CSP and SAT Encoding

Mazen Bouchur
TU Clausthal, Germany

Abstract

This paper gives a brief overview over different encodings between the constraint satisfaction problem (CSP) and the Boolean satisfiability problem (SAT). Each encoding is analyzed and evaluated based on various aspects and characteristics such as performance and complexity. Most importantly each encoding is being compared performance-wise to the unencoded counterpart with respect to the standard algorithm of each problem (DPLL in case of SAT and arc consistency based algorithm in case of CSP). We try to summarize multiple papers discussing several encodings and highlight the remarkable properties of each them, specially the SAT-based constraint solver "sugar" and its underlying order encoding.

1 Introduction

1.1 SAT

Ever since it was formulated and the Boolean satisfaction problem (SAT) is gaining an increasing interest in the field of computer science. It has a wide range of applications in various areas, ranging from hardware design to artificial intelligence (AI) and verification. SAT can be used to express many AI tasks and mathematical problems as a constraint satisfaction problems. Other theoretical applications also include automated theorem proving and model checking.

SAT can be defined as the problem of finding an assignment of the propositional variables of given Boolean formula, such that the whole formula evaluates to True (satisfied). SAT is one of the first problems that was proven to be NP-complete, which makes it gain increased importance. Being simple in its nature and construction has allowed intensive studying and optimization. Therefore, the efforts to converting problems into SAT instances to benefit from its advantages should come as no surprise. Throughout this paper it is assumed that the SAT problem is expressed in clausal CNF form to facilitate a common basis.

1.2 CSP

Many famous problems such as the n-queens' problem, map coloring and Sudoku can be easily expressed as an instance of the constraint satisfaction problem (CSP). Each CSP consists of three sets:

1) Set of variables.
2) Set of domains for each variable (usually a finite domain is chosen for all the variables).
3) Set of relations that define the constraints over a subset of the variables' set.

1.2.1 CSP Types

Based on the number of the participated variables, a constraint can be classified as unary, binary or n-ary constraint. The entire CSP can be described as binary or non-binary depending on the constraints' classification. To be consistent and for facilitate theoretical analysis only unary and binary CSP are considered in this paper.Any n-ary can be expanded and expressed in terms of binary constraints.

2 Preliminaries

In this section, the algorithms used for solving each problem and the basis criteria for the theoretical comparison between them will be formally introduced.

2.1 The Davis-Putnam-Logemann-Loveland algorithm (DPLL)

One technique to solve the satisfiability problem is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, which is a refinement of the original resolution-based Davis-Putnam (DP) algorithm. DPLL is based on four rules:

1) Tautology elimination
2) One Literal (Unit clause)
3) Pure Literal
4) Branching or splitting

For our purposes, just the one literal and the branching rules will be considered as they are sufficient for the completeness and soundness of the algorithm, and none of the other rules affect the efficiency.

One Literal: This rule eliminates unit clauses (any clause contains a single unassigned literal) by assigning the only possible value to make this literal true. Thus, no choice is necessary. By applying this rule, a large part of the entire search space can be pruned, leading to a more efficient SAT instance that is equivalent to the original problem.

Branching: When no other rule is applicable, branching is used to assign a truth value to a variable. This assignment is non-deterministic and depending on the used branching-heuristics a variable is chosen.

An unsatisfiable SAT problem will lead to apply branching and unit propagation to all branches until no variable is left. For this reason, it is usually harder to detect unsatisfiability, since all branches must be traversed.

2.2 Arc consistency and the forward checking algorithm (FC)

A standard approach to solve the constraint satisfaction problems is maintaining arc consistency (MAC) or forward checking (FC). Both algorithms are built around the idea of enforcing some level of arc consistency (AC) at each step and branch when needed. The main difference between FC and MAC is the fact that MAC tries to maintain AC on each node at every level while FC tries to avoid the extra work by maintaining AC only between the most recently instantiated variables and those that are still not instantiated.

While FC do in general much less work at each node, it needs to branch much more to exclude all wrong paths. On the other hand, MAC usually have smaller number of branches and spends in return more time at each node. This trade-off leads to different characteristics of each algorithm, which in turn prioritize one over another depending on the problem at hand.

2.3 Analysis approach

To determinate that an encoding is good enough or rather better than another (see below 2.3.1), we will compare solving a given problem with its standard algorithm to solving the encoded version of the same problem with its respective standard algorithm. For example: A CSP problem can be solved directly using MAC or it can be encoded as SAT problem (using order encoding for instance) and solved using DPLL. Note that the encoding process cannot take longer time than solving the problem itself. In concrete terms, the encoding procedure (which can be considered as a type of reduction) can be done in polynomial time since both problems are NP complete.

2.3.1 Branching for comparing

Since both MAC/FC and DPLL are backtracking based approaches, we will use the number of visited branches as indicator for the efficiency of each algorithm and in this way compare the impact of achieving arc-consistency from FC on the CSP with unit propagation from DPLL on the SAT problem. So, one algorithm outperforms (dominates) another if and only if it visits less number of branches assuming equivalent branching heuristics [4].

This criterion has been chosen because it serves as a good indicator for the performance of these algorithms in real world applications. This comparison will be aided with examples and comparisons at the end of the paper. Notice that FC is always dominated by MAC regarding this criterion, which does not always reflect the real performance but gives a good clue about the efficiency of each algorithm without diving too deep into the exact implementation.

3 CSP INTO SAT ENCODING

Mapping a CSP problem into a SAT instance can be done in many ways. Several methods of encoding a CSP instance as a propositional formula have been proposed. Some of the most common encodings will be introduced in this section.

3.1 Direct encoding

As the name suggests, the direct encoding assign for each CSP variable x and for each possible value from its domain i a new propositional variable x_i on the SAT side. The boolean variable is then True if and only if the original variable is assigned the value i. To ensure consistent logical assignment for each propositional variable, we need to extend the SAT problem to include clauses that avoid inconsistent results such as assigning a CSP variable two values at the same time or no value at all (at-least-once and at-most-once clauses) [1].

3.1.1 Example

Consider the following instance of the map coloring problem:

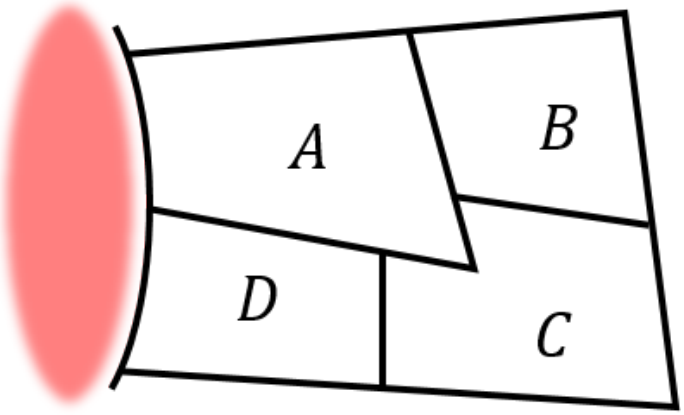

Fig. 1: Unsolved instance of the map coloring problem

The map consists of four regions A, B, C, D sharing boarder as shown in the figure 1. The purpose of this problem is to color each region of the map so that no two adjacent regions have the same color. The variables in this case are the regions themselves $\{A, B, C, D\}$ and each of them has the domain $\{r, g, b\}$ which stands for the colors red, green and blue respectively. To fully define the CSP we also need to define the constraints' set which comprise two types of constraints:

- unary constraints such as $A \neq r$ since A is already adjacent to a red colored region
- binary constraints such as $A \neq B, A \neq C, \ldots$ i.e. no two adjacent regions share the same color

Hereafter, the unary constraints are avoided by restricting the domain of A and D to not include red; on the other hand, it should be noted that usually all the variables are given the same domain to simplify the problem.

Encoding this problem directly will result in the following SAT instance: a new propositional variable is associated with each value that can be assigned to a CSP variable. So we get the variables: $\{A_g, A_b, B_r, B_g, B_b, \ldots\}$. To ensure that each CSP variable will be assigned at least one color, we have to include the following at-least-once (ALO) clauses: $\{A_g \vee A_b, B_g \vee B_r \vee B_b, \ldots\}$. Similarly, we must ensure that no CSP variable is assigned multiple colors at the same time, which can be expressed as these at-most-once (AMO) clauses: $\{A_g \oplus A_b, B_g \oplus B_r, B_g \oplus B_b, B_r \oplus B_b, \ldots\}$, where $\oplus$ stands for the logical exclusive or operation $a \oplus b \equiv (a \wedge \neg b) \vee (\neg a \wedge b) \equiv (a \vee b) \wedge \neg(a \wedge b)$.

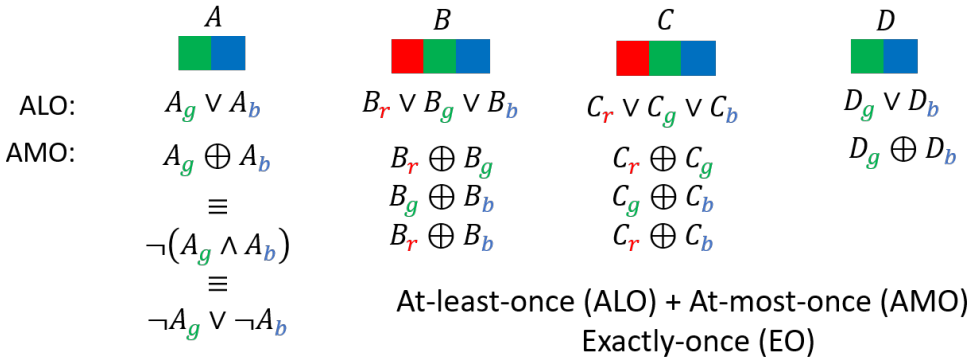

Fig. 2: Both types of clauses in direct encoded SAT

ALO and AMO clauses will enforce each CSP variable to be true only for one value from its domain, hence the name exactly-once (EO) clauses.

We still need to encode the binary constraints from CSP into some sort of clauses in SAT. This can be done easily by ensuring that no two CSP variables such as C and B share the same color, which can be translated into the following clauses:

$$\neg(C_r \wedge B_r) \quad , \neg(C_g \wedge B_g) \quad , \neg(C_b \wedge B_b)$$

The same principle applies for the reset.

The direct encoding can be categorized under a general type called the sparse encoding. Notice that the direct encoding encodes conflict points (so called conflict or no-good) assignments such as A and B can't share the same color at the same time $\neg(A_c \wedge B_c) \equiv \neg A_c \vee \neg B_c$. It is also possible to encode the allowed (so called supports) assignments, e.g. $\neg A_c \vee B_c \vee C_c \vee D_c$. This type of clauses is the basis for the other type of sparse encoding called accordingly the support encoding [1].

3.1.2 Proposition

Solving the CSP problem directly with FC compared to the direct encoded SAT instance of the same CSP problem solved with DPLL will always result in similar search tree structure. i.e. FC on the original CSP problem and DPLL on the direct encoded SAT will explore the same number of branches. This proposition assumes equivalent branching heuristics for DPLL and FC.

3.1.3 Proof idea

Considering the generated search tree, our version of DPLL comprise only two rules: the one literal and the branching rule. This means at each node in the tree we either have to branch or the given variable must be assigned a truth value of either False or True.

On the FC side branching is equivalent to its counterpart in DPLL. The second case in DPLL (unit propagation by the one literal rule) is also equivalent to applying forward checking by enforcing arc consistency and reducing the domain of each CSP variable to one value only.

Similarly, we can trace the whole structure of both trees and process with the same logic on each propositional variable. By induction, we can prove that each algorithm explore the same number of branches [4].

3.1.4 Example

Compare the following search trees of FC applied to the previous map coloring problem and DPLL applied to direct encoded instance of the same previous map-coloring problem:

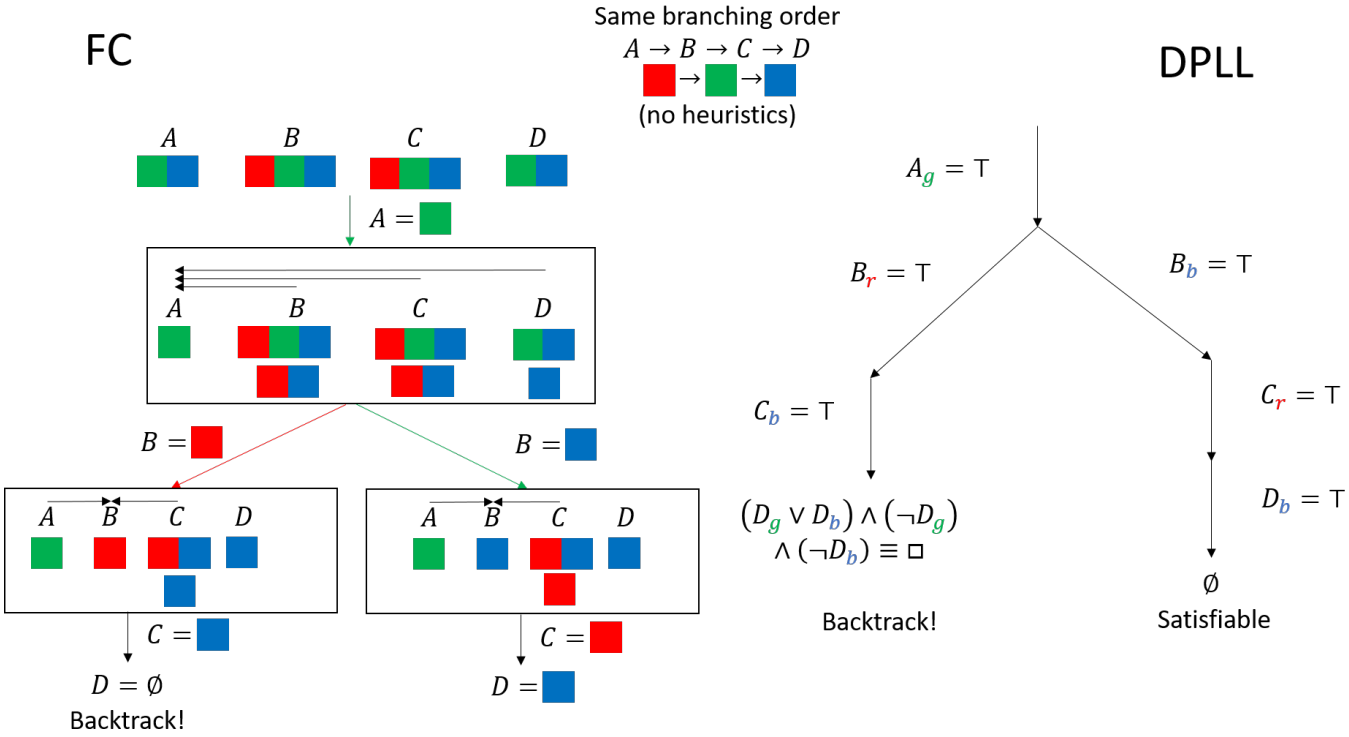

Fig. 3: Search trees of FC and DPLL

The arrows in node level on the FC side indicates enforcing arc consistency and the results are show directly underneath on the next line.

3.2 Log encoding

This encoding utilize the fact that each propositional variable can only take one of two values, i.e. either True or False.

3.2.1 Encoding the variables

First we encode each possible value from the domain with a distinct binary representation. Depending on the maximum number of bits needed to encoded all the domain values, an equal number of propositional variables for each CSP variable is generated. So if the size of domain was m and the number of CSP variables was n, we need $n\lceil \log_2 m \rceil$ propositional variable on the SAT side.

Notice that encoding each value from the CSP variable's domain means that the size of this domain must be exactly equal to 2^i, which is not necessary the case. So extra possible binary representations can emerge (implied in the formula by the ceiling function, i.e. if $\lceil \log_2 m \rceil > \log_2 m$). To avoid assigning these invalid values to any CSP variable we have to include extra clauses to ensure that a variable can't be assigned a value outside its domain.

On the other hand, unlike the direct encoding we do not need any additional clauses to ensure that each CSP variable is given a value at all (ALO clauses) nor to ensure that each CSP variable is given only one value (AMO clauses).

The same example from above is used to demonstrate the additional needed clauses. The CSP constraints are not listed explicitly but rather illustrated by the borders on the map.

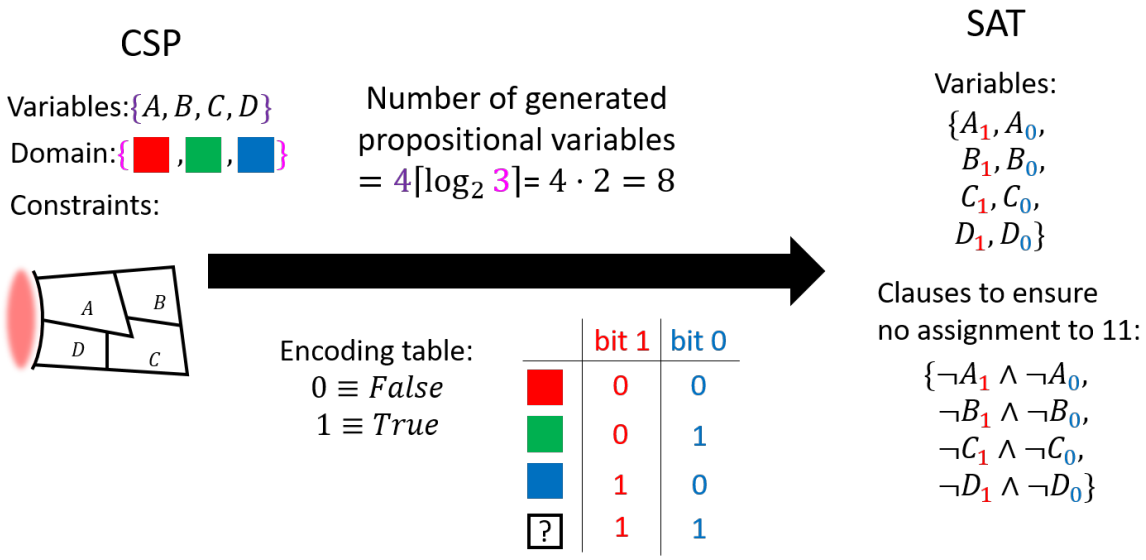

Fig. 4: Encoding CSP variables using log encoding

Notice that the log encoding generates less propositional variables than direct encoding. Anyway, this fact does not improve the performance of DPLL on the generated SAT instance because more steps are required to untangle the problem as discussed later. 3.2.3

3.2.2 Encoding the constraints

As discussed before, constraints could be unary, binary or n-ary. Since any n-ary constraint can be expanded to multiple binary constraints, the main focus will be on encoding only the unary and the binary constraints.

Encoding unary constraints is straight forward. For example to ensure that a CSP variable X can't be assigned a certain value from its domain encoded in the binary form 01, we only need to add the following clause to the SAT formula: $\neg(\neg X_1 \wedge X_0) \equiv X_1 \vee \neg X_0$. Remember that the subscript of the propositional variables X_0, X_1 corresponds to the position of the bit in the binary form of the CSP value.

Encoding binary constraints can be done in similar manner. For example to encode the CSP constraint $A \neq D$, where A and D can be either green ■ or blue ■ (i.e. their domain is {■, ■}) with the binary representations 01 and 10 respectively, we need to include the following clauses in our SAT:

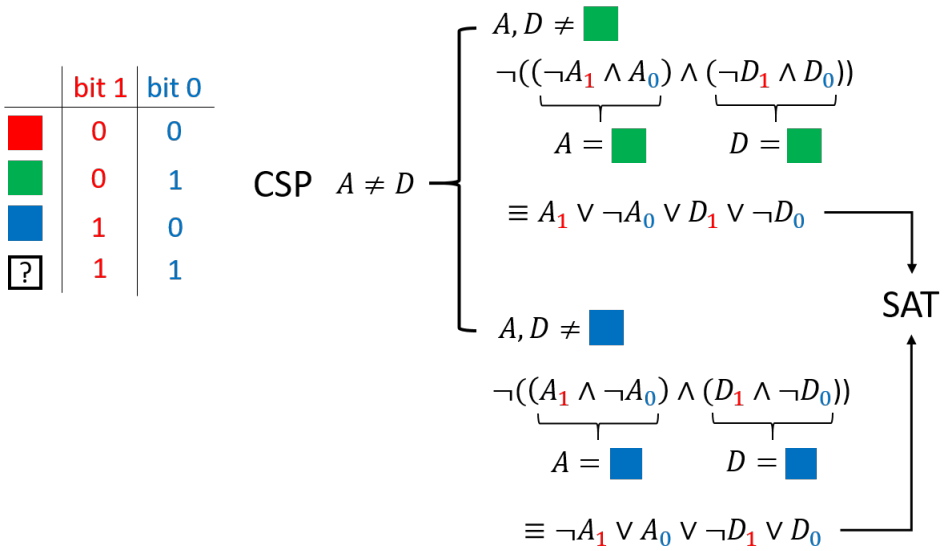

Fig. 5: Encoding binary CSP constraint using log encoding

3.2.3 Proposition

In spite of the fact that log encoding generate fewer variables on the SAT side than direct encoding, it is easy to see that DPLL applied on log encoded CSP is dominated by FC applied to the original CSP problem (assuming equivalent branching heuristics).

3.2.4 Proof idea

The proof idea is similar to the proof mentioned earlier for the direct encoding case 3.1.3. We process by comparing the search trees of each algorithm. First we show that both trees must have at least equal number of branches. By Induction on each CSP variable X, FC algorithm will branch upon it. On the SAT side DPLL will consider all propositional variables corresponding to the same CSP variable ($X_i \dots X_{\lceil \log_2 m \rceil}$), where m is the size of the domain.

Further more, we can exploit the natural logarithmic behavior of log encoding on the search tree to show that DPLL is **strictly** dominated by FC. This can be done using a simple example. Consider a CSP consists of two variables A and B with domain of size 3 and no constraints other than rolling out all assignments. FC will require three branches for each variable to show that this problem is unsatisfiable. Since the size of the domain equals 3, we need at least two bits to represent each CSP variable. Therefore, DPLL must branch at least $2*2*2 = 8$ times to show that the problem is unsatisfiable (the first 2 represents for the number of variables, the second one for the number of bits, and the last one to indicate branching on each case True or False as discussed before).

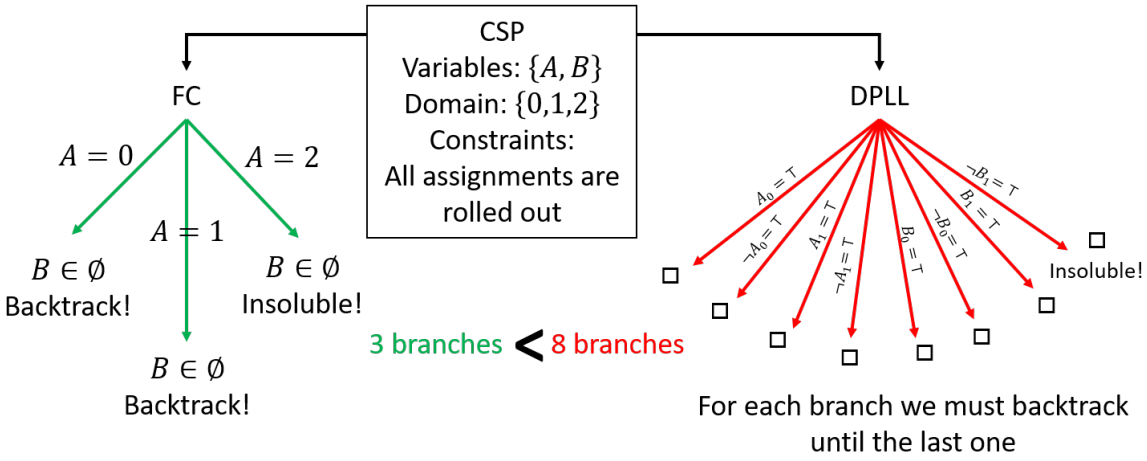

Fig. 6: Seach trees of DPLL and FC for trivial CSP

3.3 Order encoding

Order encoding took a different path to approach the problem of transforming CSP into SAT. The essential idea is extending the expressive power of Boolean expressions to represent a whole range of possible values on the CSP side as a comparison.

Despite being simple in its complexity, the order encoding is one if most efficient approaches to encode CSP into SAT and generate a relatively simpler instance that can be solved quickly under the assumption that the right conditions are met (see 3.3.5).

3.3.1 Usage

The core mechanism of many constraint solvers is based on another SAT-Solver (such as MiniSat[2]). The given CSP is first encoded using order encoding (or a variant of it) and then solved by the SAT-Solver. One of the first prominent award-winning SAT-based constraint solvers called sugar. It is based around the same idea using MiniSat as the underlying SAT-solver [2].

3.3.2 Encoding the variables

Assuming a consistent CSP domain for the variable x ranging from $l(x)$ to $u(x)$ where l and u represent the lower and upper bounds respectively, a Boolean variable is generated p_{xi} for each value i in the domain of x. p_{xi} is semantically equal to $x \leq i$, and therefor p_{xk} for $k = u(x)$ is always True and must be added as its own clause because it must be satisfied at all times.

To facilitate an unambiguous entry point for DPLL, the order encoding adds additional trivial propositional variable p_{xj} for each CSP variable x where $j = l(x) - 1$. This variable must be added as its own clause since it's clearly False at all times. Similar variables of the form p_{xk} for $k = u(x)$ are always True. Both of these clauses $\neg p_{xj} \wedge p_{xk}$ helps the unit propagation rule exploiting the SAT instance of the problem and constraint it by providing boundaries on each variable.

Since $x \leq i$ implies $x \leq i + 1$ and p_{xi} means $x \leq i$, we must also encode this fact into our SAT to ensure a consistent assignment of each variable:

$$x \leq i \rightarrow x \leq i + 1 \quad \equiv \quad \neg x \leq i \vee x \leq i + 1 \quad \equiv \quad \neg p_{xi} \vee p_{x(i+1)}$$

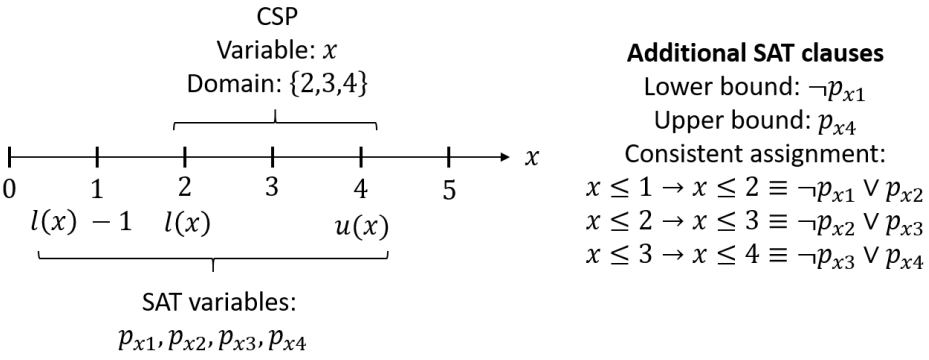

Fig. 7: An example for encoding CSP variables using order encoding

3.3.3 Encoding inequality constraints

Order encoding works best for the inequality constraints (concretely constraints of the form $\sum_i a_i x_i \leq b$). Other types of constraints will be discussed in the next section 3.3.4. The efficiency of the order encoding lays in its powerful encoding of entire conflict regions instead of single conflict points (in comparison with log and direct encodings) [2]. To encode a constraint of the form $\sum_{i=1}^{n} a_i x_i \leq b$, we must first generate the conflict regions for each variable which is a set of integer tuples $C := \{c_1, \ldots, c_n\}$ that satisfies the constraints:

$$\sum_{i=1}^{n} c_i = b - n + 1 \tag{1}$$

$$\bigwedge_{i=1}^{n} (\min(a_i \cdot x_i) \leq c_i \leq \max(a_i \cdot x_i)) \tag{2}$$

Then the required clauses to encode the constraint are:

$$\forall c_i \in C : \bigvee_{i=1}^{n} \begin{cases} x_i \leq \lfloor \frac{c_i}{a_i} \rfloor \equiv p_{x_i(\lfloor \frac{c_i}{a_i} \rfloor)} & a_i > 0 \\ \neg(x_i \leq \lceil \frac{c_i}{a_i} - 1) \rceil \equiv \neg p_{x_i(\lceil \frac{c_i}{a_i} - 1) \rceil)} & a_i < 0 \end{cases} \tag{3}$$

Example: To encode the constraint $x + y \leq 5$ for the unified domain $\{2, 3, 4\}$ we need to consider the conflict regions $C = \{(1, 3), (2, 2), (3, 1)\}$, so the following clauses must be included in our SAT formula:

$$p_{x1} \vee p_{y3} \quad (\bullet) \qquad p_{x2} \vee p_{y2} \quad (\times) \qquad p_{x3} \vee p_{y1} \quad (\circ)$$

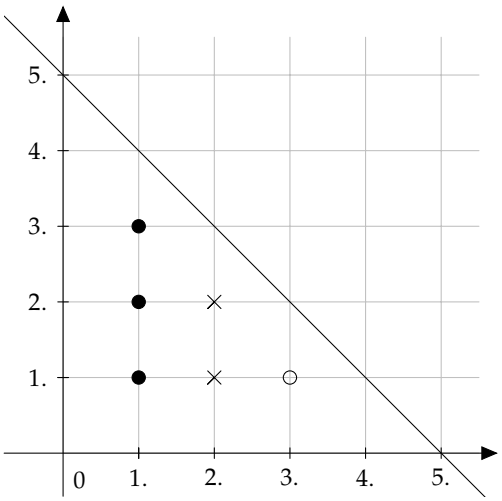

Fig. 8: An example for encoding the constraint $x + y \leq 5$ using order encoding

3.3.4 Encoding general constraints

Other types of constraints are replaced by equivalent expressions composed from one or multiple constraints of the form $\sum_i a_i x_i \leq b$ while adding extra conditions (clauses) if necessary.

For example to encode the expression $x < y$ we replace it by $x + 1 \leq y \equiv p_{(x+1)y}$, an equality expression between x and y can be then replaced by $x \leq y \wedge x \geq y \equiv x \leq y \wedge \neg(x < y) \equiv p_{xy} \wedge p_{(x+1)y}$, similarly $x \neq y \equiv x < y \vee x > y$ and so on.

The following table shows how the CSP-solver sugar tries to replace general expressions with constraints of the form $\sum_i a_i x_i \leq b$ while adding extra conditions if needed [2].

Original Constraint	Replacement	Extra condition
$\max(a, b)$	x	$(x \geq a) \wedge (x \geq b) \wedge ((x \leq a) \vee (x \leq b))$
$\min(a, b)$	x	$(x \leq a) \wedge (x \leq b) \wedge ((x \geq a) \vee (x \geq b))$
$\lvert a \rvert$	$\max(a, -a)$	
$a \mod b$	r	$(a = cq + r) \wedge (0 \leq r) \wedge (r < c)$

3.3.5 Importance

In theory and assuming worst-time performance, the order encoding is no better than the other type of encodings, but in real-world applications it has been shown that the order encoding have better performance than the direct encoding on instances of the graph coloring and open-shop scheduling problems. [3]

Moreover, the order encoding, unlike the other encodings discussed earlier, is the first encoding that can translate a CSP into a so called tractable SAT, which means that the SAT problem can be solved by a SAT-solver in polynomial time. However, this statement is true only for certain classes of CSP problems. It has been proven that the following tractable CSP classes can be encoded into a tractable SAT problems using the order encoding: [1]

1) constraint-closed
2) max-closed
3) connected row-convex

All the previous classes are some type of formality imposed on the constraints' set of the CSP problem. Comparisons and tables of SAT-solvers performance on the direct, log and order encoding of different tractable CSP instances can be found in [1].

3.4 Hybrid encoding integrating Log and Order encoding

Exploiting the pros of each encoding, a new hybrid encoding emerged. Log encoding represents each value in CSP's domain as binary integer and as a result generates small SAT problems regarding the number of propositional variables and the number of constraints (compared to the direct encoding since no additional clauses are required). On the other hand, the order encoding is known for its performance, then it generates additional variables that bounds the SAT problem and make it more liable for being solved directly by the one literal rule using unit propagation instead of branching. But each encoding has its limitations; for instance: the order encoding generate huge SAT instances when the CSP domain or the arity of the constraints becomes large. Likewise, the log suffers from poor performance in general.

3.4.1 Core idea

Other encodings have already proposed merging multiple encodings together to benefit from each of them, but no true hybridization have been applied. Each variable was encoded with both types of used encodings and then additional clauses were added to channel both types of generated constraints together. This proposed encoding works without channeling the constraints and using a criterion to determinate the appropriate encoding for each variable. Therefore, the resulted constraints can contain both types of encoded variables and don't require additional channeling constraints. Log and order encodings operates as partial encoding under the regard of the criterion. The authors proposed the domain product as the classification criterion but other criteria could have been used. [5]

3.4.2 Performance test

The authors implemented their own CSP-Solver and tested it against two sets of benchmark problems. The first set was handmade and the second one was selected from a CSP solver competition. In both tests the hybrid encoding showed better results and superior performance compared to applying the partial encodings alone. Detailed tables with exact figures and details about the test environment are provided in the original paper [5]. It should be noted that this some problems with high artiy and domain size could be only solved with the hybrid encoding in a reasonable amount of time.

4 SAT INTO CSP ENCODINGS

In this section, we discuss the opposite direction by introducing several ways for encoding SAT into CSP and briefly summarize the pros and cons of each encoding.

4.1 Dual encoding

For each clause in the CNF-formatted SAT a CSP variable is generated. The domain of each variable consists of the possible truth assignments that satisfies the corresponding clause expressed as a tuple. Any two clauses that share a propositional variable must also have a binary constraint between their respective dual variables. These constraints assure consistent assignment of the common propositional variable, such that no variable is assigned two different values at the same time.

The following example shows how we could encode a SAT problem with two clauses and one common propositional variable. Considering the clauses: $x_1 \vee x_2$ and $x_3 \vee \neg x_2$. The first clause results in the dual variable D_1 with the domain: $\{(T,T),(T,F),(F,T)\}$ where T and F represent the truth value of the corresponding Boolean variable at the respective position within the tuple. The second clause results in $D_2 \in \{(T,F),(T,T),(F,F)\}$. Since x_2 appears in both clauses, we must ensure that it won't have different values in each variable through the binary constraint: $D_1[2] = \neg D_2[2]$ which assures that the second element of the tuple assigned to D_1 is the inverse of the second element of the D_2 tuple.

4.1.1 Theoretical comparison

It has been proven [4] that FC on the dual encoded SAT instance does more work than DPLL on the original problem. The proof is based on the fact that a truth assignment under unit propagation in DPLL implies an elimination of contradictory values on the CSP side. By induction, we can see that generating the empty clause in DPLL (which indicates that the problem is unsatisfiable regarding the current branch) implies a domain wipe out after applying arc-consistency by FC.

4.2 Hidden variable encoding

Just like dual encoding, the hidden variable encdoing assign a dual variable for each SAT clause. Furthermore, each propositional variable x_i is simulated on the CSP side with a binary variable $X_i \in \{T, F\}$. This encodings allows constraints to be easily formulated in terms of thier binary variables. For example the clause $x_1 \vee \neg x_2$ results in one dual variable $D_1 \in \{(T, F), (F, F), (T, T)\}$ and two binary CSP variables X_1, X_2 and additionally the coupling constraints $D_1[1] = x_1$ and $D_1[2] = \neg x_2$ which constraints any assignment of the elements of the tuple assigned to D_1 to be consistent with the propositional variables x_1, x_2 represented by their CSP counterparts X_1, X_2.

4.2.1 Theoretical comparison

The hidden variable encoding maintain equivalent relation between assigning truth values committed by one literal rule on the DPLL side and eliminating contradictory values by enforcing arc-consistency by FC/MAC on the node level. This translates to equivalence relation between generating empty clauses and empty domain [4]. This in turn means that DPLL will essentially branch in general as much as FC/MAC applied to the encoded version of the same problem, and the same amount of work is expected by each algorithm.

4.3 Literal encoding

The literal encoding express each clause as a variable with a domain consisting of propositional variables' values that satisfy the corresponding clause. Constraints are required between any CSP variables if and only if their domains contain contradictory propositional variables. For example the clauses $x_1 \vee x_2$ and $x_3 \vee \neg x_2$ can be expressed as the following variables $D_1 \in \{x_1, x_2\}, D_2 \in \{x_3, \neg x_2\}$ and the constraint $\neg(D_1 = x_2 \wedge D_2 = \neg x_2)$ which rules out this inconsistent assignment for x_2.

4.3.1 Theoretical comparison

This encoding shows similar results as the hidden variable encoding regarding the relation between generating the empty clause and empty domains as a result of enforcing arc-consistency. Anyway MAC is strictly dominated by DPLL applied to the original. Strictness can be proved easily by contradiction. Consider any unsatisfiable $k-$SAT instance (i.e. a SAT problem with k propositional variable) with 2^k clauses where $k > 2$. DPLL needs 2^{k-1} branch to arrive at the empty clause while MAC takes $k!$ branches regardless of the used heuristics [4].

4.3.2 Advantage of literal encoding

CSP domain size can have great impact on the performance of any arc-consistency based algorithm (like MAC or FC) and must be taken into consideration. Let n be the number of SAT clauses. Dual and hidden variable encodings generate CSP instance with domain size in the order of $O(2^n)$ while the literal encoding generate CSP instances with domain size of $O(n)$

5 Conclusion

We discussed some mappings between the constraint satisfaction problem (CSP) and the Boolean satisfiability problem (SAT). Some theoretical background has been introduced and based on general comparing criterion the encodings have been analyzed and compared against each other. It has been shown that the choice of an encoding has a large impact on the efficiency and solvability of the generated problem. Encoding CSP into simpler SAT problem has led to efficient instances of the original problem that can be solved quickly. This claim was supported with theoretical arguments and proofs specially in the case of some tractable CSP classes encoded using the order encoding. Based on similar encoding techniques various new SAT-based constraint solves have been developed. On the other hand, a variety of SAT to CSP encodings have been introduced. Although no SAT to CSP encoding has proven itself to be superior, a clearer picture of the relation between CSP and SAT has emerged and further studies can build upon these results.

REFERENCES

[1] J. Petke and P. Jeavons, "The order encoding: From tractable csp to tractable sat." in *SAT*. Springer, 2011, pp. 371–372.

[2] N. Tamura and M. Banbara, "Sugar: A csp to sat translator based on order encoding," *Proceedings of the Second International CSP Solver Competition*, pp. 65–69, 2008.

[3] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara, "Compiling finite linear csp into sat," *Constraints*, vol. 14, no. 2, pp. 254–272, 2009.

[4] T. Walsh, "Sat v csp," *Principles and Practice of Constraint Programming–CP 2000*, pp. 441–456, 2000.

[5] T. Soh, M. Banbara, and N. Tamura, "A hybrid encoding of csp to sat integrating order and log encodings," in *Tools with Artificial Intelligence (ICTAI), 2015 IEEE 27th International Conference on*. IEEE, 2015, pp. 421–428.

[6] P. Barahona, S. Hölldobler, and V.-H. Nguyen, "Efficient sat-encoding of linear csp constraints." in *ISAIM*, 2014.

[7] P. Jeavons and J. Petke, "Local consistency and sat-solvers," *Journal of Artificial Intelligence Research*, vol. 43, pp. 329–351, 2012.