The log-support encoding of CSP into SAT

Marco Gavanelli[1]

Dept. of Engineering, Ferrara University,
WWW home page: `http://www.ing.unife.it/docenti/MarcoGavanelli/`

Abstract. It is known that Constraint Satisfaction Problems (CSP) can be converted into Boolean Satisfiability problems (SAT); however how to encode a CSP into a SAT problem such that a SAT solver will efficiently find a solution is still an open question. Various encodings have been proposed in the literature. Some of them use a logical variable for each element in each domain: among these very successful are the *direct* and the *support* encodings. It is known that a SAT solver based on the DPLL procedure obtains a propagation similar to Forward Checking on a direct-encoded CSP, and to Maintaining Arc-Consistency on a support-encoded CSP.
Other methods, such as the log-encoding, are more compact, and use a logarithmic number of logical variables to encode domains. However, they lack the propagation power of the direct and support encodings, so many SAT solvers perform better on direct/support encodings than in the log-encoding, as witnessed by many works in the literature.
In this paper, we propose a new encoding that combines the log and support encodings. The new encoding, called *log-support*, has a logarithmic number of variables, and uses support clauses to obtain improved propagation. Experiments on Job-Shop scheduling problems and randomly-generated problems show the effectiveness of the proposed approach, with respect to other popular approaches.

1 Introduction

One of the methodologies for solving Constraint Satisfaction Problems (CSP) relies on the conversion into a different type of problem, e.g., the boolean satisfiability (SAT) problem, which has the same power from a complexity point of view. This methodology has obvious advantages, such as the wide availability of free, efficient, SAT solvers. SAT solvers have recently reached significant levels of efficiency, and new solvers are proposed, tested and compared every year in annual competitions [2, 21]. There are both complete SAT solvers, based on systematic search (typically, variants of the DPLL procedure, by Davis, Putnam, Logemann, and Loveland [5]), and incomplete solvers, often based on local search.

In general, there are different ways to produce a SAT instance from a given CSP, usually called *encodings* [24, 18]. Very popular encodings assign a logical variable to each possible element of a CSP domain, i.e., for each CSP variable i and each value v in its domain, there is a logical variable $x_{i,v}$ that is true iff variable i takes value v. The reason for such a domain representation is that it lets the SAT solver achieve *pruning* in a similar way to a CSP solver applied to the original CSP. When the SAT solver infers that a logical variable $x_{i,v}$ is false, it means that the corresponding CSP variable i cannot take value v: the value v has been pruned. The most popular CSP-SAT encoding was called *direct encoding* by Walsh [24]; the DPLL applied to the SAT encoded CSP mimics the Forward Checking on the original CSP [10, 24]. Gent [11] proposes the *support encoding*, that has the same representation of domains, but a different representation of constraints. He proves that unit propagation (used in DPLL solvers) applied to the support encoding of a CSP to SAT, achieves the same pruning of arc-consistency on the original CSP. Stronger types of consistency are proven in [6].

On the other hand, using a SAT variable for each element in the domains generates a huge search space. Indeed, it lets the SAT solver perform powerful arc-consistency propagation, but at a cost: the search space of SAT is exponential in the number of logical variables. Let us draw again a parallel with constraint solvers: many efficient constraint solvers [1, 20, 14] use a compact representation of domains. A domain is often represented with its *bounds* $\{l..u\}$ so there is no need to enumerate all the values in the domain. For example, the domain $\{1..100\}$ is represented just with the two integer numbers 1 and 100. When intermediate values are deleted, the domain is represented often as unions of intervals, and only when there is no other possibility they switch to an enumeration of all the elements. Constraint solvers give up even the arc-consistency propagation in some cases, in favour of a cheaper bound-consistency. For example, most solvers will avoid arc-consistency propagation of the constraint $X = 2 * Y$, that would require deleting all the odd values from the domain of X, forcing to switch from a compact representation to the enumerative form of the domain.

In logarithmic encodings, each domain is represented by $\lceil \log_2 d \rceil$ SAT variables [15, 13, 8, 24, 9, 18]. Such encodings can be tailored for specific constraints, such as the *not equal* constraint [9]. In general, however, such encodings lack the ability to remove single values from domains, which yields less powerful propagation when compared to CSP solvers. In log encodings, one can either remove half of the elements in the domain of a variable, or none. This can be useful as a branching heuristics, but only for assigning the most significant bit[1]: fixing the least significant bit means removing either all the even elements, or all the odd (which is seldom useful). However, such encodings promise to save memory, and to reduce the search space when compared to linear encodings: the search space is exponential in the number of SAT variables, so reducing their number could boost a significant improvement. Concerning constraints, log encodings usually adopt a representation similar to the direct encoding.

In this paper, we propose a new encoding, called log-*support*, that uses support clauses in a logarithmic encoding. The codification of domains can be either the usual binary representation, or based on a Gray code, in order to maximise

[1] If a number n is represented with the m bits $\langle b_{m-1}, \dots, b_0 \rangle$, i.e., $n = \sum_{i=0}^{m-1} b_i 2^i$, we call b_{m-1} the *most significant bit* and b_0 the *least significant bit.*

the propagation power of support clauses. We apply the new encodings on randomly generated problems and on benchmark job-shop scheduling problems, and compare the performances of the SAT solvers Chaff [17] and MiniSat [7] on the encoded problems.

2 Preliminaries and Notation

A *Constraint Satisfaction Problem* (CSP) is a triple $\langle X, D, C\rangle$ where X is a set of variables, ranging on some domains D, and subject to a set of constraints C. To simplify the presentation, in this paper we focus on *binary CSPs*, i.e., problems in which all the constraints involve at most two variables. We will indicate with $n = |X|$ the number of variables, with d the maximum cardinality of the domains. The symbols i and j will usually refer to variables, while v and w are values in a domain.

A *Satisfiability problem* (SAT) is also built on a set of variables, which can take only values true and false. In the paper, we will call them *logical variables* or *SAT variables* to avoid confusion with the CSP variables. We will often indicate the values true and false with the numbers 1 and 0. A SAT problem contains a logical formula built on the logical variables. The formula is typically required to be in conjunctive normal form, i.e., a set of *clauses*, i.e., disjunctions of literals of the logical variables. A solution to a SAT problem is an assignment of values true, false to the logical variables, such that all clauses are true (i.e., at least one of the literals in each clause is true).

3 A Survey on Encodings

3.1 Direct encoding

In the *direct encoding* [24] there is a logical variable $x_{i,v}$ for each CSP variable i and domain value v. For each CSP variable i, a clause (referred to as *at-least-one* clause) imposes that i takes at least one of the values in its domain:

$$x_{i,1} \vee x_{i,2} \vee \ldots \vee x_{i,d} \tag{1}$$

Symmetrically, a set of clauses (called *at-most-one* clauses) forbid the variable i to take two values at the same time:

$$\begin{array}{cccc} \neg x_{i,1} \vee \neg x_{i,2} & \neg x_{i,1} \vee \neg x_{i,3} & \ldots & \neg x_{i,1} \vee \neg x_{i,d} \\ & \neg x_{i,2} \vee \neg x_{i,3} & \ldots & \neg x_{i,2} \vee \neg x_{i,d} \\ & & & \vdots \\ & & & \neg x_{i,d-1} \vee \neg x_{i,d} \end{array} \tag{2}$$

Finally, a set of clauses encodes the *constraints*. For each pair of inconsistent assignments $i \mapsto v$, $j \mapsto w$ s.t. $(v, w) \notin c_{i,j}$, we have a *conflict clause*:

$$\neg x_{i,v} \vee \neg x_{j,w} \tag{3}$$

As an example, consider the following CSP: $A \leq B$, with A and B ranging over the set of values $\{0, 1, 2\}$. The direct encoding produces the clauses:

at-least-one		$a_0 \vee a_1 \vee a_2$		$b_0 \vee b_1 \vee b_2$
at-most-one	$\neg a_0 \vee \neg a_1$	$\neg a_0 \vee \neg a_2$	$\neg b_0 \vee \neg b_1$	$\neg b_0 \vee \neg b_2$
		$\neg a_1 \vee \neg a_2$		$\neg b_1 \vee \neg b_2$
conflict	$\neg a_1 \vee \neg b_0$	$\neg a_2 \vee \neg b_0$	$\neg a_2 \vee \neg b_1$	

In this encoding, at-most-one clauses can be removed: in case the solution provided by the SAT solver contains more than one CSP value for a CSP variable, any of the values can be selected [18].

3.2 Support Encoding

In the Support Encoding [16, 11], domains are represented in the same way as in the direct encoding, i.e., we have *at-least-one* and *at-most-one* clauses. Constraints, instead, are based on the notion of support. If an assignment $i \mapsto v$ supports the assignments $j \mapsto w_1$, $j \mapsto w_2$, ..., $j \mapsto w_k$, we impose that

$$x_{i,v} \rightarrow x_{j,w_1} \vee x_{j,w_2} \vee \ldots \vee x_{j,w_k}$$

i.e., we impose a *support clause*:

$$\neg x_{i,v} \vee x_{j,w_1} \vee x_{j,w_2} \vee \ldots \vee x_{j,w_k} \tag{4}$$

In particular, if an assignment $i \mapsto v$ does not support any value, the support clause reduces to $\neg x_{i,v}$. If an assignment $i \mapsto v$ supports all values for a variable j, no clause is required, so the CSP in the running example is represented as:

at-least-one		$a_0 \vee a_1 \vee a_2$		$b_0 \vee b_1 \vee b_2$
at-most-one	$\neg a_0 \vee \neg a_1$	$\neg a_0 \vee \neg a_2$	$\neg b_0 \vee \neg b_1$	$\neg b_0 \vee \neg b_2$
		$\neg a_1 \vee \neg a_2$		$\neg b_1 \vee \neg b_2$
support		$\neg a_1 \vee b_1 \vee b_2$		$\neg b_0 \vee a_0$
		$\neg a_2 \vee b_2$		$\neg b_1 \vee a_0 \vee a_1$

3.3 Log Encoding

In the log encoding [15, 24, 9], domains are represented with $m = \lceil \log_2 d \rceil$ logical variables: each of the 2^m combinations represents a possible assignment. More precisely, for each CSP variable i we have logical variables x_i^b, where $x_i^b = 1$ iff bit b of the value assigned to i is 1. In this encoding, *at-least-one* and *at-most-one* clauses are not necessary; however, in case the cardinality of domains is not a power of two, we need to exclude the values in excess, with the so-called *prohibited-value clauses* [18] (although the number of these clauses can be reduced [9]). If value v does not belong to the domain of i, and v is represented with the binary digits $\langle v_{m-1}, \ldots, v_0 \rangle$ (i.e., $v = \sum_{b=0}^{m-1} 2^b v_b$), we can impose that

$$\neg \left(\bigwedge_{b=0}^{m-1} \neg (v_b \oplus x_i^b) \right) \tag{5}$$

where the symbol $\oplus$ stands for exclusive or. Intuitively, $\neg(s \oplus b)$ is the literal b if s is true, and the literal $\neg b$ if s is false. Equation (5) is converted into the the prohibited-value clause

$$\bigvee_{b=0}^{m-1} v_b \oplus x_i^b.$$

In the running example, the domains have 3 values, so two bits are needed. However, with 2 bits we have 2^2 combinations, so we have a spare combination: the value 3. We add a prohibited-value clause for each domain:

$$\neg a_1 \vee \neg a_0 \qquad \neg b_1 \vee \neg b_0.$$

Each constraint can have a specific encoding, tailored for its propagation (see [9] for the not-equal constraint). In general, however, they are encoded with conflict clauses. If two assignments $i \mapsto v$, $j \mapsto w$ are in conflict, we impose a clause of length $2m$:

$$\left(\bigvee_{b=0}^{m-1} v_b \oplus x_i^b\right) \vee \left(\bigvee_{b=0}^{m-1} w_b \oplus x_j^b\right)$$

where v_b and w_b are the binary representations of the values v and w.

In the running example, we will have:

prohibited-value	$\neg a_1 \vee \neg a_0$	$\neg b_1 \vee \neg b_0$
conflict	$a_1 \vee \neg a_0 \vee b_1 \vee b_0$	$\neg a_1 \vee a_0 \vee b_1 \vee b_0$
	$\neg a_1 \vee a_0 \vee b_1 \vee \neg b_0$	

4 The log-support encoding

In the log-encoding, we have a number of conflict clauses, each consisting of $2m$ literals; unluckily, the length of clauses typically influences negatively the performance of a SAT solver.

Gent [11] proved that the DPLL procedure applied to a support-encoded CSP performs powerful propagation of constraints, equivalent to the arc-consistency in the original CSP. One could think of applying support clauses to logarithmic encodings; the most intuitive formulation is probably the following. If an assignment $i \mapsto v$ supports the assignments $j \mapsto w_1$, $j \mapsto w_2$, ..., $j \mapsto w_k$, we could impose, as in the support encoding, that

$$v \rightarrow w_1 \vee w_2 \vee \ldots \vee w_k$$

and then encode in binary form the values v and w_i. However, the binary form of a value w_i is a conjunction of literals, so the formula becomes

$$\left(\bigwedge_b \neg(v^b \oplus x_i^b)\right) \rightarrow \left(\bigwedge_b \neg(w_1^b \oplus x_j^b)\right) \vee \left(\bigwedge_b \neg(w_2^b \oplus x_j^b)\right) \vee \ldots \vee \left(\bigwedge_b \neg(w_k^b \oplus x_j^b)\right) \tag{6}$$

which, translated in conjunctive normal form, generates an exponential number of clauses[2].

We convert in clausal form only implications that have exactly one literal in the conclusion. Let us consider, as a first case, only the most significant bit. In our running example, the assignment $A \mapsto 2$ supports only the assignment $B \mapsto 2$. We can state that, whenever A takes value 2, the most significant bit of B must be true, i.e., we can impose:

$$a_1 \wedge \neg a_0 \rightarrow b_1$$

that results in the support clause $\neg a_1 \vee a_0 \vee b_1$. This clause is enough to rule out two conflicting assignments: $(A \mapsto 2, B \mapsto 0)$ and $(A \mapsto 2, B \mapsto 1)$. So, we can remove from the log-encoded CSP two conflict clauses (consisting of 4 literals each) by adding one support clause (consisting of 3 literals). In our example:

prohibited-values	$\neg a_1 \vee \neg a_0$	$\neg b_1 \vee \neg b_0$
support	$\neg a_1 \vee a_0 \vee b_1$	
conflict	$a_1 \vee \neg a_0 \vee b_1 \vee b_0$	

Note that this transformation is not always possible: we can substitute some of the conflict clauses with one support clause only if all the binary form of supported values agrees in the most significant bit. In other words,

– if a value v in the domain of variable i supports only values greater than or equal to 2^{m-1} in the domain of variable j, we impose that

$$\left(\bigvee_{b=0}^{m-1} \neg(v_b \oplus x_i^b) \right) \rightarrow x_j^{m-1}$$

and avoid imposing the conflict clauses involving v and the values w such that $w < 2^{m-1}$

– if a value v in the domain of variable i supports only values less than 2^{m-1} in the domain of variable j, we impose that

$$\left(\bigvee_{b=0}^{m-1} \neg(v_b \oplus x_i^b) \right) \rightarrow \neg x_j^{m-1}$$

and avoid imposing the conflict clauses involving v and the values w such that $w \geq 2^{m-1}$

– Otherwise, if some of the elements supported by v are in the interval $[0..2^{m-1}-1]$ and some are in the interval $[2^{m-1}..2^m-1]$, we do not perform the transformation, and impose the corresponding conflict clauses, as in the log-encoding.

[2] Actually one could avoid the exponential number of clauses by introducing new variables; we leave this issue for future research.

In general, a support clause has length $m+1$ and lets us remove $d/2$ conflict clauses (of length $2m$). This offers a significant reduction of the number of conflict clauses when the domains are big, and the assignments that satisfy the constraint are all grouped either in the first half or in the second half of the domain. This happens rather often in many significant constraints (for instance, $>$, $\geq$, $<$, $=$).

To sum-up, this encoding has the following features:

– same number ($n\lceil\log_2 d\rceil$) of logical variables required by the log-encoding;
– reduced number of conflict clauses (of length $2\lceil\log_2 d\rceil$), substituted by support clauses (of length $\lceil\log_2 d\rceil + 1$).

Extending to the other bits Of course, we can apply the same scheme to the other direction (from B to A), and to other bits (not just to the most significant one). In the running example, we can add the following support clauses:

$$b_1 \vee b_0 \vee \neg a_0 \qquad b_1 \vee \neg b_0 \vee a_1 \qquad b_1 \vee \neg b_0 \vee \neg a_1$$

and, in this case, remove all the conflict clauses.

```
for each $v \in dom(i)$
  MaskAnd = $2^m - 1$; // all bits = 1
  MaskOr = 0;
  for each $w \in dom(j)$
    if supports($i \mapsto v$,$j \mapsto w$)
      then MaskAnd = MaskAnd$\wedge w$ //Applies AND to each bit
           MaskOr = MaskOr$\vee w$ //Applies OR to each bit
    endif;
  end foreach $w$;
  for $b = 0$ to $m$
    if ($b$-th bit of MaskAnd) = ($b$-th bit of MaskOr)
      insert support clause;
    endif;
  end for;
end foreach $v$
```

Fig. 1. Algorithm for finding support clauses

At first sight, finding if the support values share a same bit seems to require scanning the set of supports for each bit, i.e., for each pair variable/value, one needs to check m times the supporting values. However, such operation can be performed with simple bit manipulations, and there is no need to scan for the supports more than once, as shown in Figure 1.

Reducing further the conflict clauses Suppose that a value v in the domain of variable i conflicts with two consecutive values w and $w+1$ in the domain of j.

Suppose that the binary representation of the numbers w and $w+1$ differs only for the least significant bit b_0. In this case, we can represent both the values w and $w+1$ using only the $m-1$ most significant bits, so we can impose one single conflict clause of length $2m-1$. This simple optimization can be considered as applying binary resolution [6] to the two conflict clauses.

This scheme can then be extended to sets of consecutive conflicting values whose cardinality is a power of two.

4.1 The quest for bound-consistency

Support clauses can rule out many conflict clauses. Moreover, if we focus on the most significant bit, we can see that a support clause can split the domain in two halves, dividing the elements higher than $2^{m-1}-1$ from those smaller than 2^{m-1}. In a sense, this type of pruning goes in the direction of bound consistency. In bound consistency, the representation of the domain is simply given by the two bounds, which are numbers represented with the usual binary representation of integers (i.e., with a logarithmic number of bits). The same space occupation (logarithmic number of bits) holds for the log and log-support encodings.

In bound consistency, the logarithmic space representation of domains allows us to represent domains as intervals: all intervals are representable, both during bound-consistency propagation and during search. In the SAT-encoded CSP, the current domain of a variable can be thought as the union of the possible configurations allowed by the logical variables that are still not assigned. For instance, a CSP variable A with domain $[0..3]$ is represented in log-encoding with 2 variables a_1, a_0. If (for instance, during the DPLL search) a_1 is bound to `true` (i.e., value 1), the possible values that $\langle a_1, a_0\rangle$ can take are 10 and 11, and we can interpret the domain of A as being currently $\{2, 3\}$.

Again, if during the DPLL search the logical variables corresponding to the CSP variable A are in the following states[3]

$$a_4 \mapsto 1 \qquad a_3 \mapsto 0 \qquad a_2 \mapsto U \qquad a_1 \mapsto U \qquad a_0 \mapsto U$$

the possible values that A can take are 16..23. This situation resembles bound consistency: all the available values are in an interval. However, in the situation

$$a_4 \mapsto U \qquad a_3 \mapsto 0 \qquad a_2 \mapsto U \qquad a_1 \mapsto U \qquad a_0 \mapsto 1 \tag{7}$$

the available values for A are $\{1, 3, 5, 7, 17, 19, 21, 23\}$, and they do not represent an interval. Since CP solvers are very fast and rely often on bound consistency, we might try to obtain the same propagation; with this aim in mind, the situation of Eq. 7 can be considered a "wasted" configuration. By using the classical binary representation of numbers, one obtains an interval only if the first k (most significant) logical variables are all ground, and all the remaining SAT variables of the domain are unassigned. Thus, the number of representable intervals is

[3] Where 'U' stands for an unassigned logical variable, i.e., a variable that still has not an associated value.

$\sum_{k=0}^{m} 2^k = 2^{m+1} - 1$. When the size of a domain d is a power of 2, $m = \log_2 d$, so the number of possible intervals is $2d - 1$.

However, we might think of rearranging the domain values in the hope that all configurations represent intervals. One may wonder if there exists a logarithmic encoding such that all the possible intervals can be represented with the assignments of (some of) the logical variables. Since we have d values, we have $\lceil \log_2 d \rceil$ logical variables. Since each SAT variable can be in one of three possible states (value 1, value 0, unassigned), the possible domains of a CSP variable during search are $3^{\lceil \log_2 d \rceil}$. The possible sub-intervals are $d(d+1)/2$, which is greater than $3^{\lceil \log_2 d \rceil}$ for $d > 2$, so rearranging the values is not enough to get bound consistency: more than $\lceil \log_2 d \rceil$ logical variables are needed (per domain).

One may ask whether there exists an encoding with a number of logical variables per domain greater than $\lceil \log_2 d \rceil$ (as in the log and log-support encodings) but significantly smaller than d (as in the direct and support encodings) in which unit propagation is equivalent to (or stronger than) bound consistency on the original CSP. In a CP solver, domains only shrink during propagation. The SAT solver shrinks the domain by fixing the value of a logical variable, so we require that all element removals are obtained by fixing the value of a logical variable.

Theorem 1. *Let $D = [l..u] \subset \mathbb{Z}$ an integer interval, $k = |D|$. Let $\{0, 1, U\}^m$ (for some integer m) a set. Let $a \preceq b \Leftrightarrow [\forall i (a_i = b_i \vee a_i = U)]$ a partial order on $\{0, 1, U\}^m$. Let $\mathcal{I} \subseteq \wp(D)$ the set of non empty intervals included in D. Let $f : \mathcal{I} \mapsto \{0, 1, U\}^m$ an injective function such that $X \subseteq Y \Rightarrow f(X) \preceq f(Y)$.*

Then, $m \geq k - 1$.

Proof. Consider the sequence of intervals $I_0 \equiv D$, $I_1 \equiv [l+1..u]$, $\ldots I_{k-1} \equiv \{l\}$. Each interval is strictly included in the previous, so $f(I_j) \prec f(I_{j+1})$. But $a \prec b$ implies that the number of U symbols in a is greater than the number of U symbols in b. Thus, I_0 has at least $k - 1$ symbols U, and $m \geq k - 1$.

Theorem 1 gives a lower bound on the number of variables required for achieving a propagation strong at least as bound-consistency. $k - 1$ is also an upper bound: with $k - 1$ variables we can achieve arc-consistency, by using a support encoding of k variables in which the k-th variable is not explicitly represented (as it is true when all the other $k - 1$ are false). So, a linear number of logical variables is needed to achieve bound-consistency. But this is the same number of variables needed for the stronger arc-consistency, so bound-consistency cannot reduce the number of SAT variables required (w.r.t. arc-consistency).

However, we can try an approximation of bound consistency: we can rearrange the values in the domains such that a higher number of configurations reachable during the DPLL search represent intervals. The Gray code is a suitable candidate.

4.2 Gray code

The *Reflected Binary Code* was introduced by Gray [12] (and then named after him). In the Gray code, with m bits one can represent 2^m different values, as in the classical binary code. However, the encoded version of any two consecutive numbers differs only for one bit (Figure 2). The Gray encoding of a binary number b can be simply obtained as $g = b \oplus (b/2)$, i.e., the exclusive OR of the number b and its right shift of one position.

0	0000	4	0110	8	1100	12	1010
1	0001	5	0111	9	1101	13	1011
2	0011	6	0101	10	1111	14	1001
3	0010	7	0100	11	1110	15	1000

Fig. 2. 4 bit Gray code

So, by encoding the values in the CSP domains with a Gray code, all intervals of size 2 are representable, while in the classical binary code only half of them are representable: those in which the lower bound is even and the upper bound is odd. For instance, in a 4-bit binary code we can represent the domain $\{2, 3\}$ (configuration 001U) but we cannot represent the domain $\{3, 4\}$. In the 4-bit Gray code, $\{2, 3\}$ is represented by configuration 001U and $\{3, 4\}$ by 0U10. There are $d-1$ intervals of size 2. Since, as in the classical binary representation, when the first k bits are fixed we also have an interval of possible values, the number of intervals representable with the Gray code is at least $\frac{5}{2}d$ (since of the intervals of size 2, half are also possible with the classical binary representation).

With a Gray representation, the running example is encoded as follows:

prohibited-values	$\neg a_1 \vee a_0$	$\neg b_1 \vee b_0$
support	$\neg a_1 \vee a_0 \vee b_0$	$b_1 \vee b_0 \vee \neg a_0$
	$\neg a_1 \vee \neg a_0 \vee b_0$	$b_1 \vee b_0 \vee \neg a_1$
	$\neg a_1 \vee \neg a_0 \vee b_1$	$b_1 \vee \neg b_0 \vee \neg a_1$

By using a Gray code, the number of support clauses has increased from 4 to 6 (50%), while (in this case) no conflict clauses are necessary. The intuition is that a higher number of support clauses should allow for more powerful propagation, but in some cases it could also increase the size of the SAT problem. However, each support clause has one CSP value in the antecedent and one of the bits in the conclusion, so for each CSP constraint there are at most $2d\lceil \log_2 d \rceil$ support clauses. The number of conflict clauses in the log-encoding cannot be higher than the number of pairs of elements in a domain, so d^2. Recall also that conflict clauses are longer than support clauses, so we can estimate the size of the (Gray) log-support encoding to be smaller than that of the log-encoding, when d is sufficiently large.

Finally, it is worth noting that various codes can exist having the same property of the Gray code; however while the reflected binary Gray code, as presented here, can be computed with two simple binary operations (namely shift and exclusive or), in general finding other codes with the same property is an NP-complete problem (it is equivalent to finding an Hamiltonian circuit on a hypercube). In future work we plan to study codes that maximise the number of intervals that can be represented.

5 Experimental Results

In order to test the effectiveness of the log-support encoding and its Gray variant (in the following, the Gray-encoding), we developed a series of experiments, based on randomly-generated CSPs and on Job-Shop Scheduling benchmarks.

5.1 Randomly generated problems

The first set of experiments is based on randomly generated CSPs. A random CSP is often generated given four parameters [22]: the number n of variables, the size d of the domains, the probability p that there is a constraint on a given pair of variables, and the conditional probability q that a pair of assignments is consistent, given that there is a constraint linking the two variables.

In order to exploit the compact representation of log-encodings, we focussed on CSPs with a high number of domain values. In order to keep the running time within reasonable bounds, we had to keep small the number of CSP variables.

The log-support encoding was developed for constraints in which the set of satisfying assignments is connected, and we can easily foresee that a Gray code will have no impact on randomly generated constraint matrices. Thus, to test the applicability of the Gray encoding, we used a different generation scheme, in which satisfying assignments have a high probability to be grouped in clusters. Note that also real-life constraints typically have their satisfying assignments grouped together, and not completely sparse.

For each constraint (selected with independent probability p) on two variables A and B, we randomly selected a pair of values v and w respectively from the domains of A and B. The pair (v, w) works as an "attractor": the probability that a constraint is satisfied will be higher near the point (v, w) and will be smaller far from that point. More precisely, the probability that a pair of assignments (a, b) is satisfied is proportional to the euclidean distance between (a, b) and the attractor (v, w): $q = 1 - \alpha\sqrt{(a-v)^2 + (b-w)^2}$, where α is a coefficient that normalises the value of q in the interval 0..1. A posteriori, we grouped the experiments with a same frequency of satisfied assignments, and plotted them in the graph of Figure 3.

This set of experiments was performed running zChaff 2004.5.13 [17] and with MiniSAT 1.14 [7] on a Pentium M715 processor 1.5GHz, with 512MB RAM. A memory limit was set at 150MB, and a timeout at 1000s. Each of the point is the geometric mean of at least 25 experiments, where the conditions of timeout or out of memory are represented by 1000s. The timing results include both the time spent for the encoding and for solving the problem; however, the encoding time was always negligible. Note that to perform the experiments we did not generate a DIMACS file, because the time for loading the DIMACS could have been large (see also the discussion in the nex section).

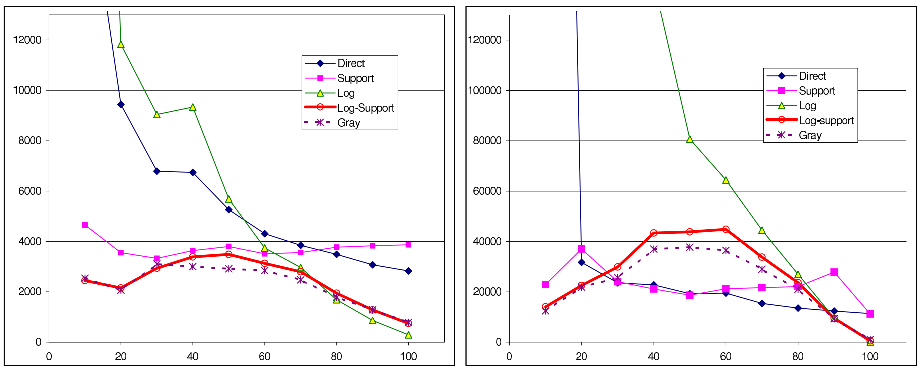

Fig. 3. Experiments on randomly-generated problems ($n = 7$, $d = 512$, $p = 30$ and q from 10 to 100). Times in ms. Left: Chaff, Right: MiniSat

From the graphs, we can see that the log encoding is the slowest when the constraints are very tight (small values of q). This is probably due to the fact that in the log-encoding we have limited propagation of constraints, which makes hard proving unsatisfiability. On the other hand, when the constraints are very loose (q near 80-90%), the log encoding performs better than the direct and support encodings.

The support encoding is often the best option for MiniSat, while Gray performed best in the zChaff experiments. Moreover, the log-support/Gray encodings are often competitive. For high values of q, the log-support/Gray encodings keep the same behaviour of the log-encoding. This is reasonable, because when q is near 100% very few support clauses are inserted. On the other hand, when the probability q is small, the support clauses have a strong influence, and allow the SAT solver to detect infeasibility orders of magnitude faster than in the log-encoding. Both the log-support and the Gray encodings are typically faster than the direct encoding.

Finally, the Gray encoding is slightly faster than the log-support, probably due to the fact that more support clauses are present.

5.2 Job-Shop Scheduling Problems

We applied the encodings to the set of Job-Shop Scheduling Problems taken from the CSP competition 2006 [23] (and originally studied in [19]). These problems involve 50 CSP variables with variable domain sizes, from 114 to 159.

The results are given in Figure 4: the plots show the number of problems that were solvable within a time limit given in abscissa (the higher the graph, the better). The experiments were performed with zChaff 2004.5.13 [17] and with MiniSat 1.14 [7]. The results were obtained on a Pentium M715 processor 1.5GHz, with 512MB RAM.

The log encoding performed worst, and both solvers were able to solve only a limited subset of the problems within 500s.

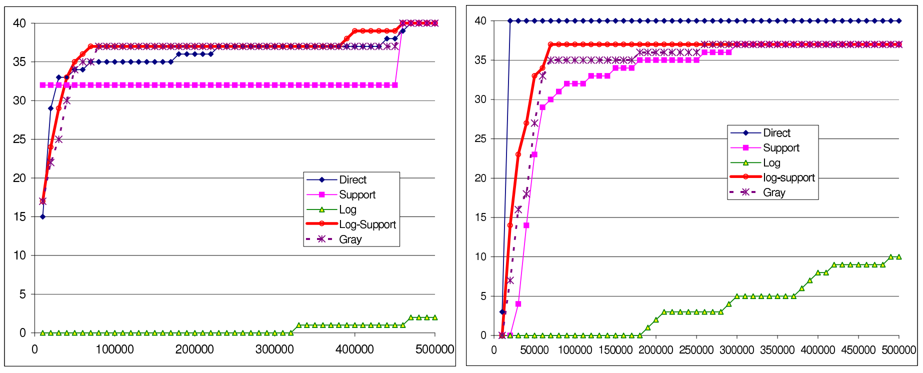

Fig. 4. Experiments on Job-Shop scheduling problems. Left: Chaff, right: MiniSAT

In the experiments performed with Chaff, the support encoding was able to solve some of the problems very quickly; however, given a longer timeout, the log-support was typically the best choice (it was able to solve more problems). In the experiments with MiniSat, the best encoding was the direct, possibly because of the special handling of binary clauses implemented in MiniSat. Notice that for both solvers the support encoding performed worse than the log-support and the Gray encoding.

In this set of instances, the Gray encoding did not provide any improvement with respect to the log-support.

Chaff required on average 65MB of RAM to solve a direct-encoded CSP, 56MB to solve a support-encoded CSP, and only 19MB to solve a problem encoded with log-support or Gray.

The size of the generated SAT problem is also instructive and can give an estimate of the size of a DIMACS file containing the encoding. A log-encoded CSP used on average about 10^7 literals; considering that the number of logical variables is about $n\lceil\log_2 d\rceil \approx 50 \cdot 8 = 400$, and that each literal is represented in DIMACS with 5.5 bytes (3 ASCII characters for representing the number itself, plus one of space, and half, on average, for the sign), we can estimate a DIMACS file of about 55MB. The log-support/Gray encodings generate on average about $1.7 \cdot 10^6$ literals, thus the average size of a DIMACS can be estimated as 9.5MB.

On the other hand, the linear encodings use nd logical variables, that in our instances is about 6600, so four bytes are necessary in the text file DIMACS format to represent the variable, that gives on average 6.5 bytes per literal. The direct encoding used on average $2.3 \cdot 10^6$ literals, that gives 14MB for the DIMACS, and the support $7 \cdot 10^6$, that gives a 45MB DIMACS file.

We can conclude that the log-support is a significant improvement with respect to the log encoding, both in terms of solution time and size of the generated SAT problem. The direct encoding is often faster than the log-support, but it requires more memory for Chaff to solve them, and the DIMACS file is much larger. Thus the log-support and Gray encodings could be interesting solution methods in cases with limited memory.

6 Conclusions and future work

In this paper we proposed a new encoding, called log-support, and its variant in Gray code, for mapping CSPs into SAT. The log-support uses a logarithmic number of SAT variables for representing the domains of CSP variables, as in the well-known log-encoding. Experiments show that the new encodings outperform the traditional log-encoding, and are often competitive with direct and support encodings. Moreover, the size of the encoded SAT instance is typically a fraction of the size required by the direct and support encodings.

In future work, we plan to define a platform for defining CSPs, in the line of previous research. Cadoli et al. [4, 3] developed a language and an architecture for encoding problems belonging to the class NP into DIMACS form, and then use a SAT solver to perform the search. We believe that such architecture could be enriched by populating it with a variety of the many encodings proposed in recent years, and with the log-support/Gray encodings.

Other optimisations could be performed on the log encodings. We cite the *binary encoding* [8], that uses a logarithmic number of logical variables to encode domains, and it avoids imposing *prohibited value* clauses by encoding a domain value with a variable number of SAT variables. In future work, we plan to experiment with a variation of the log-support that exploits the same idea.

Finally, we plan to study the applicability of the proposed encodings to non-binary CSPs.

Acknowledgements The author would like to thank the anonymous reviewers for helping improving the presentation of the paper, and for the many interesting suggestions for future work.

This work has been partially supported by the MIUR PRIN 2005 project *Vincoli e preferenze come formalismo unificante per l'analisi di sistemi informatici e la soluzione di problemi reali.*

References

1. *ECL^iPS^e User Manual, Release 5.2.* London, UK, 2001.
2. D. Le Berre and L. Simon. SAT competition, 2005. www.satcompetition.org/2005/.
3. M. Cadoli, T. Mancini, and F. Patrizi. SAT as an effective solving technology for constraint problems. In *ISMIS 2006*, volume 4203 of *LNCS*, pages 540–549.
4. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artificial Intelligence*, 162(1-2):89–120, 2005.
5. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
6. Y. Dimopoulos and K. Stergiou. Propagation in CSP and SAT. In F. Behamou, editor, *CP 2006*, number 4204 in LNCS, pages 137–151. Springer-Verlag, 2006.

7. Niklas Een and Niklas Sörensson. MiniSat - a SAT solver with conflict-clause minimization. In *SAT 2005*, 2005.
8. A. Frisch and T. Peugniez. Solving non-boolean satisfiability problems with stochastic local search. In B. Nebel, editor, *IJCAI 2001*, pages 282–290, 2001.
9. A. Van Gelder. Another look at graph coloring via propositional satisfiability. In *Computational Symposium on Graph Coloring and its Generalizations*, 2002.
10. R. Genisson and P. Jegou. Davis and Putnam were already forward checking. In *Proc. of the 12th ECAI*, pages 180–184. Wiley, 1996.
11. I.P. Gent. Arc consistency in SAT. In F. van Harmelen, editor, *ECAI'2002*, pages 121–125. IOS Press, July 2002.
12. F. Gray. Pulse code communication, March 1953. U. S. Patent 2 632 058.
13. H. Hoos. SAT-encodings, search space structure, and local search performance. In T. Dean, editor, *IJCAI 99*, pages 296–303. Morgan Kaufmann, 1999.
14. ILOG S.A., France. *ILOG Solver*, 5.0 edition, 2003.
15. K. Iwama and S. Miyazaki. SAT-variable complexity of hard combinatorial problems. In *IFIP World Computer Congress*, pages 253–258. North-Holland, 1994.
16. S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45:275286, 1990.
17. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC 2001*, pages 530–535. ACM.
18. S. Prestwich. Local search on SAT-encoded colouring problems. In E. Giunchiglia and A. Tacchella, editors, *SAT 2003*, volume 2919 of *LNCS*, pages 105–119.
19. N.M. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86(1):1–41, 1996.
20. *SICStus Prolog user manual, Release 3.11.0*, 2003. www.sics.se/isl/sicstus/.
21. C. Sinz. The SAT race, 2006. http://fmv.jku.at/sat-race-2006/.
22. Barbara M. Smith and Martin E. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artif. Intell.*, 81(1-2):155–181, 1996.
23. M. van Dongen, C. Lecoutre, and O. Roussel. Second international competition of CSP and Max-CSP solvers, 2006. http://www.cril.univ-artois.fr/CPAI06/.
24. T. Walsh. SAT v CSP. In R. Dechter, editor, *CP 2000*, volume 1894 of *LNCS*, pages 441–456. Springer-Verlag, 2000.