$bit1(\mathbf{x}) \wedge bit1(\mathbf{y}) \wedge (bit2(\mathbf{x}) \oplus bit2(\mathbf{y})) \neq \mathbf{0}.$$

TABLE IV
Binary Code of Ternary Values

ternary value	bit1	bit2
0	1	0
1	1	1
$-$	0	0

If the intersection is not empty, then it can be determined by the following bit vector operations:

$$bit1(\mathbf{x} \cap \mathbf{y}) = bit1(\mathbf{x}) \vee bit1(\mathbf{y}),$$

$$bit2(\mathbf{x} \cap \mathbf{y}) = bit2(\mathbf{x}) \vee bit2(\mathbf{y}).$$

Hint: $\oplus$ indicates the exclusive-or, $\mathbf{0}$ is the vector the components of which are all equal to 0. Hence, by using some very fast and very simple parallel bit vector operations (available on the hardware level), we can find the intersection of two ternary vectors. It is easy to see that two or more vector operations have to be applied when the problem depends on more than 64 variables.

We can see that the intersection is empty if the two ternary values 0 and 1 meet each other in at least one component. Then the two sets of binary vectors corresponding to the two ternary vectors are *disjoint sets*. We call the two vectors *orthogonal to each other*. This property is very useful, because orthogonal vectors avoid it that some binary vectors will be elements of both sets. This concept can be extended to more than two vectors. If there is a set of ternary vectors $\{\mathbf{t}^1, \mathbf{t}^2, \ldots, \mathbf{t}^k\}$ then this set is an orthogonal set of ternary vectors if all pairs $(\mathbf{t}^i, \mathbf{t}^j)$ with vectors of the given set consist of orthogonal vectors.

We have already seen that for a given conjunction C the solution set of the equation $C = 1$ can be described by one single ternary vector. Let us consider now the equation $C = 0$ for the same conjunction $x_1\overline{x}_2x_3\overline{x}_5$. The value of C is equal to 0 if $x_1 = 0$ or $\overline{x}_2 = 0$ $(x_2 = 1)$, or $x_3 = 0$, or $\overline{x}_5 = 0$ $(x_5 = 1)$. These results (these four solution sets) can be represented using the following ternary matrix:

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ 0 & - & - & - & - \\ - & 1 & - & - & - \\ - & - & 0 & - & - \\ - & - & - & - & 1 \end{pmatrix}.$$

This matrix has to be understood as the union of the four sets generated by the four ternary vectors in row 1, 2, 3 and 4, resp. Over the years we made the experience that the intersection between such solution sets very often is not empty, and it requires quite some time to figure out the real number of solutions of a given problem. Therefore we are using immediately the *orthogonal coding* of the solution sets. The first row covers all solutions with $x_1 = 0$. Therefore in the second row it can be assumed that $x_1 = 1$, the value $x_2 = 1$ will be sufficient to ensure that $C = 0$ holds. By continuing this way, we reach the following orthogonal representation:

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ 0 & - & - & - & - \\ 1 & 1 & - & - & - \\ 1 & 0 & 0 & - & - \\ 1 & 0 & 1 & - & 1 \end{pmatrix}.$$

This matrix (these four vectors) show all the binary vectors with $C = 0$, each such vector is the element of precisely one of these sets. There are 30 vectors with $C = 0$. The equation $C = 1$ has only the solution vector $(101 - 0)$.

The program system XBOOLE has included the possibility for the *orthogonalization* of a given set of ternary vectors, it also offers two algorithms for the *orthogonal minimization* of such a representation.

In set theory there is the analogue rule: if two sets A and B are subsets of a set E and the complement $\overline{B}$ is taken with regard to E, then the set A can be split into two disjoint subsets of E by

$$A = A \cap (B \cup \overline{B}) = (A \cap B) \cup (A \cap \overline{B}).$$

One vector with the value $-$ in one position can be replaced by two vectors with the values 0 and 1 in this position. All the other values remain unchanged. We can also use the rules of the propositional logic:

$$y \vee \overline{y} = 1, \quad x = x \wedge 1 = x \wedge (y \vee \overline{y}) = xy \vee x\overline{y}.$$

One crucial point that will be met during the solution of many problems is the intersection of sets that are given by ternary matrices with some or even many rows. In set theory we use the *distributive law* in the following way:

$$\begin{aligned}(M_1 \cup M_2 \cup \ldots) \cap (N_1 \cup N_2 \cup \ldots) = \\ (M_1 \cap N_1) \cup (M_1 \cap N_2) \cup \ldots \cup \\ (M_2 \cap N_1) \cup (M_2 \cap N_2) \cup \ldots\end{aligned}$$

When these operations are transferred to ternary vectors, then we have to intersect each ternary vector of the first matrix with each vector of the second matrix, according to the previous definition. If not many intersections are empty, then the number of rows of the final matrix will be more or less equal to the product of the number of rows of the first and the second matrix, and this product can grow very fast indicating the amount of memory required to store the vectors, and also the time to process the resulting matrix again by a next operation.

As a tiny example we go back to the problem

$$(x_1 \vee \overline{x}_2 \vee x_4) \wedge (x_2 \vee x_3 \vee \overline{x}_5) = 1$$

and combine the two solution matrices by intersection:

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ 1 & - & - & - & - \\ 0 & 0 & - & - & - \\ 0 & 1 & - & 1 & - \end{pmatrix} \cap$$

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ - & 1 & - & - & - \\ - & 0 & 1 & - & - \\ - & 0 & 0 & - & 0 \end{pmatrix} =$$

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ 1 & 1 & - & - & - \\ 1 & 0 & 1 & - & - \\ 1 & 0 & 0 & - & 0 \\ 0 & 0 & 1 & - & - \\ 0 & 0 & 0 & - & 0 \\ 0 & 1 & - & 1 & - \end{pmatrix}.$$

The vector

$$(- - \ldots - -)$$

represents the whole B^n, the set of all binary vectors of n components.

Another operation that will be used very often is the *complement* of a given set with regard to B^n. In order to do this, we take the set

$$B^n = (- - \ldots - -)$$

and build the set that is *orthogonal* to the given set.

Example. Let be given

$$\mathbf{t} = \begin{pmatrix} 1 & 0 & - & 0 & - \end{pmatrix}$$

(describing a set of four binary vectors) and

$$B^5 = \begin{pmatrix} - & - & - & - & - \end{pmatrix}$$

We are using the same orthogonalization algorithm as above and find successively the following three ternary vectors (subsets of B^n) which are orthogonal to the given vector:

$$\mathbf{T} = \overline{\mathbf{t}} = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ 0 & - & - & - & - \\ 1 & 1 & - & - & - \\ 1 & 0 & - & 1 & - \end{pmatrix}.$$

The original vector $\mathbf{t}$ represented four binary vectors, the orthogonal matrix $\mathbf{T}$ generates 28 vectors, and 2^5 is equal to 32. For the complement of a union of several sets we are using the rule

$$\overline{A_1 \cup A_2 \cup \ldots \cup A_i} = \overline{A}_1 \cap \overline{A}_2 \cap \ldots \cap \overline{A}_i.$$

For each ternary vector of a given matrix we have to calculate the complement and thereafter to intersect all these complements.

Example. As an example we go back to the solution matrix of $x_1\overline{x}_2x_3\overline{x}_5 = 0$ (in orthogonal format) and calculate the complement (i.e. the solution of $x_1\overline{x}_2x_3\overline{x}_5 = 1$). This will be done by calculating the complement of the following four vectors separately:

$$\mathbf{t}^1 = \begin{pmatrix} 0 & - & - & - & - \end{pmatrix} \Rightarrow \overline{\mathbf{t}^1} = \begin{pmatrix} 1 & - & - & - & - \end{pmatrix},$$

$$\mathbf{t}^2 = \begin{pmatrix} 1 & 1 & - & - & - \end{pmatrix} \Rightarrow \overline{\mathbf{t}^2} = \begin{pmatrix} 0 & - & - & - & - \\ 1 & 0 & - & - & - \end{pmatrix},$$

$$\mathbf{t}^3 = \begin{pmatrix} 1 & 0 & 0 & - & - \end{pmatrix} \Rightarrow \overline{\mathbf{t}^3} = \begin{pmatrix} 0 & - & - & - & - \\ 1 & 1 & - & - & - \\ 1 & 0 & 1 & - & - \end{pmatrix},$$

$$\mathbf{t}^4 = \begin{pmatrix} 1 & 0 & 1 & - & 1 \end{pmatrix} \Rightarrow \overline{\mathbf{t}^4} = \begin{pmatrix} 0 & - & - & - & - \\ 1 & 1 & - & - & - \\ 1 & 0 & 0 & - & - \\ 1 & 0 & 1 & - & 0 \end{pmatrix},$$

followed by

$$\overline{\mathbf{t}^1} \cap \overline{\mathbf{t}^2} \cap \overline{\mathbf{t}^3} \cap \overline{\mathbf{t}^4} = \mathbf{t} = \begin{pmatrix} 1 & 0 & 1 & - & 0 \end{pmatrix},$$

and this result already has been seen as the solution of $C = 1$.

In many applications, particularly when they are based on constraints, we will not consider the complements simultaneously, the vectors t1, t2, t3 and t3 will be computed or generated sequentially and used in sequential order:

$$S = ((((B^n \setminus t^1) \setminus t^2) \setminus t^3) \setminus t^4).$$

This procedure can save a lot of memory and processing time.

V. Immediate Solution of the SAT-Problem

Using these ternary vectors as the basic data structure, we are able to solve SAT-problems directly. We will use the following small example:

$$(x_1 \vee \overline{x}_2 \vee \overline{x}_3)(x_2 \vee \overline{x}_4 \vee \overline{x}_5)(\overline{x}_1 \vee x_4 \vee x_5)(x_2 \vee x_3 \vee \overline{x}_5) = 1.$$

This equation is equivalent to the system of four single equations:

$$\begin{aligned} x_1 \vee \overline{x}_2 \vee \overline{x}_3 &= 1, \\ x_2 \vee \overline{x}_4 \vee \overline{x}_5 &= 1, \\ \overline{x}_1 \vee x_4 \vee x_5 &= 1, \\ x_2 \vee x_3 \vee \overline{x}_5 &= 1. \end{aligned}$$

The first equation now will be transformed into a ternary matrix (a *set* or *list* of ternary vectors):

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ 1 & - & - & - & - \\ 0 & 0 & - & - & - \\ 0 & 1 & 0 & - & - \end{pmatrix}.$$

This matrix shows all the vectors that satisfy the first equation. If $x_1 = 1$, then the values of the other variables are not important. If $x_1 = 0$, then $\overline{x}_2$ must be equal to 1, i.e. $x_2 = 0$. Finally, if $x_1 = 0$ and $x_2 = 1$, then x_3 must be equal to 0. Double solutions cannot exist. It is very characteristic that each vector of the matrix includes more information than the previous vectors. The number of vectors in the resulting matrix is equal to the number of variables in the disjunction. In the example each disjunction has three variables (an example of a 3-SAT-problem).

If we repeat this procedure for the other three equations, then we get the following four matrices:

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ 1 & - & - & - & - \\ 0 & 0 & - & - & - \\ 0 & 1 & 0 & - & - \end{pmatrix} \quad \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ - & 1 & - & - & - \\ - & 0 & - & 0 & - \\ - & 0 & - & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ 0 & - & - & - & - \\ 1 & - & - & 1 & - \\ 1 & - & - & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ - & 1 & - & - & - \\ - & 0 & 1 & - & - \\ - & 0 & 0 & - & 0 \end{pmatrix}$$

In order to get the final solution, these four matrices have to be combined by intersection (see above). Each line of one matrix has to be combined with each line of the next matrix, empty intersections can be omitted.

For the first and second matrix, we get,

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ 1 & - & - & - & - \\ 0 & 0 & - & - & - \\ 0 & 1 & 0 & - & - \end{pmatrix} \bigcap$$

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ - & 1 & - & - & - \\ - & 0 & - & 0 & - \\ - & 0 & - & 1 & 0 \end{pmatrix} =$$

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ 1 & 1 & - & - & - \\ 1 & 0 & - & 0 & - \\ 1 & 0 & - & 1 & 0 \\ 0 & 0 & - & 0 & - \\ 0 & 0 & - & 1 & 0 \\ 0 & 1 & 0 & - & - \end{pmatrix},$$

and the final result is equal to

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ 0 & 0 & - & - & 0 \\ - & 0 & 1 & 0 & 1 \\ 1 & 1 & - & - & 1 \\ 0 & 1 & 0 & - & - \\ 1 & - & - & 1 & 0 \end{pmatrix}.$$

This matrix of ternary vectors represents all solutions of the original SAT-problem. Since the value $-$ represents 0 as well as 1, the equation has 18 solutions.

By using these methods, many SAT-problems already can be solved. As a small example we use a combinatorial problem

on the chessboard. It is naturally well known and only will be used to demonstrate the modeling and the solution of such a problem.

Example. How many configurations can be found with n queens on a chessboard $n \times n$ not attacking each other?

We explain the procedure with a board 4×4. The generalization then will be quite easy. The queen can move horizontally, vertically and diagonally. Therefore in each column, each row and each diagonal at most one queen can be placed. We describe the possibilities by a matrix 4×4 as follows:

4	.	.	.	.
3	.	.	.	.
2	.	.	.	.
1	.	.	.	.
	1	2	3	4

Logic variables describe the position of the queens and are defined as follows:

$$x_{ij} = \begin{cases} 1 & \text{if there is a queen on the field } (i,j), \\ 0 & \text{otherwise,} \end{cases}$$

for $i, j = 1 \ldots 4$.

The requirement to place a queen in the first column can be described by the following equation:

$$x_{11} \vee x_{12} \vee x_{13} \vee x_{14} = 1.$$

The constraints can be defined by a list of *implications* which have to be satisfied simultaneously. The implication is expressed in our daily language by *if x then y*, very often the representation $x \to y$ is used. x is called the *assumption*, and y is the *conclusion*. It can be transformed into the format of a clause by

$$(x \to y) = \overline{x} \vee y.$$

A queen on the field $(1,1)$, for instance, produces the following constraints:

$$\begin{aligned} &x_{11} \to \overline{x}_{12}, \quad x_{11} \to \overline{x}_{13}, \quad x_{11} \to \overline{x}_{14}, \\ &x_{11} \to \overline{x}_{21}, \quad x_{11} \to \overline{x}_{31}, \quad x_{11} \to \overline{x}_{41}, \\ &x_{11} \to \overline{x}_{22}, \quad x_{11} \to \overline{x}_{33}, \quad x_{11} \to \overline{x}_{44}. \end{aligned}$$

The satisfaction of these constraints will not allow another queen in the first column, in the first row and in the main diagonal from $(1,1)$ to $(4,4)$.

When we eliminate the implications, then we get

$$\begin{aligned} &\overline{x}_{11} \vee \overline{x}_{12} = 1, \quad \overline{x}_{11} \vee \overline{x}_{13} = 1, \quad \overline{x}_{11} \vee \overline{x}_{14} = 1, \\ &\overline{x}_{11} \vee \overline{x}_{21} = 1, \quad \overline{x}_{11} \vee \overline{x}_{31} = 1, \quad \overline{x}_{11} \vee \overline{x}_{41} = 1, \\ &\overline{x}_{11} \vee \overline{x}_{22} = 1, \quad \overline{x}_{11} \vee \overline{x}_{33} = 1, \quad \overline{x}_{11} \vee \overline{x}_{44} = 1. \end{aligned}$$

Now we will *not* use the intersection of all these clauses, it is better to combine all these constraints into one equation and use a ternary vector directly. In doing so, we get after some simplifications the equation

$$\overline{x}_{11} \vee \overline{x}_{12}\overline{x}_{13}\overline{x}_{14}\overline{x}_{21}\overline{x}_{31}\overline{x}_{41}\overline{x}_{22}\overline{x}_{33}\overline{x}_{44} = 1.$$

The conjunction with the requirement results in

$$\begin{aligned} &(x_{11} \vee x_{12} \vee x_{13} \vee x_{14}) \wedge \\ &\quad (\overline{x}_{11} \vee \overline{x}_{12}\overline{x}_{13}\overline{x}_{14}\overline{x}_{21}\overline{x}_{31}\overline{x}_{41}\overline{x}_{22}\overline{x}_{33}\overline{x}_{44}) = 1. \end{aligned}$$

The term $(x_{11}\overline{x}_{12}\overline{x}_{13}\overline{x}_{14}\overline{x}_{21}\overline{x}_{31}\overline{x}_{41}\overline{x}_{22}\overline{x}_{33}\overline{x}_{44}) = 1$ is here the interesting part. For its representation we use a ternary vector with 16 components directly:

x_{11}	x_{12}	x_{13}	x_{14}	x_{21}	x_{22}	x_{23}	x_{24}	x_{31}	x_{32}	x_{33}	x_{34}	x_{41}	x_{42}	x_{43}	x_{44}
1	0	0	0	0	0	–	–	0	–	0	–	0	–	–	0

We see directly the queen on the field $(1,1)$ indicated by the value 1 and the attacked fields indicated by the value 0.

It is quite easy to use the structure of a ternary vector directly, without writing down all the different clauses. If we use the same approach for the other fields of the first column, then we get immediately the following matrix:

x_{11}	x_{12}	x_{13}	x_{14}	x_{21}	x_{22}	x_{23}	x_{24}	x_{31}	x_{32}	x_{33}	x_{34}	x_{41}	x_{42}	x_{43}	x_{44}
1	0	0	0	0	0	–	–	0	–	0	–	0	–	–	0
0	1	0	0	0	0	0	–	–	0	–	0	–	0	–	–
0	0	1	0	–	0	0	0	0	–	0	–	–	–	0	–
0	0	0	1	–	–	0	0	–	0	–	0	0	–	–	0

It is easy to see the four possibilities for a queen in the first column and the consequences for the other columns (fields which are still free). With a little bit of training it is possible to write down this matrix for each column directly, or to generate it by means of a small program (for larger values of n).

The second, third and fourth column will be considered in the same way, and the intersection of these four matrices shows the final result:

x_{11}	x_{12}	x_{13}	x_{14}	x_{21}	x_{22}	x_{23}	x_{24}	x_{31}	x_{32}	x_{33}	x_{34}	x_{41}	x_{42}	x_{43}	x_{44}
0	0	1	0	1	0	0	0	0	0	0	1	0	1	0	0
0	1	0	0	0	0	0	1	1	0	0	0	0	0	1	0

We can see two solutions which are horizontally and vertically symmetric:

4	.	.	Q	.
3	Q	.	.	.
2	.	.	.	Q
1	.	Q	.	.
	1	2	3	4

4	.	Q	.	.
3	.	.	.	Q
2	Q	.	.	.
1	.	.	Q	.
	1	2	3	4

We solved this problem for all values of n up to $n = 18$ and did not face any problems, except looking through all these millions of solutions. The SAT-formula that describe how 18 queens can be placed on a 18×18 chessboard not attacking each other needs 324 Boolean variables. There are 666,090,624 solutions of this problem.

VI. Order of Clauses

The order of clauses in the given expression does not change the solution, but strongly influences the calculation effort. Thus sorting of the clauses is a useful introductory step.

Let us assume a 3-SAT-problem. The partial solution set of each clause consists of $2^3 - 1 = 7$ solution vectors which

can be expressed by 3 ternary vectors. If two clauses do not overlap in any variable the intersection of these partial solution sets covers $7 \times 7 = 49$ partial solutions of six variables in $3 \times 3 = 9$ ternary vectors. The number of solutions in an intermediate partial set becomes smaller when the variables of the clauses overlap. The reduction in terms of number of ternary vectors depends on the position of the overlapping variable. The reason for that is that the orthogonalization does not create a symmetric representation. This observation leads to the idea of a lexicographic order.

A. Lexicographic Order

We assume a natural order of the variables and study first the effect of sorting the clauses according to a lexicographic order. We start with $\overline{x}_1 < x_1 < -$. This means that we write down first all clauses with $\overline{x}_1$ in the first component, followed by all components with x_1 in the first component, thereafter all clauses with $-$ in the first component. If two clauses have the same first component, then the second component is considered in the same way etc. This ordering leads to a significant reduction in terms of ternary vectors.

If we have two clauses of three literals with x_1 in the first component, then the corresponding ternary matrices of partial solution sets will have the values 100 in the first column, and we see the following situation for the intersection:

$$\begin{pmatrix} 1 & \cdots \\ 0 & \cdots \\ 0 & \cdots \end{pmatrix} \cap \begin{pmatrix} 1 & \cdots \\ 0 & \cdots \\ 0 & \cdots \end{pmatrix} \Rightarrow \cdots$$

If we have two clauses of three literals, one has x_1, the second has $\overline{x}_1$ in the first component, then we can see the following situation:

$$\begin{pmatrix} 1 & \cdots \\ 0 & \cdots \\ 0 & \cdots \end{pmatrix} \cap \begin{pmatrix} 0 & \cdots \\ 1 & \cdots \\ 1 & \cdots \end{pmatrix} \Rightarrow \cdots$$

In the first situation the intersection will have at most five ternary vectors, in the second case even only four. These are reductions of $5/9 = 55.6\%$ or even $4/9 = 44.4\%$ in terms of the number of ternary vectors.

B. Rearrange Order of Clauses by Influence of Variables

The lexicographic order is improved in Algorithm 1 slightly. The larger the number of literals of one variable the stronger is the enhancement based on its lexicographic order. For that reason Algorithm 1 counts first the number of literals for each variable of the Boolean expression and arranges the clauses controlled by these numbers starting with the highest value.

C. Rearrange Order of Clauses by Number of Used Variables

It follows from the structure of clauses in 3-SAT-problems that some calculations can be done in subspaces. As mentioned above, the intersection of two sets of different variables leads on the one hand to a solution set, where the calculated number of ternary vectors is equal to the product of the numbers of ternary vectors in the given sets. This is the worst case. On the other hand the same intersection operation reduces the number of remaining possible solutions if the variables of two sets overlap completely. This is the best case.

Algorithm 1 Rearrange order of CNF by execution of be : `InfVarOrder`$(BooleanExpression\ be)$

Require: Boolean expression $be = f(\mathbf{x})$, given in CNF
Ensure: $ben = f(\mathbf{x})$ arranged such that the first clause includes the variable of the highest occurrence in the CNF be and the last clause includes the variable of the lowest occurrence in the CNF be

```
for all var in be do
    NL[var] ← NumberOfLiterals(be, var)
end for
ben ← ∅
while be ≠ ∅ do
    var_max ← (var|NL[var] ≥ max_allvar(NL[var])
    for all clause in be do
        if var_max in clause then {move clause}
            ben ← CON(ben, clause)
            be ← DTV(be, clause)
        end if
    end for
    NL[var_max] ← 0
end while
return ben
```

Algorithm 2 is based on this fact. Like Algorithm 1 the Algorithm 2 counts first the number of literals for each variable of the Boolean expression. Secondly, for each clause is a weight calculated. This weight is the sum of values calculated in the first step associated to the variables of the clause. Thirdly, the clauses will be rearranged such that the clause with the highest weight is selected as the first. The further clauses are selected due to a decreasing order of the weights with the restriction that the number of variables in the ordered expression is extended as late as possible.

VII. Intersection versus Difference

Solving a SAT-problem means in general to find all solutions of an equation where a conjunctive form of the left hand side is equal to one. The clauses in this conjunctive form are connected by conjunction operations. Knowing the partial solution sets of all single clauses, the intersection of these sets becomes the main operation to be executed.

In case of the 3-SAT-problem each partial solution set of a single clause consists of three ternary vectors while their complement can be expressed by a single ternary vector. Hence, the question for an approach based on these complement vectors arises. In such an approach the intersection operation is replaced by the difference operation. The properties of these alternative approaches will be studied in detail in the following two subsections.

A. ISC-Based Algorithm

The ISC-based algorithm realizes basically the method described in section V and uses the restriction-based approach.

Algorithm 2 Rearrange order of CNF by execution of be : `UsedVarOrder`($BooleanExpression\ be$)

Require: Boolean expression $be = f(\mathbf{x})$, given in CNF
Ensure: $ben = f(\mathbf{x})$ arranged such, that the set of variables covered by the subset, from the first to the i-th clause, growths depending on i as small as possible

1: **for all** var in be **do**
2: $NL[var] \leftarrow$ `NumberOfLiterals`(be, var)
3: **end for**
4: $w_{max} \leftarrow 0$
5: **for all** $clause$ in be **do**
6: $WC[clause] \leftarrow 0$
7: **for all** var in $clause$ **do**
8: $WC[clause] \leftarrow WC[clause] + NL[var]$
9: **end for**
10: **if** $w_{max} < WC[clause]$ **then** {new best clause}
11: $w_{max} \leftarrow WC[clause]$
12: $clause_{max} \leftarrow clause$
13: **end if**
14: **end for**
15: $ben \leftarrow clause_{max}$
16: $be \leftarrow$ `DTV`$(be, clause_{max})$
17: **while** $be \neq \emptyset$ **do**
18: $soc_{best} \leftarrow 0$
19: **for all** $clause$ in be **do**
20: $cover \leftarrow$ `SV_ISC`$(be, clause)$
21: $soc \leftarrow$ `SV_SIZE`$(cover)$
22: **if** $soc = 3$ **then** {clause completely covered}
23: $ben \leftarrow$ `CON`$(ben, clause)$
24: $be \leftarrow$ `DTV`$(be, clause)$
25: **else if** $soc > soc_{best}$ **then**
26: $soc_{best} \leftarrow soc$
27: $clause_{best} \leftarrow clause$
28: **end if**
29: **end for**
30: $ben \leftarrow$ `CON`$(ben, clause_{best})$
31: $be \leftarrow$ `DTV`$(be, clause_{best})$
32: **end while**
33: **return** ben

That means that the given TVL in conjunctive form of the function $f(\mathbf{x})$ is first changed into the TVL of the associated disjunctive form of the function $\overline{f(\mathbf{x})}$ using De Morgan's law in the XBOOLE operation NDM [3]. This very fast operation changes the clauses into restrictions. The characteristic equation of each clause (e.g. $(b \vee \overline{c} \vee h) = 1$) is transformed into a restrictive equation $\overline{b}\, c\, \overline{h} = 0$. It is a benefit of this approach that the ternary vector of such a conjunction of literals describes directly a partial restriction set (*prs*). We use all partial restriction sets given as ternary vectors of the function $\overline{f(\mathbf{x})}$ to create in a loop partial solution sets (*pss*) by means of a complement operation (CPL [3]) which then can immediately be used to calculate the intersection (ISC [3]) with the previous intermediate solution S_{i-1}:

$$S_i = S_{i-1} \cap pss_i = S_{i-1} \cap \overline{prs}_i. \tag{1}$$

Thus, the core of the ISC-based algorithm is

$$S[i] = ISC(S[i-1], CPL(prs[i])) \tag{2}$$

where $S_0 = 1$, represented by a single ternary vector with dashes only.

An advantage of the orthogonal ternary representation of a partial solution set is the possibility that seven partial binary solution vectors are expressed by three disjoint ternary vectors. A disadvantage of this representation is the asymmetry of the columns. This asymmetry is observable by different numbers of dashes, precisely 0, 1, and 2, in the columns of the variables given in the clause. Due to the results of the analysis in Section V a controlled complement operation is required which creates zero dashes in that column of the partial solution set fitting to the column of the intermediate solution matrix having the smallest number of dashes.

A further disadvantage of a partial solution set is the a priori segmentation of the set into three subsets. The disadvantageous effects of this segmentation are:

1) The time for the comparison of these three subsets with each vector of the intermediate solution is three times higher than the processing of a single set.
2) There is an unnecessary segmentation of the intermediate solution matrix which requires more time and space in the following calculation steps.

The second effect can be explained by the following example. Let us assume that the intersection of the first $k-1$ clauses has created the intermediate solution set S_{k-1}

$$S_{k-1} = \left(\begin{array}{ccccc} a & b & c & d & e \\ \hline 1 & 0 & - & 1 & - \\ 1 & 1 & - & - & 1 \\ 0 & 1 & 1 & - & - \\ 0 & 0 & 0 & - & - \end{array}\right),$$

and the next clause k is $(c \vee d \vee e)$.

Based on (1) and (2) we get

$$S_k = S_{k-1} \cap \left(\begin{array}{ccccc} a & b & c & d & e \\ \hline - & - & 1 & - & - \\ - & - & 0 & 1 & - \\ - & - & 0 & 0 & 1 \end{array}\right) =$$

$$\left(\begin{array}{ccccc} a & b & c & d & e \\ \hline 1 & 0 & 1 & 1 & - \\ 1 & 0 & 0 & 1 & - \\ 1 & 1 & 1 & - & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & - & - \\ 0 & 0 & 0 & 1 & - \\ 0 & 0 & 0 & 0 & 1 \end{array}\right).$$

It can be seen that the first row of S_{k-1} builds solution vectors with the first and second row of the partial solution set of the clause k. The created two solution sets may be expressed by a single ternary vector. The second vector of S_{k-1} builds even three solution vectors with all three rows of the partial solution set. These three solution vectors may be expressed by

a single ternary vector, too. A significant reduction from 8 to 5 solution vectors is possible.

It should be mentioned that in this example each one of the six possible permutations of the representation of the partial solution set leads to the same number of solution vectors.

B. DIF-Based Algorithm

Another even more fundamental change of the solution philosophy arises when we merge the operations complement and intersection of (1) and (2) into the difference operation (DIF [3]). Instead of using the intersection with the partial solution set of three vectors we exclude the partial restriction set from the intermediate solution matrix.

$$S_i = S_{i-1} \setminus prs_i. \tag{3}$$

Thus, the core of the DIF-based algorithm is

$$S[i] = DIF(S[i-1], prs[i]) \tag{4}$$

where $S_0 = 1$ is represented again by a single ternary vector that includes only dashes. The exclusion of the non-solution vectors by means of the DIF-operation is quite easy, the vectors of the first matrix will be orthogonalized with regard to the vector to be eliminated and the common vectors are thrown away.

The disadvantages of the ISC-based algorithm change into advantages of the DIF-based algorithm. Instead of three vectors in a partial solution set there is a single vector in a partial restriction set (*prs*). This reduces the time for comparison for the set by a factor of three. Of cause, there is no asymmetry in the representation of *prs* so that no decision about the order of the columns is necessary. Finally, it is especially important that unnecessary segmentations of solution sets in the intermediate solution matrix are omitted.

This very important effect can be illustrated by solving the same task as in the ISC-based algorithm. The partial restriction set for the clause $(c \vee d \vee e)$ is $prs = (--000)$, because if each of the variables is equal to 0, then the clause is equal to 0 – no solution exists in this case. Based on (3) and (4), we get

$$S_k = \begin{pmatrix} a & b & c & d & e \\ \hline 1 & 0 & - & 1 & - \\ 1 & 1 & - & - & 1 \\ 0 & 1 & 1 & - & - \\ 0 & 0 & 0 & - & - \end{pmatrix} \setminus \left(\begin{array}{ccccc} a & b & c & d & e \\ \hline - & - & 0 & 0 & 0 \end{array} \right) =$$

$$\begin{pmatrix} a & b & c & d & e \\ \hline 1 & 0 & - & 1 & - \\ 1 & 1 & - & - & 1 \\ 0 & 1 & 1 & - & - \\ 0 & 0 & 0 & 1 & - \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

The DIF-based algorithm creates directly the minimal solution. In the example the partial restriction set is orthogonal to the first three vectors of the intermediate solution matrix – these vectors remain unchanged. Only the last vector must change, it will be replaced by the two vectors $(0001-)$ and (00001). The vector (00000) has been excluded. The representation of the matrix for $S_k = S_{k-1} \setminus (--000)$ only needs 5 lines instead of 8. The efficiency of this 'tiny' step becomes visible in Figure 1. Due to smaller number of rows the DIF-based Algorithm finds the solution significantly faster.

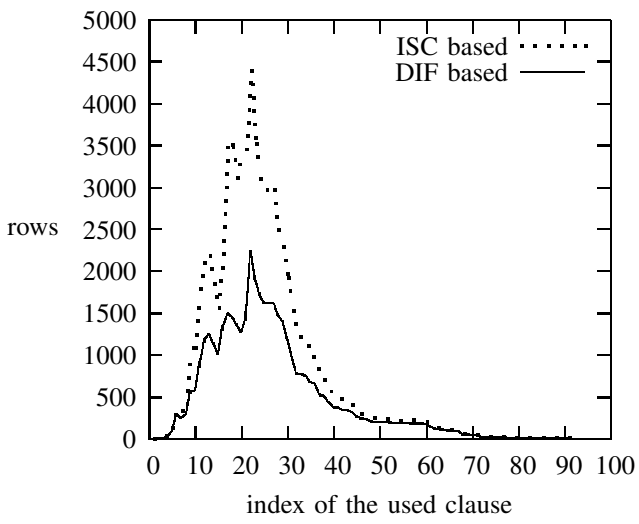

Fig. 1. Comparison of the number of ternary vectors in the intermediate TVL using the *ISC-based* and the *DIF-based* algorithm for the solution of the SAT-benchmark uf20-01 that depends on 20 variables and 91 clauses.

VIII. Splitting of Intermediate Solutions

We already have seen that the sorting of the clauses and the application of intersection and difference, resp., can result in considerable improvements of the efficiency of the solutions of SAT-problems. However, it can be necessary to deal with the large amount of intermediate ternary vectors that can reach a size where the available memory is not sufficient at all (even when at the end only a small number of solutions will appear). In this section we will show a general approach which overbears any memory restrictions while solving SAT-problems.

In order to control the needed memory space, at the beginning of the problem solution a so-called *splitlimit* will be defined. When the number of ternary vectors reaches the value of $splitlimit$, half of the set of ternary vectors existing at this point of time will be stored, together with the clauses that still have to be considered for these vectors. The solution process continues with the second half of ternary vectors as long as possible. This corresponds to the distributivity rule of set theory: it holds for sets A, A_1, A_2 with $A = A_1 \cup A_2$ that

$$A \cap B = (A_1 \cup A_2) \cap B = (A_1 \cap B) \cup (A_2 \cap B).$$

One intersection $A_1 \cap B$ will be taken into consideration immediately, the other part A_2 will be stored and handled later.

Now the processing continues with the set A_1 until all the solutions for this half have been found. The solutions can be stored, and the processing goes back to the part A_2 which has

been stored. In this way the final overall solution can be found by consideration of the two parts. Because of the orthogonality of the vectors in the intermediate solution again each solution will be found precisely once.

If one of the sets A_1 or A_2 reaches again the value of *splitlimit*, then the split can be applied again. In this way we can get, for instance, two sets A_{11} and A_{12} (if A_1 is split). Since this split can appear at many occasions, a management structure will be required that is able to care about all these subsets and the respective sets of clauses.

Table V shows some results of this approach for examples of 75 and 100 variables evaluated in [5]. The additional time for storing the subsets and some communication overhead are justified by a considerable extension of the solvability of problems.

TABLE V
THE INFLUENCE OF THE SPLITTING SIZE

splitlimit	**uf75-01**		**uf100-01**	
	time	memory	time	memory
100	105.90 s	6 MB		
1 000	25.49 s	6 MB	2876.40 s	7 MB
10 000	19.84 s	9 MB	2085.26 s	13 MB
100 000	20.03 s	34 MB	2750.87 s	59 MB
1 000 000	24.96 s	156 MB	2152.25 s	379 MB
10 000 000	23.89 s	273 MB		

It is quite obvious that it is not easy to determine the optimal value of *splitlimit*. However, the results indicate that for the given examples of 3-SAT-problems it might be in the range between 10 000 and 100 000.

IX. PARALLEL APPROACHES

Now we will indicate some possibilities to use *parallel processing* for the solution of SAT-problems. This possibility can be based on the fact that the two problems that exist after the splitting can be solved completely independent on each other. If one processor handles the set A_1, another processor the set A_2, then it is only necessary to copy the set of clauses still to be processed, and the two processors can continue working on the two smaller sub-problems. If there is a larger number of processors, then several splittings can be used, with a rather small amount of communication overhead. The communication mainly has to deal with the splitting of the intermediate matrices and the sending of the remaining clauses.

A. Controlled Distribution of Subtasks

In [6], [5] the following intelligent strategy has been used. Basically it follows the split-approach introduced in Section VIII but one master processor distributes the subtasks to available client processors. In case of parallel computing the additional question of the distribution of the split subfunctions $f_i(\mathbf{x})$ arises. The answer to this question will be influenced by two opposite effects:

- there can be a large overhead for communication when many small subtasks are distributed to the processors,
- it can be very time-consuming when a few large subtasks must be solved in parallel.

In order to control the distribution of the subtasks, two parameters are used:

1) *interimlimit*: the number of conjunctions required to split an intermediate solution function into subfunctions, and
2) *distlimit*: the number of conjunctions of the intermediate solution allocated to a processor as source of a subtask to solve.

The *interimlimit* affects to solution process of the SAT-problem in two ways. On the one hand, if the number of intermediate solutions is smaller than the value of *interimlimit* then the problem is solved very fast on a single processor and time-consuming distributions are completely avoided. On the other hand, if the number of intermediate solutions is larger than the value of *interimlimit*, there are enough intermediate solutions used to create subtasks for the available processors. An optimal range of the *interimlimit* is $10^2 \ldots 10^4$.

Table VI shows the effect of the *distlimit* for *splitlimit* = 10,000, *interimlimit* $>$ 1,000, and 5 processors in order to solve SAT-problems of 50, 75 and 100 variables.

TABLE VI
THE INFLUENCE OF DISTLIMIT

	time in seconds		
distlimit	**uf50-01**	**uf75-01**	**uf100-01**
1	4.59	7.20	342.73
5	0.97	5.25	343.25
10	0.52	5.64	342.35
25	0.25	7.10	349.48
50	0.17	7.18	371.30
100	0.21	6.65	584.16
250	0.16	6.07	671.51
500	0.14	9.92	943.94
1000	0.17	18.89	

The first column of Table VI shows that the time for communication grows for smaller values of the control parameter *distlimit* and influences significantly the runtime of the smallest evaluated SAT-problem of 50 variables. Vice versa, it can be observed from the last two columns of Table VI that the time to solve larger subtasks dominates the time for communication in case of SAT-problems of larger numbers of variables. Hence, a small value for the control parameter *distlimit* should be chosen.

A more detailed analysis is given in Table VII. Here we have used 8 processors which solve the SAT-problem **uf100-01** of 100 variables controlled by the parameter values *interimlimit* $>$ 1,000 that determines the number of basic intermediate solutions, and *splitlimit* = 10,000 for the solution of the subtasks. In this experiment the runtime of the equal distribution is compared in contrast to the allocation of vectors on demand controlled by the value of *distlimit*.

A *distlimit* of 1 means that one vector will be allocated to a processor as soon as it is idle. This processor continues to process this single vector and the clauses remaining for this vector. This strategy should be the most appropriate strategy to get a balanced distribution of the computational efforts. The almost constant runtime of all 8 processors is reached due

TABLE VII
EQUAL DISTRIBUTION VERSUS ALLOCATION OF VECTORS ON DEMAND – IT MEANS FOR THE CPU LABELED IN THE FIRST COLUMN: T THE TIME IN SECONDS, AND R THE NUMBER OF REQUESTS FOR A SUBTASK TO SOLVE

evaluated CPU	**equal distribution**		**distlimit**					
			50		**5**		**1**	
	T	*R*	*T*	*R*	*T*	*R*	*T*	*R*
CPU 1	49.11	1	79.41	3	166.56	46	175.38	173
CPU 2	62.37	1	333.19	2	201.62	12	175.17	230
CPU 3	193.25	1	169.05	4	167.34	30	176.71	108
CPU 4	24.33	1	232.87	3	218.29	8	175.05	132
CPU 5	109.21	1	181.80	5	179.84	30	177.70	80
CPU 6	633.19	1	148.88	1	180.01	22	175.45	90
CPU 7	176.08	1	115.84	3	166.38	55	175.07	148
CPU 8	106.01	1	116.86	3	166.42	28	175.10	193
max.	**633.19**		**333.19**		**218.29**		**177.70**	

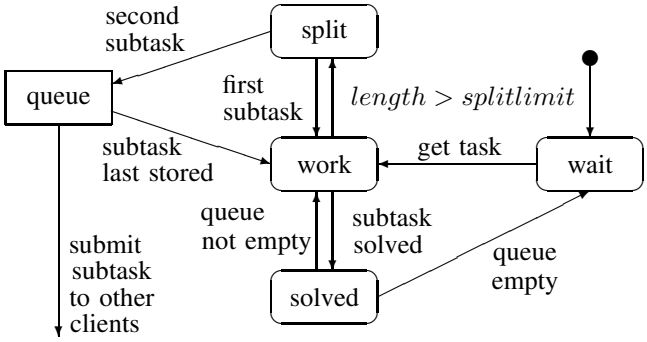

Fig. 2. Actions of the client in the algorithm: *Parallel Backtracking*.

to the different numbers of solved subtasks indicated by the values in the column labeled be R (requests for a subtask).

An *equal distribution* of the intermediate solutions, however, splits the set of vectors at the beginning into equal parts, and each processor continues on its own. This is the worst case because the longest runtime for a subtask determines the overall runtime.

Table VII shows that $distlimit = 1$ is a good choice. Extending the value of *distlimit* to 50 approximately doubles the runtime.

B. Parallel Backtracking

The runtime required to solve SAT-problems depends for the approach of controlled distribution (see Subsection IX-A) on the values of the three control parameters *interimlimit*, *distlimit* and *splitlimit*. The last is necessary to fix the basic memory problem. The other two parameters are needed for control only. In order to avoid the influence of these two parameters, an alternative approach called *Parallel Backtracking* was suggested in [6], [5].

The benefits of the *Parallel Backtracking* are:

- it organizes its parallel work itself,
- it utilizes a just suitable number of processors, and
- it needs a single parameter only: the *splitlimit*.

Figure 2 shows the actions of the algorithm running on the processors ready to solve the SAT-problem. Initially, the client is waiting for a task to solve in the action *wait*. After receiving a task the client solves it in the action *work*. If the given task has been solved (initially it is the complete task and later on it will be a subtask), the client continues with the action *solved*. In this action the client checks whether the queue includes a further subtask. If there is a further subtask to solve, the client takes the last stored subtask and solves it in the action *work*; otherwise the client returns to the action *wait*.

In the case that the client has calculated more than *splitlimit* intermediate solutions, the client moves to the action *split* where the client splits the intermediate solutions into two subtasks; the client stores the second subtask into the queue and continues to solve its own first subtask. The queue is used commonly by all clients.

Table VIII shows that the approach of *Parallel Backtracking* scales very well. The *splitlimit* for the approach of *Parallel Backtracking* was fixed to 10,000. Of cause, when the SAT-problem can be solved in a couple of seconds, adding more processors does not speed up the solution process because only the created subtasks can be solved by the involved client processors. This can be seen in the column of the 3-SAT-problem `uf75-01` of 75 variables. The high quality of the approach of *Parallel Backtracking* becomes visible for the 3-SAT-problem `uf100-01` of 100 variables in the last column of Table VIII. The achieved speedup is greater than 90% for up to 24 processors. That means that up to this number of processors the required solution time is almost the quotient of time needed be a single processor divided by the number of processors that have been used in parallel.

TABLE VIII
SOLVING 3-SAT-PROBLEMS OF 75 AND 100 VARIABLES USING THE PARALLEL BACKTRACKING ON DIFFERENT NUMBERS OF PARALLEL PROCESSORS

	processing time in s	
number of processors	uf75-01	uf100-01
1	19.94	1391.16
2	11.13	696.87
4	6.53	362.39
8	3.96	175.71
12	3.07	120.09
16	2.92	89.81
24	2.90	63.18
32	2.47	50.94
40	2.38	40.19
48	2.33	36.98
56	2.27	30.66
64	2.24	29.94

Both approaches *Controlled Distribution of Subtasks* and *Parallel Backtracking* were implemented using the same library MPI (message passing interface). The benefits of the *Parallel Backtracking* are visible in Figure 3.

C. Computation on the GPU

NVIDIAs Compute Unified Device Architecture *CUDA* allows to use the huge number of processor cores of the Graphics Processing Units *GPU* to speed up time-consuming tasks. In [7] we studied the utilization of the *GPU* for a special SAT-problem, called Unate Covering Problem *UCP*. Such problems must be solved in circuit design.

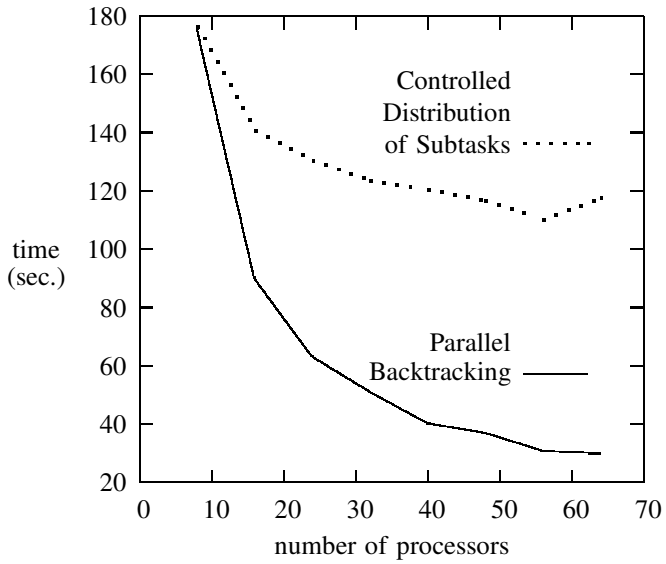

Fig. 3. Comparison of the approaches *Controlled Distribution of Subtasks* (dotted line) and *Parallel Backtracking* (solid line) for the 3-SAT-problem *uf100-1* of 100 Boolean variables.

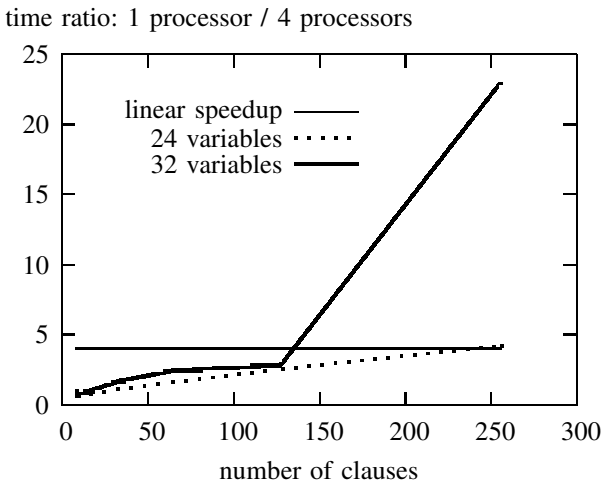

Fig. 4. Time ratio between the sequential solution on a single processor core and the parallel solution *Uniform Distribution* on 4 processor cores for solving the SAT-problem *UCP* using the DIF-based algorithm.

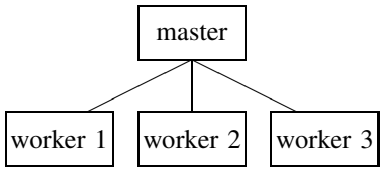

Fig. 5. Master – worker architecture of a PC with 4 cores.

There are two peculiarities of the *UCP* in comparison to the general SAT-problem:

1) all variable in the clauses of the SAT-formula appear in positive polarity, and
2) the smallest number of assignments 1 to the variables which solve the SAT-equation is wanted.

We do not want to repeat the details of the used approach of matrix multiplication on the GPU of this research in an early stage. Using GPU of 64 cores to solve a *UCP* of 256 clauses with 16 variables the run time was reduced by a factor of 3.4 [7]. We are sure that much larger improvements will be possible. One additional technical problem is that all data must be moved first from the main memory to the memory of the GPU before the power of the GPU can be used for computation. After the computation additional time is needed to move the results back to the main memory.

D. Computation on a Multi-Core CPU

The recent improvements in computer hardware [8] increase the computation power of modern PCs and lead to challenges for software development due to the multi-core architecture. How the relatively small number of processor cores can be utilized efficiently? In [9] we published some approaches which show that a strong improvement can be reached even with 4-cores of the CPU of a modern PC. As discussed already in Subsection IX-C we solved the special SAT-problem *UCP*.

First we simply split the Boolean space into four subspaces of equal sizes and applied the DIF-based algorithm of Subsection VII-B. This approach, called *Uniform Distribution*, has a bad load-balancing but reached for the largest benchmark (256 clauses of 32 variables) the shortening form 734.171 seconds using one single processor core to 31.931 seconds using all four cores of the CPU in parallel. Hence, the runtime could be reduced in this case by a factor of 22.99 which is significantly larger than the number of the used 4 processor cores. Figure 4 shows the details of these benchmark experiments.

Next we improved the load-balancing in the approach *Adaptive Distribution* in such a way that a larger number of subtasks is created by a master process and assigned on demand to three worker processes. Figure 5 shows the used architecture of the 4 processor cores.

Despite the restriction from four to three working processor cores the much better load-balancing and the smaller subtasks reduced the runtime of the previously mentioned example to 3.099 seconds. Hence, the time could be reduced in this case by an additional factor of 10.3 in comparison to the *Uniform Distribution* using 4 cores which is an improvement in comparison to a single processor core of 236.9. The reason of this impressive speedup is based on a special utilization of the implemented concurrent approach. Each worker sends the results of the unate covering problem for the assigned subspace to the master process. These results include both the minimal number of values 1 in this partial solution and the number of such minimal solutions. The master process handles the partial solutions in the following way:

- If the master process already knows solutions with a smaller number of values 1, the received solutions with a larger number of values 1 are omitted immediately.
- If the master process already knows solutions with the same number of values 1, the received solutions are accumulated.

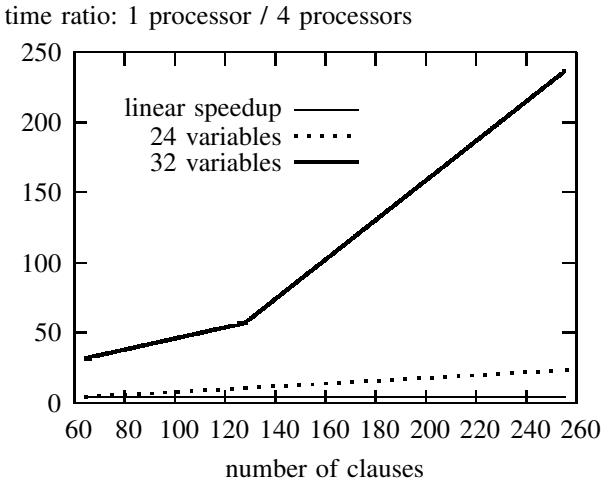

Fig. 6. Time ratio between the sequential solution on a single processor core and the parallel solution *Adaptive Distribution* on 4 processor cores for solving the SAT-problem *UCP* using the DIF-based algorithm.

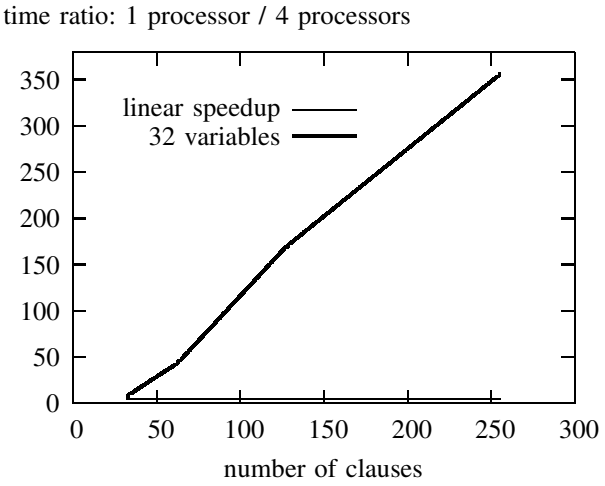

Fig. 7. Time ratio between the sequential solution on a single processor core and the parallel solution *Intelligent Master* on 4 processor cores for solving the SAT-problem *UCP* using the DIF-based algorithm.

- If the master process knows so far only solutions with more values 1, the stored accumulated solution is replaced by the new better solution.

Using this simple algorithm, the master process knows the smallest number of values 1 found so far by the concurrent worker processes. On each request of a worker for the next subtask, the master process sends both the number of the next subspace and the smallest solution found so far. This information helps the worker process to simplify the evaluation algorithm because large solutions must not be taken into account anymore. Figure 4 shows the details of these benchmark experiments.

The serious super-linear speedup was reached because the available set of processor cores is not used only for the computation of the assigned subtasks, but mainly as source of knowledge that restricts the effort for the subsequent subtasks.

Finally, we created the *Intelligent Master* approach. The knowledge about the smallest number of values 1 of a solution known so far is not only used by the workers but also by the master. This approach relies on the aim that solution vectors with the smallest number of values 1 are wanted. A subspace is defined by fixing a certain number of variables; (x1 = 0; x2 = 1; x3 = 1; x4 = 0; x5 = 1) defines, for instance, one of 32 subspaces where the first 5 variables are fixed. If a solution of two values 1 is already known, it can be concluded that no solution results from this subspace since already three variables have the value set to 1.

The measured runtime of the previously mentioned *UCP* example of 32 variables in 256 clauses is 2.061 seconds. Hence, the runtime could in this case be reduced by an additional factor of 1.5 in comparison with the *Adaptive Distribution* using 4 cores which is an improvement in comparison to a single processor core of 356.2. Figure 4 shows the serious super-linear speedup reached by the *Intelligent Master* approach for benchmark experiments of 32 variables. The distributed solution itself is a source for the remarkable super-linear speedup, even when a small number of cores is used.

X. Higher Level Models

Mapping a real-life problem into s SAT-model is a very strong formalization. Relationships of the real life problem are expressed by a huge number of very simple clauses. Properties of the given problem are distributed over a large number of clauses and will not be explicitly visible anymore.

Alternatively higher-level models can be applied. We introduce this higher-level approach by means of a well-known example.

Over the last years a Japanese game with the name **Sudoku** became very popular. It is played mostly on a board with 9×9 fields, but other square numbers are also possible, such as 4×4 or 16×16 or even 25×25. It is easy to understand and a bit challenging for human beings, and it can be used comfortably to spend waiting time on airports or similarly. But there are also mathematical and logical properties that deserve some attention.

There is a quadratic board of, for instance, size 9×9. In each column, in each row and in nine sub-squares of size 3×3 the values $1, \ldots, 9$ have to be set such that in each column, in each row and in each sub-square each value is used once and only once.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Some values already have been set, and the other values have to be found according to the existing values. We enumerate the columns from the left to the right and the rows

bottom-up (in the same way as a normal planar coordinate system).

We know at least two papers that are using SAT-models of the game and existing SAT-solvers for the solution of the problem [10], [11]. These papers used the following approach: a binary variable x_{ijk} describes the content of the field $(i,j), 1 \leq i,j,k \leq 9$:

$$x_{ijk} = \begin{cases} 1 & \text{if the value on the field } (i,j) = k \\ 0 & \text{if the value on the field } (i,j) \neq k \end{cases}$$

The transformation into a SAT-problem uses several steps:

$$(x_{ij1} \vee x_{ij2} \vee x_{ij3} \vee x_{ij4} \vee x_{ij5} \vee x_{ij6} \vee x_{ij7} \vee x_{ij8} \vee x_{ij9}) = 1$$

expresses the requirement that one of the numbers $1, \ldots, 9$ must be used for the field (i,j). Such a disjunction must be written for each field of the board which results in 81 clauses which must be satisfied simultaneously.

The second step expresses all the constraints for rows, columns and squares as clauses as well. For example for the field $(1,1)$ and the value 1 on this field, no other value can be on this field:

$$x_{111} \rightarrow \overline{x}_{112},\ x_{111} \rightarrow \overline{x}_{113},\ \ldots,\ x_{111} \rightarrow \overline{x}_{118},\ x_{111} \rightarrow \overline{x}_{119}.$$

The same set of clauses must be written for the other values $2, \ldots, 9$ on the same field. The requirements for the first column can be expressed in the same way:

$$x_{111} \rightarrow \overline{x}_{121},\ x_{111} \rightarrow \overline{x}_{131},\ \ldots,\ x_{111} \rightarrow \overline{x}_{181},\ x_{111} \rightarrow \overline{x}_{191}.$$

The constraints for the row are given as

$$x_{111} \rightarrow \overline{x}_{211},\ x_{111} \rightarrow \overline{x}_{311},\ \ldots,\ x_{111} \rightarrow \overline{x}_{811},\ x_{111} \rightarrow \overline{x}_{911},$$

and finally we must consider the value 1 in the respective square:

$$x_{111} \rightarrow \overline{x}_{221},\ x_{111} \rightarrow \overline{x}_{321},\ \ldots,\ x_{111} \rightarrow \overline{x}_{231},\ x_{111} \rightarrow \overline{x}_{331}.$$

Again all these clauses have to be written for all numbers from 1 to 9 and finally for all fields. By using the rule

$$x \rightarrow y = \overline{x} \vee y$$

the whole set of implications can be transformed into disjunctions, all of them must be satisfied at the same time, and this is the problem in SAT-format. Each satisfying set of values for the binary variables is a solution of the Sudoku.

We will show that this game easily can be modeled by using a logic equation, with *ternary vectors* as the most appropriate data structure. Actually, the logic equation does not even have to be written down; **the ternary vectors can be generated directly**.

We are using the same encoding as the two other papers mentioned above:

$$x_{ijk} = \begin{cases} 1 & \text{if the value on the field } (i,j) = k \\ 0 & \text{if the value on the field } (i,j) \neq k \end{cases}$$

The constraints can be stated by one single conjunction for each number on each field:

$$\begin{aligned} K_{111} = & x_{111}\overline{x}_{112}\overline{x}_{113}\overline{x}_{114}\overline{x}_{115}\overline{x}_{116}\overline{x}_{117}\overline{x}_{118}\overline{x}_{119} \wedge \\ & \overline{x}_{121}\overline{x}_{131}\overline{x}_{141}\overline{x}_{151}\overline{x}_{161}\overline{x}_{171}\overline{x}_{181}\overline{x}_{191} \wedge \\ & \overline{x}_{211}\overline{x}_{311}\overline{x}_{411}\overline{x}_{511}\overline{x}_{611}\overline{x}_{711}\overline{x}_{811}\overline{x}_{911} \wedge \\ & \overline{x}_{221}\overline{x}_{231}\overline{x}_{321}\overline{x}_{331}. \end{aligned}$$

This conjunction describes completely the setting of 1 on the field $(1,1)$ and **all the consequences**. There are 729 of such conjunctions which are defined uniquely. It is important to understand that not only the requirement in terms of 9 variable x_{ijk} of each field (i,j) is taken into consideration, but the conjunctions K_{ijk} so that **all the consequences** resulting from a given setting are used immediately. The existing knowledge or constraints are directly built into the ternary vectors.

Now we must express the possibilities of the game. In order to do this, we can use one of the following four types of equations.

1) The equation

$$\begin{aligned} & (K_{111} \vee K_{112} \vee K_{113} \vee K_{114} \vee \\ & K_{115} \vee K_{116} \vee K_{117} \vee K_{118} \vee K_{119}) = 1 \end{aligned}$$

describes that one of the 9 values must be assigned to one field (the field $(1,1)$ is only an example).

2) The equation

$$\begin{aligned} & (K_{111} \vee K_{121} \vee K_{131} \vee K_{141} \vee \\ & K_{151} \vee K_{161} \vee K_{171} \vee K_{181} \vee K_{191}) = 1 \end{aligned}$$

describes that the values 1 must be assigned to one of the fields in a column (column 1 and value 1 are only examples).

3) The equation

$$\begin{aligned} & (K_{111} \vee K_{211} \vee K_{311} \vee K_{411} \vee \\ & K_{511} \vee K_{161} \vee K_{711} \vee K_{811} \vee K_{911}) = 1 \end{aligned}$$

describes that the values 1 must be assigned to one of the fields in a row (row 1 and value 1 are only examples).

4) The equation

$$\begin{aligned} & (K_{111} \vee K_{121} \vee K_{131} \vee K_{211} \vee \\ & K_{221} \vee K_{231} \vee K_{311} \vee K_{321} \vee K_{331}) = 1 \end{aligned}$$

describes that the values 1 must be assigned to one of the fields in a subsquare (the first subsquare and value 1 are only examples).

Each type of these equations generates a system of 81 disjunctions that must be satisfied at the same time. They are completely equivalent, one system can be selected once and for ever. All the conjunctions are represented by ternary vectors, and **this representation can be generated before any real game** which is given by special settings. Each ternary vector will have 729 components, and all intersections from the left to the right have to be calculated. Since the values which have already been set result in one vector for the given field, many of these matrices will have only one row (30 in the given example). Therefore it is advisable (however, not necessary),

to intersect the single vectors first, because thereafter many intersections will be empty. This algorithm does not need any considerations for special cases. If there are no solutions, then the intersection will be empty at a given point of time, if there is a unique solution, then we will have precisely one vector in the final intersection, and more than one solution will be expressed by the respective number of vectors.

The solution set S consists of all binary vectors of the length 729 that solve the SAT-problem of the given Sudoku. Each solution vector includes exactly 81 values 1 that indicate the solution numbers associated to the fields. The remaining 648 components of each solution vector carry the value 0. Thus, by taking the index (i, j, k) of the values 1 in the solution vector, a representation of the value k in the field(i, j) of column i and row j can be established.

As a summary we can see that the solution of the problem has two steps. The first phase covers the modeling of the problem and the calculation of partial solution sets (or solution candidates). Of course the first phase depends on the problem to be solved – in our case any Sudoku game.

The second phase mainly considers the different action possibilities and combines these possibilities by $\vee$ operations. The intersections of the different possible actions are evaluated by the intersections of the respective ternary vectors.

The advantage of this new approach in comparison with the known traditional SAT-models is the simultaneous assignment of values to many variables. In case of a 9×9 Sudoku a single assignment specifies additionally 28 variables of the solution space, and this strongly restricts the remaining search space.

XI. One Interesting Example and Experimental Results

There are many interesting problems of the game – one of these problems, for instance, is the question: How many settings are necessary to define one and only one unique solution? In [12] an example with 16 settings has been given that has precisely two solutions. It is not known whether there are other examples with 16 settings and one or two solutions. It is known, however, that there are many unique solutions when the number of settings is equal to 17. The concept of a unique solution still has to be defined properly because such operations like relabeling of entries, reflection, rotation, ... of a valid Sudoku give other valid Sudokus.

We used the Sudoku of Figure 8 as an example. The two solutions are shown in Figure 9. The second solution can be found easily by exchanging 8 and 9 in the first solution.

The program has built the game matrix (which can also be stored before any real game) in the first phase using 515 milliseconds. The two solutions have been determined by the solution program in the second phase after 16 milliseconds.

In case of a 16×16 board the matrix of the partial solution sets required approximately 2 Megabyte. Each row of this matrix includes one value 1, 54 values 0. The remaining values of the 4096 variables are filled with dashes. Therefore we decided to store only the index values of the elements with the value 0 and 1 and to generate any vector of a partial solution set at the time when it is required. Without any other changes

1								5
				3				
		2		4				
	3	4				7		
			2		6			1
2					5			
	7						3	
					1			

Fig. 8. Difficult Sudoku example having only 16 settings.

Solution 1

1	8	3	9	6	7	4	2	5
4	6	9	5	3	2	8	1	7
7	5	2	1	4	8	6	9	3
6	2	1	4	7	3	5	8	9
5	3	4	8	1	9	7	6	2
8	9	7	2	5	6	3	4	1
2	1	6	3	8	5	9	7	4
9	7	5	6	2	4	1	3	8
3	4	8	7	9	1	2	5	6

Solution 2

1	9	3	8	6	7	4	2	5
4	6	8	5	3	2	9	1	7
7	5	2	1	4	9	6	8	3
6	2	1	4	7	3	5	9	8
5	3	4	9	1	8	7	6	2
9	8	7	2	5	6	3	4	1
2	1	6	3	9	5	8	7	4
8	7	5	6	2	4	1	3	9
3	4	9	7	8	1	2	5	6

Fig. 9. All solutions of the Sudoku given in Figure 8 having only 16 settings.

the problem of a 16×16 Sudoku that maps into a problem of 4096 variables and 111616 clauses could be solved within about two and a half minutes.

XII. Applications and Experimental Results

Based on this methodology, many other problems have been solved. It will not be very difficult to apply the same methodology.

1) It is expected that on a chessboard of size $n \times n$ with k additional pawns $n + k$ queens can be placed without threatening each other. Here one solution has been represented for one pawn on a board of size 8×8.

							Q
		Q					
				Q			
	Q			**P**		Q	
				Q			
Q							
					Q		
			Q				

One result for two pawns is shown next.

							Q
				Q			
		Q					
Q				**P**	Q		
			Q				
	Q	**P**				Q	
				Q			
		Q					

2) The case of $k = 0$ is the "normal" problem of arrangements of queens on a chessboard $n \times n$ that has been solved as well up to $n = 18$.
3) There are many problems asking for minimum and maximum numbers, for instance, how many bishops are **at least** required to cover all the fields on a chessboard, or how many bishops can at most be placed on a chessboard without covering each other etc. These problems also have been solved on boards of size $m \times n$ for many values of m and n.
4) The same relates to graph problems, such as Hamiltonian and Eulerian paths in a graph, the minimum number of nodes with a given property or the maximum number of nodes with a given property etc.
5) As a last example we will show the solution of coloring problems. Our method can be applied to color any graph. As example we use the graph called *Birkhoff's Diamond* shown in Figure 10 (a).
The structure of a graph can be described using an adjacency matrix. A value 1 in the row i and column j indicates an edge from node i to node j in the graph. In case of an undirected graph we get a symmetric adjacency matrix. The graph *Birkhoff's Diamond* has the following adjacency matrix.

$$A_{BD} = \begin{pmatrix} 0100011100 \\ 1010000110 \\ 0101000010 \\ 0010100011 \\ 0001011001 \\ 1000101000 \\ 1000110101 \\ 1100001011 \\ 0111000101 \\ 0001101110 \end{pmatrix}$$

Using the adjacency matrix (5) the partial solution sets can be generated directly. The logic variables describe whether a certain color is assigned to a node of the graph or not. Hence, the number of required variables is equal

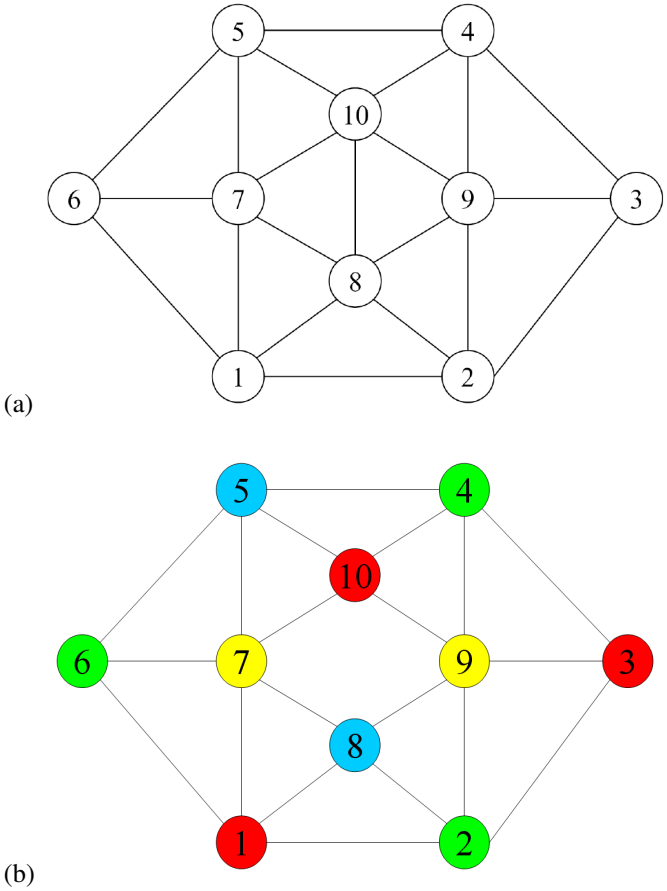

Fig. 10. Birkhoff's diamond: (a) uncolored graph, (b) one solution using 4 colors.

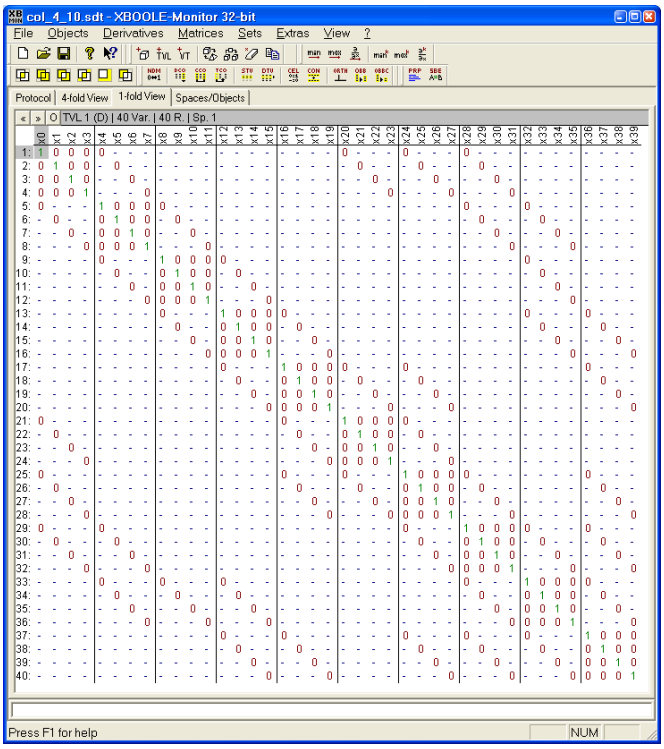

Fig. 11. Generated matirx of the partial solution sets to color the graph of Birkhoff's diamond using four colors.

TABLE IX
CALCULATION OF ALL SOLUTIONS TO COLOR THE BIRKHOFF'S DIAMOND USING 3, 4 OF 5 COLORS

number of				time in
nodes	colors	variables	solutions	seconds
10	3	30	0	0.00
10	4	40	576	0.00
10	5	50	40800	0.02

TABLE X
CALCULATION OF ALL SOLUTIONS TO COLOR THE BIRKHOFF'S DIAMOND AND GRAPHS THAT INCLUDE TWO OR FOUR SUCH GRAPHS USING 4 COLORS.

number of				time in
nodes	colors	variables	solutions	seconds
10	4	40	576	0.00
20	4	80	99888	0.20
40	4	160	100800	4.97

to the product of the number of nodes and considered colors.

$$x_{cn} = \begin{cases} 1 & \text{if the color on the node } n = c \\ 0 & \text{if the color on the node } n \neq c \end{cases}$$

These partial solution sets cover the restrictive rules. When a color is assigned to one node, it is not allowed that:

a) another color is assigned to the same node, and
b) the some color is assigned to another node connected by an edge.

Figure 11 shows the generated matrix of the partial solution sets (*mpss*) in the XBOOLE monitor [3], [4]. Each row represents a conjunction K_{cn} covering seven, eight or nine components.

The second phase of the new two-phase SAT-solver is controlled by the requirement clauses. For graph coloring we have the simple requirement that there must be one color assigned to each node of the graph. In order to find all allowed assignments of four colors for the graph of Figure 10, we must solve the equation:

$$\bigwedge_{i=1}^{10} (K_{1\,i} \vee K_{2\,i} \vee K_{3\,i} \vee K_{4\,i}) = 1 \ . \qquad (5)$$

The time to solve this equation using the operations `UNI` and `ISC` of XBOOLE [3] was less than a single time-tick (15 ms). Figure 10 (b) shows one of the 576 solutions that have been found.

Two experiments demonstrate the power of this approach. In the first experiment we calculated all solutions to color Birkhoff's diamond using three, four or five colors. Table IX summarizes these results.

In the second experiment we created several larger graphs: we combined first two Birkhoff's diamonds using some additional edges and thereafter four Birkhoff's diamonds in a similar way. Table X summarizes these results.

XIII. CONCLUSIONS

There are several results presented in this paper.

1) Many finite discrete constraint-related problems can be modeled as a SAT-problem. It has been shown that it is not necessary to write down the huge number of clauses of the conjunctive forms which must be solved by a SAT-solver. Based on the explored properties of the problem, it is possible to generate partial solution sets of the restrictive properties of the problem.
2) A new implicit two-phase SAT-solver has been used. In the first phase this SAT-solver creates partial solution sets which are used in the second phase to calculate the solution without any further decisions.
3) The matrix of the partial solution sets describes general constraints of the problem without any consideration of clauses.
4) The use of the partial solution sets in the second phase of the SAT-solver allows to solve the SAT-problems very fast. The partial solution sets help to restrict significantly the enormous search space. The remaining clauses of the problem are replaced by unions of partial solution sets which speed up the solution procedure.
5) In many applications the Boolean modeling can be considered as very efficient, and it is not necessary to develop special algorithms; it is much easier to use a general methodology based on ternary vectors.

REFERENCES

[1] B. Steinbach, "XBOOLE – A Toolbox for Modelling, Simulation, and Analysis of Large Digital Systems. System Analysis and Modeling Simulation." Gordon & Breach Science Publishers, 1992, vol. 9, no. 4, pp. 297–312.

[2] D. Bochmann and B. Steinbach, *Logikentwurf mit XBOOLE*. Berlin: Verlag Technik, 1991.

[3] C. Posthoff and B. Steinbach, *Logic Functions and Equations – Binary Models for Computer Science*. Dordrecht, The Netherlands: Springer, 2004.

[4] B. Steinbach and C. Posthoff, *Logic Functions and Equations – Examples and Exercises*. Springer Science + Business Media B.V., 2009.

[5] W. Wessely, "Parallele Lösung großer Boolescher Probleme," Master's thesis, Freiberg University of Mining and Technology, 2009.

[6] B. Steinbach, W. Wessely, and C. Posthoff, "Several Approaches to Parallel Computing in the Boolean Domain," in *1st International Conference on Parallel, Distributed and Grid Computing (PDGC 2010)*, P. Chaudhuri, S. Ghosh, R. K. Buyya, J. Cao, and D. Dahiya, Eds. Solan, H.P., India: Jaypee University of Information Technology Waknaghat, October 2010, pp. 6–11.

[7] E. Paul, B. Steinbach, and M. Perkowski, "Application of CUDA in the Boolean Domain for the Unate Covering Problem," in *Boolean Problems, Proceedings of the 9th International Workshops on Boolean Problems*, B. Steinbach, Ed. Freiberg: Freiberg University of Mining and Technology, September 2010, pp. 133–142.

[8] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: the Hardware/Software Interface*. Burlington, USA: Morgan Kaufmann Publishers, 2009.

[9] B. Steinbach and C. Posthoff, "Parallel Solution of Covering Problems Super-Linear Speedup on a Small Set of Cores," in *GSTF International Journal on Computing, Global Science and Technology Forum (GSTF)*, vol. 1, no. 2, Singapore, 2011, pp. 113–122.

[10] I. Lynce and J. Ouaknine, "Sudoku as a SAT Problem," in *Proc. 9th International Symposium on Artificial Intelligence and Mathematics*, 2006.

[11] T. Weber, "A SAT-based Sudoku Solver," in *LPAR-12, The 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 2005, pp. 11–15, Short Paper Proceedings.

[12] G. Royle, "Minimum Sudoku." [Online]. Available: http://www.csse.uwa.edu.au/~gordon/sudokumin.php

[13] W. Gropp, R. Thakur, and E. Lusk, *Using MPI-2: Advanced Features of the Message Passing Interface*. MA, USA: MIT Press, Cambridge, 1999.