Learning SAT encodings

Felix Ulrich-Oltean

December 9, 2019

Script for Literature Review Seminar

Solving Problems by Searching for Truth: from the CSP to SAT via constraint encodings

0.1 Setting the Scene

Scope of this seminar

My research is about Constraint Satisfaction Problems and in particular finding efficient ways of expressing a CSP as an instance of the Boolean Satisfiability Problem.

To give you a little bit of context, I last studied at university in 2000 when I did my Computer Science degree here at York. That means I've got a lot of catching up to do.

My focus early on in the PhD has been to begin to understand the key principles and methods in the field, to consider the foundational contributions to the state of the art and to get a sense of the direction of travel or the strands of enquiry generating considerable research activity.

Today I will try to set out broadly my research context by referring to literature that has helped me so far.

Constraint Satisfaction: Hands-on

As this is a seminar rather than a lecture, there is a little bit of thinking involved, if you're up for it.

Fill the Christmas baubles using the numbers 1 to 12 **exactly once** each, so that no connected baubles have consecutive numbers AND every clique of 3 baubles has a sum of no more than 20.

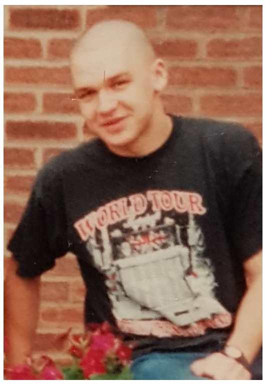


Figure 0.1: York the first time round

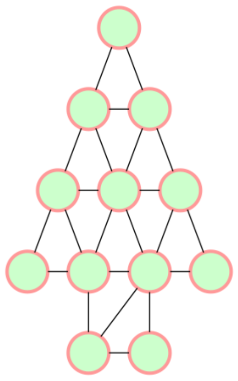


Figure 0.2: Christmas Tree Puzzle

Constraint Satisfaction In Real Life

Real-life applications are widespread, a few examples being:

- Scheduling or timetabling especially where several additional constraints exist. Cambazard et al.[1] address the "post enrolment-based course timetabling problem" in which students, lecturers and rooms are to be scheduled in a way which meets both some hard constraints (i.e. requirements) and some soft constraints (desirable features of a solution).
- Ansótegui et al. [2] feature the Multi-mode Resource-Constrained Project Scheduling Problem (MRCPSP) which seeks to schedule a number of tasks in a project, taking into account that tasks must obey a precedence ordering, use resources (which may be renewable or exhaustible), and can be carried out in different "modes". In the same paper, the authors also tackle a version of the Nurse Scheduling Problem.
- In his "Boolean Satisfiability and Beyond" paper [3], Järvisalo reviews the use of SAT in computational argumentation as well as in a variety of machine learning applications, such as structure learning of Bayesian networks, as described for example in [4].
- I came across an interesting application to Master-Key Lock systems in Martin Hořeňovský's Masters thesis [5].
- Many other applications exist such as model checking (or formal verification of systems) and cryptographic schemes which rely on the hardness of Boolean Satisfiability. A lot of the papers I have read begin with a listing of the applications of the CSP.

Formal definition

Having attempted a toy CSP ourselves, and seen some of the industrial applications, let's try to give a formal definition.

A Constraint Satisfaction Problem (CSP) consists of:

- A set of variables $X = \{X_1, \ldots, X_n\}$
- A set of domains corresponding to each variable $D = \{D_1, \ldots, D_n\}$

Figure 0.3: Mission control for Apollo 13 (the Hollywood version)

Figure 0.4: Nurses' schedule on a ward

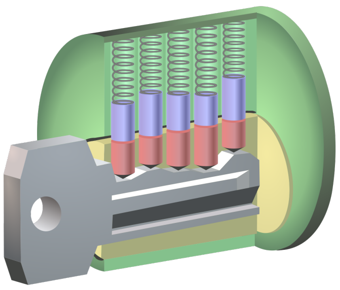


Figure 0.5: A key and lock mechanism

- A set of constraints, each concerned with a subset of the variables and imposing a relation:

$$C = \{\langle S_j, R_j\rangle | S_j \subseteq X, R_j \subseteq \{D(S_{j,1})\times, \ldots, \times D(S_{j,m})\}, m = |S_j|\}$$

Apt [6, p. 9] formalises a CSP $\mathcal{P}$ as a set of constraints $\mathcal{C}$ and a set of domain expressions $\mathcal{DE}$:

$$\mathcal{P} = \langle \mathcal{C}; \mathcal{DE}\rangle$$

which allows a more compact definiton.

For instance, our Christmas Tree problem becomes

$$\begin{aligned}
&\langle \text{alldifferent}(x_1 \ldots x_{12}),\\
&\forall a, b, c : \mathit{isClique}(a, b, c) \Rightarrow x_a + x_b + x_c \leq 20,\\
&\forall d, e : \mathit{edge}(d, e) \Rightarrow |x_d - x_e| > 1;\\
&x_1 \ldots x_{12} \in \{1 \ldots 12\}\rangle
\end{aligned}$$

where *isClique*() and *edge*() have the obvious meaning for our undirected graph representing the puzzle.

The Problem → Solution pipeline

Solving a CSP can be done in many ways, but generally the following steps are involved:

$$\text{Problem} \xrightarrow{\mathit{modelling}} \text{Model} \xrightarrow{\mathit{encoding}} \text{Formula} \xrightarrow{\mathit{solving}} \text{Result}$$

This is in some ways analogous to the idea of writing software more generally

$$\text{Specification} \xrightarrow{\mathit{programming}} \text{Source code} \xrightarrow{\mathit{compiling}} \text{Machine code} \xrightarrow{\mathit{execution}} \text{Application}$$

When considering the "Model" and the "Formula", there are similarities in terms of the compromises to be made between readability and expressive power on the one hand and performance or control on the other.

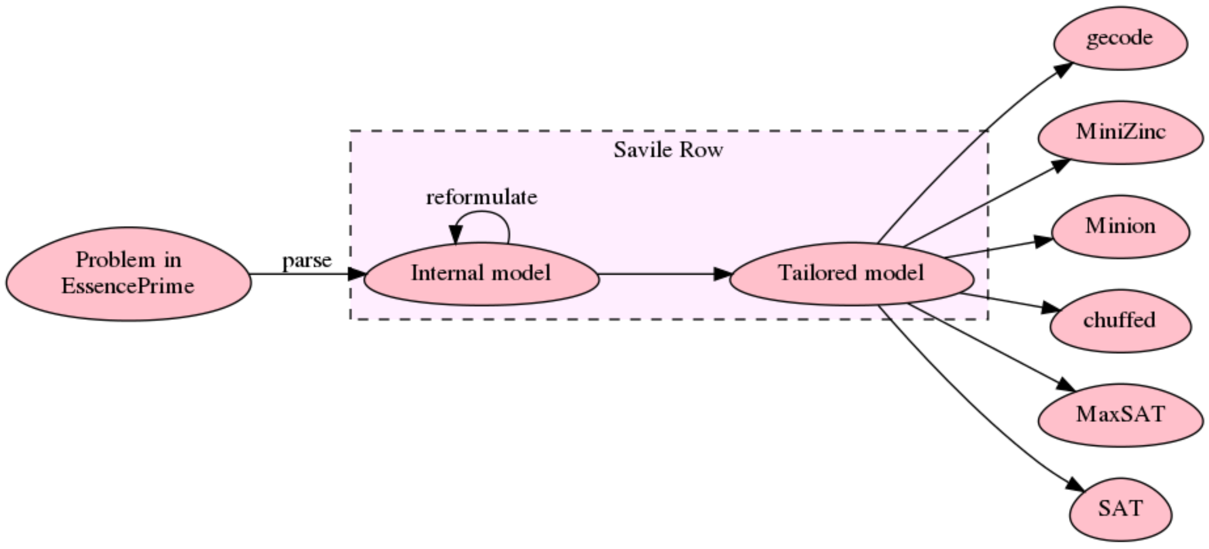


Figure 0.6: Schematic of Savile Row

Savile Row

Peter Nightingale [7] introduced Savile Row, a “modelling assistant” tool which takes a model written in a high-level language called EssencePrime [8] and can prepare a “tailored” definition for any of a range of possible back-end solvers. In the process of translating, it can apply various reformulations and optimisations, some of which apply in all cases, others being target-specific. One exciting aspect is that Savile Row can produce SAT formulae in the format expected by SAT solvers (dimacs cnf), so the shiniest latest SAT solver can be employed to perform the (almost) final solving step, before Savile Row re-constructs the solution into the original EssencePrime format.

In a later paper [9], Nightingale describes in more detail the reformulation steps that Savile Row employs to make a more compact and efficient model for the back end solver. This journal paper also provides a very extensive empirical analysis of the effectiveness of different combinations of reformulations on performance.

To give you an idea of what a problem definition looks like in EssencePrime, Figure 0.7 shows a model that could describe our Christmas Tree problem.

0.2 SAT Solvers

Going back to the analogy of conventional programming, it can help to understand a little about the back-end solver operation in order to prepare “better” code. A big part of my learning has been about SAT solvers.

Why SAT solvers as a back end?

The simplicity of the input format means that there are hundreds of SAT solvers available, and they are being developed all the time, advancing the scale of problem that can be tackled. Yearly SAT solver competitions are held, inviting contestants to describe their solver characteristics and to make their solvers open source. The proceedings of recent SAT competitions [10, 11] are very informative.

```
language ESSENCE' 1.0

letting VALS be domain int(1..12)

letting EDGE be [
[false,true ,true ,false,false,false, false,false,false,false,false,false],
[true ,false,true ,true ,true ,false, false,false,false,false,false,false],
[true ,true ,false,false,true ,true , false,false,false,false,false,false],
[false,true ,false,false,true ,false, true ,true ,false,false,false,false],
[false,true ,true ,true ,false,true , false,true ,true ,false,false,false],
[false,false,true ,false,true ,false, false,false,true ,true ,false,false],

[false,false,false,true ,false,false, false,true ,false,false,false,false],
[false,false,false,true ,true ,false, true ,false,true ,false,true ,false],
[false,false,false,false,true ,true , false,true ,false,true ,true ,true ],
[false,false,false,false,false,true , false,false,true ,false,false,false],
[false,false,false,false,false,false, false,true ,true ,false,false,true ],
[false,false,false,false,false,false, false,false,true ,false,true ,false],
]
find A : matrix indexed by [VALS] of VALS

such that

allDiff(A),

$ no two adjacent assignments are consecutive
forAll x,y : VALS .
  (x!=y) -> (EDGE[x,y] -> ( |A[x]-A[y]| != 1 ) ),

forAll x,y : VALS .
  (x!=y /\ EDGE[x,y]) ->
    (forAll z: VALS . (EDGE[x,z] /\ EDGE[y,z]) -> (A[x]+A[y]+A[z] < 21)),

$ symmetry
A[7] < A[9]
```

Figure 0.7: A definition of the Christmas Tree Problem in EssencePrime for Savile Row

The results give useful insights into the approaches that are most productive and advances in aspects such as parallel searching and in how hybrid algorithms are being developed.

How do SAT solvers work?

A reminder of the famous Boolean Satisfiability Problem.

Given a propositional formula, can a complete assignment of variables be found such that the formula is satisfied? For example, the formula

$$F = (a \vee \neg b) \wedge (b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$$

can be satisfied by the assignment $A = \{a \rightarrow true, b \rightarrow false, c \rightarrow true\}$

The most naive approach would be to systematically try out all possible assignments and declare victory when an assignment satisifes or defeat when all possible assignments have been tried. Clearly this is potentially a very slow approach.

Table 0.1: Rules in DPLL

Original rule	Use in SAT solver
I. Rule for the Elimination of One-Literal Clauses	Unit Propagation or Boolean Constraint Propagation
II. Affirmative-Negative Rule	Pure Literal Assignment
III. Rule for Eliminating Atomic Formulas	Not used directly

Backtracking and DPLL

Davis and Putnam [12], working on an automated proof system, introduced a set of rules by which a formula expressed in Conjunctive Normal Form (CNF) could be simplifed iteratively to reach a result more quickly.

Together with Logemann and Loveland [13] they then tweaked these rules and produced a working computer program - the template for the now famous DPLL algorithm.

Their three rules are shown in Table 0.1.

For example, working with the formula

$$\begin{aligned} F =&(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge \\ &(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \end{aligned} \tag{1}$$

We first look for clauses with a single unassigned literal, or for "pure literals". We find none at first, so we make a choice... let's start with $A = \{x_1 \to 0\}$, giving us a new formula:

$$\begin{aligned} F =&(0 \vee x_2 \vee x_3) \wedge (0 \vee x_2 \vee \neg x_3) \wedge (0 \vee \neg x_2 \vee x_3) \wedge \\ &(0 \vee \neg x_2 \vee \neg x_3) \wedge (1 \vee x_2 \vee x_3) \wedge (1 \vee x_2 \vee \neg x_3) \end{aligned} \tag{2}$$

Again, no single unassigned literals, so choose another variable. Now $A = \{x_1 \to 0, x_2 \to 0\}$, leading to:

$$\begin{aligned} F =&(0 \vee 0 \vee x_3) \wedge (0 \vee 0 \vee \neg x_3) \wedge (0 \vee 1 \vee x_3) \wedge \\ &(0 \vee 1 \vee \neg x_3) \wedge (1 \vee 0 \vee x_3) \wedge (1 \vee 0 \vee \neg x_3) \end{aligned} \tag{3}$$

Now as we scan across, we can see a "unit clause" straight away, as the first clause has just one unassigned literal. We therefore set the value of x_3 accordingly, so our assignment is now $A = \{x_1 \to 0, x_2 \to 0\, x_3 \to 1$ and our formula becomes:

$$\begin{aligned} F =&(0 \vee 0 \vee 1) \wedge (0 \vee 0 \vee 0) \wedge (0 \vee 1 \vee 1) \wedge \\ &(0 \vee 1 \vee 0) \wedge (1 \vee 0 \vee 1) \wedge (1 \vee 0 \vee 0) \end{aligned} \tag{4}$$

We now have a problem - the current assignment has made the second clause "empty" or unsatisfiable, so we backtrack, undoing the last variable **choice**. Remember that $x_3 \to 1$ has been set by unit propagation, not as a choice. So we undo the assignment $x_2 \to 0$ and any unit propagation that followed. We now assign $x_2 \to 1$ and have:

$$\begin{aligned} F =&(0 \vee 1 \vee x_3) \wedge (0 \vee 1 \vee \neg x_3) \wedge (0 \vee 0 \vee x_3) \wedge \\ &(0 \vee 0 \vee \neg x_3) \wedge (1 \vee 1 \vee x_3) \wedge (1 \vee 1 \vee \neg x_3) \end{aligned} \tag{5}$$

The only unsatisfied clauses are the third and fourth; unit propagation will set $x_3 \to 1$ for the third clause but then fail on the fourth clause. So, we backtrack all the way to our choice of x_1 and change our assignment to $x_1 \to 1$, giving us:

$$\begin{aligned} F =& (1 \lor x_2 \lor x_3) \land (1 \lor x_2 \lor \neg x_3) \land (1 \lor \neg x_2 \lor x_3) \land \\ & (1 \lor \neg x_2 \lor \neg x_3) \land (0 \lor x_2 \lor x_3) \land (0 \lor x_2 \lor \neg x_3) \end{aligned} \tag{6}$$

Nothing can unit propagate yet, so we choose $x_2 \to 0$ (as before) and obtain:

$$\begin{aligned} F =& (1 \lor 0 \lor x_3) \land (1 \lor 0 \lor \neg x_3) \land (1 \lor 1 \lor x_3) \land \\ & (1 \lor 1 \lor \neg x_3) \land (0 \lor 0 \lor x_3) \land (0 \lor 0 \lor \neg x_3) \end{aligned} \tag{7}$$

We observe the same issue as before: a conflict in the final two clauses. If we backtrack and change our assignment for x_2, we finally satisfy all our clauses, and the assignment to x_3 makes no difference:

$$\begin{aligned} F =& (1 \lor 1 \lor x_3) \land (1 \lor 1 \lor \neg x_3) \land (1 \lor 0 \lor x_3) \land \\ & (1 \lor 0 \lor \neg x_3) \land (0 \lor 1 \lor x_3) \land (0 \lor 1 \lor \neg x_3) \end{aligned} \tag{8}$$

Learning clauses, GRASP, CDCL

The DPLL algorithm uses chronological backtracking which means undoing the last variable choice, but this can often lead to a blockage in the same clause later on, which was not dependent on that particular assignment.

Silva and Sakallah introduced GRASP [14], a solver which used Conflict-Driven Clause Learning. All the modern winning SAT solvers use the CDCL method, which again uses unit propagation to make assignments which follow logically. This time, when a conflict occurs, the algorithm has a diagnosis step which tries to figure out which combination of assignments is blocking the formula - a new clause is then created which avoids failing at the same stage again. The algorithm then uses “back-jumping” to undo several assignments - back to the most recent assignment which caused the conflict.

CDCL uses an implication graph to help work out which assignments caused the conflict and to generate a new learnt clause.

Figure 0.8 shows how the assignments are represented using an implication graph in [14].

In fact there are a lot more details to how these solvers work, including:

- throwing away a lot of learned clauses to avoid being slowed down by a huge formula,
- variable choice heuristics, and
- re-starts (throwing away all assignments, but keeping the learnt clauses).

Importance of unit propagation

In both of these cases, the main driving force is unit propagation. This is very important when it comes to representing problems in CNF in a way that SAT solvers can make the most of. For example, keeping clauses short means that once a variable in the clause is assigned (without satisfying the clause), another implication follows sooner.

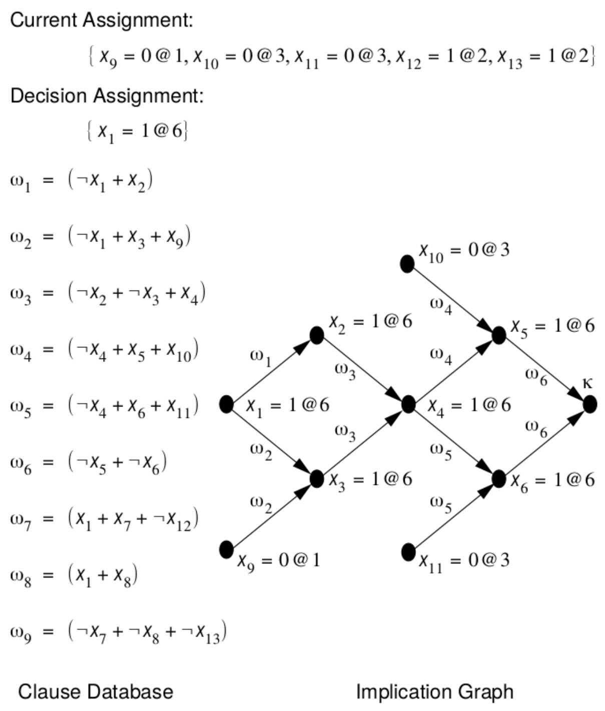

Current Assignment:
{ $x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x_{13} = 1@2$ }
Decision Assignment:
{ $x_1 = 1@6$ }
$\omega_1 = (\neg x_1 + x_2)$
$\omega_2 = (\neg x_1 + x_3 + x_9)$
$\omega_3 = (\neg x_2 + \neg x_3 + x_4)$
$\omega_4 = (\neg x_4 + x_5 + x_{10})$
$\omega_5 = (\neg x_4 + x_6 + x_{11})$
$\omega_6 = (\neg x_5 + \neg x_6)$
$\omega_7 = (x_1 + x_7 + \neg x_{12})$
$\omega_8 = (x_1 + x_8)$
$\omega_9 = (\neg x_7 + \neg x_8 + \neg x_{13})$
$x_{10} = 0@3$
$x_2 = 1@6$
$x_5 = 1@6$
$x_1 = 1@6$
$x_4 = 1@6$
κ
$x_3 = 1@6$
$x_6 = 1@6$
$x_9 = 0@1$
$x_{11} = 0@3$
Clause Database
Implication Graph

Figure 0.8: An implication graph created by unit propagation leading to a conflict

0.3 SAT Encodings

So we come to the focus of my research area. As we've seen, SAT solvers generally expect a formula in Conjunctive Normal Form (CNF). The usual file format is the DIMACS CNF standard.

We need therefore a method to translate our problem with its variables and constraints into an instance of SAT.

Not all is true or false

Non-boolean variables need to be represented as booleans. There are basic rules that can be applied, e.g. integers can be encoded using a unary encoding (e.g. $a = 6$ where $a \in \{0 \dots 15\}$ would be represented by $a_0 = 0, a_1 = 0, \dots, a_6 = 1, \dots, a_{15} = 0$) or a binary encoding.

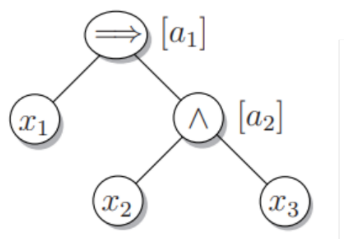
Naive encoding

If we limit ourselves to propositional logic, then any formula can be easily translated into Conjunctive Normal Form by using DeMorgan's laws and distributive laws, replacing implication and simplifying double negation, as follows:

$$\begin{aligned} \neg(a \vee b) &\longrightarrow \neg a \wedge \neg b \\ \neg(a \wedge b) &\longrightarrow \neg a \vee \neg b \\ a \vee (b \wedge c) &\longrightarrow (a \vee b) \wedge (a \vee c) \\ a \Rightarrow b &\longrightarrow \neg a \vee b \\ \neg\neg a &\longrightarrow a \end{aligned}$$

Example 1.21. Given a propositional formula

$$x_1 \implies (x_2 \land x_3) \ ,$$



$$\begin{aligned} a_1 &\iff (x_1 \implies a_2) \ , \\ a_2 &\iff (x_2 \land x_3) \ . \end{aligned}$$

$$\begin{aligned} &(\neg a_2 \lor x_2) && \land \\ &(\neg a_2 \lor x_3) && \land \\ &(a_2 \lor \neg x_2 \lor \neg x_3) \ . \end{aligned}$$

$$\begin{aligned} &(a_1 \lor x_1) && \land \\ &(a_1 \lor \neg a_2) && \land \\ &(\neg a_1 \lor \neg x_1 \lor a_2) \ , \end{aligned}$$

$$(a_1)$$

Figure 0.9: A basic example of the Tseitin encoding from Decision Procedures [16, pp. 12–13]

Tseitin's encoding

However, using these rules naively can lead to very large formulae. Tseitin [15] introduced an encoding which used additional variables but cut the size of the resulting formula. The original and new formulae are **equisatisfiable**, i.e. one is satisfiable iff the other one is. [16] gives a simple example to illustrate this process, shown in Figure 0.9.

This is a useful start to thinking about more concise encodings. A lot of work has gone into developing encodings for particular constraints.

Encoding constraints specifically

Our focus is on implementing constraints - we've already seen the *allDifferent* constraint in our Christmas Tree puzzle to ensure no variable is assigned the same value. Many other constraints exist. Beldiceanu and colleagues set up a catalogue of global constraints [17] with hundreds of entries.

Our own Alan Frisch conducted a review [18] of some of the encodings available at the time for the "at-most-k" constraint - in this paper the encodings are discussed and empirically compared on a range of problems. More recently, an example of work in this area is [19], which takes four different encodings of the "Pseudo-boolean" constraints (more on this later) and combines them with the At-Most-One constraint to obtain four new encodings which are then tested and compared. Significant reductions in the size of the encodings are made when these constraints are combined.

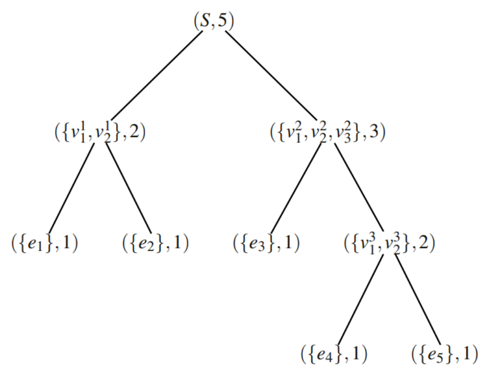


Figure 0.10: The binary tree for the Totalizer encoding

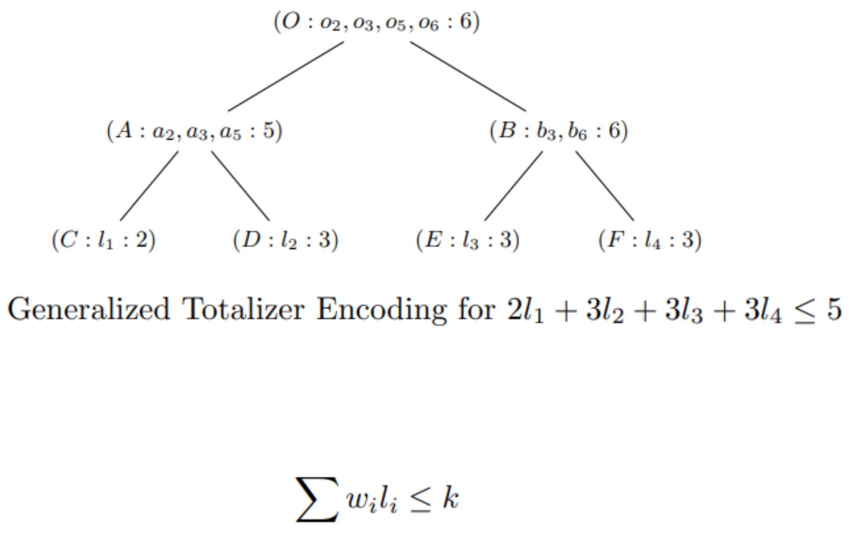


Figure 0.11: The binary tree for the Generalized Totalizer encoding

An example of development in representing constraints

To give an idea of how an encoding is built upon, here is an example of a line of enquiry or development.

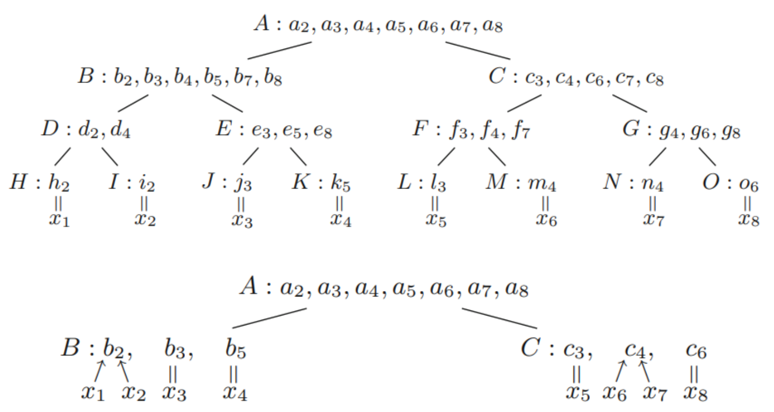
The "totalizer/comparator" encoding [20] was conceived as a way to implement a cardinality constraint, i.e. given a set of decision variables $e_1, \ldots\ e_n$, constrain the number of variables set to true to a range of values between μ and ρ inclusive. This is achieved by building a binary tree (Figure 0.10) and propagating the number of "true" variables up the tree. At the root node, the total is checked against the limits of the constraint.

This scheme was then used in [21] to encode the "Pseudo-boolean" constraint. This constraint is over a set of decision variables and their associated "weights", so that

$$w_1 l_1 + w_2 l_2 + \ldots + w_n l_n \leq k$$

The binary tree used in the totalizer was now slightly modified (see Figure 0.11) to account for the weights and thus the "Generalized Totalizer" encoding was born.

In 2019, Bofill et al. published a paper in which they combined the At-Most-k constraint with the Pseudo-boolean constraint – they considered four different encodings for the PBC and adapted each one to take into account a partition on the decision variables



At the top: binary tree of $GT(2x_1+2x_2+3x_3+5x_4+3x_5+4x_6+4x_7+6x_8 \leq 7)$. At the bottom: binary tree of $GGT(2x_1+2x_2+3x_3+5x_4+3x_5+4x_6+4x_7+6x_8 \leq 7, \{\{x_1, x_2, x_3, x_4\}, \{x_5, x_6, x_7, x_8\}\})$.

Figure 0.12: The binary tree for the GGT encoding

such that only one variable was allowed to be true in each “AMO group”. One of the four results was then the Generalized Generalized Totalizer encoding. Figure 0.12 shows how the tree is collapsed to reduce the number of variables requried.

What’s in an encoding?

Size but also propagability. As Bailleux et al. [22] state,

> Obviously, all things being equal, the more a solver propagates, the more efficient it is. On the other hand, encodings which UP-maintain GAC generally produce larger formulae than the other ones because they must encode each potential implication of a literal. Of course, larger formulae slow down unit-propagation. It is then not always clear which is the best trade-off between the size of encodings and their ability to enforce propagations.

0.4 Applying Machine Learning

Expert choices

How a problem is modelled and subsequently encoded can make a huge difference to the feasability of finding a solution. The idea of this PhD project is to see if we can machine learn how to make these choices.

I have found some interesting applications of machine learning to the type of process we’re interested in.

A review of portfolio solvers

In [23], we have a survey paper which considers many attempts to select the best search algorithm to be applied to combinatorial problems. There are many interesting aspects to bear in mind:

- The make-up of the portfolio of algorithms - how does one choose which algorithms are worthy of inclusion?
- The scoring of the algorithms? The portfolio as a whole can be considered, with ML predicting the best algorithm, or each algorithm can be scored such that the prediction is its performance.
- Should one algorithm only be chosen for a problem, or should we hedge our bets and schedule different algorithms to work on the problem according to some time allocation scheme? Algorithms can also be set off in parallel, again choosing a distribution according to some scoring method.
- Some approaches were offline, i.e. the problem was presented, the algorithm was chosen and then the solving began; in other cases, the algorithm choice was being made and reviewed as the problem was being solved.

SATZilla [24] is perhaps the most famous example of a portfolio solver, making use of third-party SAT solvers and selecting them per problem instance. Portfolio-based SAT solvers have been banned from the recent SAT races as they're deemed to reduce the incentive to improve the individual solvers themselves.

Two-step (hierarchical) approach

Peter suggested I look at Barry Hurley's PhD thesis [25] which makes several interesting contributions.

He proposed and implemented a "hierarchical portfolio", making choices in different stages: first, deciding between a CSP representation and a SAT representation; then choosing different SAT encodings; finally choosing which solver to employ.

He also makes an interesting observation that many modern SAT solvers involve randomness and therefore it is possible that when running the same solver on the same problem you can get wildly different results - this is something to consider when training.

In-solver ML

Earlier we saw that modern CDCL SAT solvers learn new clauses based on conflict diagnosis. These learnt clauses can soon become so numerous that they slow down the operation, so they are regularly culled. Mate Soos [26] uses machine learning to identify which learnt clauses are least likely to be used again later on. These can then be discarded. His clause-selection algorithm is implemented as a C++ library which can then be called by the SAT-solver. This approach shows promise, especially as the training data they used can be greatly expanded.

0.5 Possible Direction for PhD

Automating choices

The current plan is to attempt to automate some of the choices made in Savile Row. As mentioned previously, in [9] Savile Row is able to apply some reformulations which can in some cases dramatically reduce the size of the resulting encoding and lead to big performance gains. However, for some problem instances the reformulation does not lead to performance improvements and can indeed slow down the process. Currently, these reformulations are manually switched on or off. Savile Row also allows the user to make other choices, such as which SAT encodings to use for certain constraints. The goal is to use ML to automate these switches (or at least to make suggestions), so that a prediction is made as to which combination of reformulations and encodings will give the best performance.

Objective significance of encoding

Every paper which introduces a new SAT encoding tends to show off improved solving prowess often on a range of benchmark problems. Without casting aspersions on the authors, it's difficult to say in general whether that particular encoding is "the best" – it may be that different encodings are more successful for different problem instances. So I would like to investigate this effect thoroughly, across a wide range of problems and specifically to look at how combinations of constraints in problems are associated with the performance of different encodings.

Bibliography

[1] H. Cambazard, E. Hebrard, B. O'Sullivan, and A. Papadopoulos, "Local search and constraint programming for the post enrolment-based course timetabling problem," vol. 194, no. 1, pp. 111–135.

[2] C. Anstegui, M. Bofill, J. Coll, N. Dang, J. L. Esteban, I. Miguel, P. Nightingale, A. Z. Salamon, J. Suy, and M. Villaret, "Automatic detection of at-most-one and exactly-one relations for improved SAT encodings of pseudo-boolean constraints," in *International Conference on Principles and Practice of Constraint Programming*, pp. 20–36, Springer.

[3] M. Järvisalo, "Boolean Satisfiability and Beyond: Algorithms, Analysis and AI Applications," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pp. 4066–4069, AAAI Press.

[4] J. Berg, M. Järvisalo, and B. Malone, "Learning optimal bounded treewidth Bayesian networks via maximum satisfiability," pp. 86–95.

[5] M. Hořeňovský, "Performance analysis of a master-key system solver."

[6] K. Apt, *Principles of Constraint Programming.* Cambridge University Press.

[7] P. Nightingale, Akgün, I. P. Gent, C. Jefferson, and I. Miguel, "Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination," in *International Conference on Principles and Practice of Constraint Programming*, pp. 590–605, Springer, Springer, 2014.

[8] P. Nightingale and A. Rendl, "Essence' Description," vol. abs/1601.02865.

[9] P. Nightingale, Akgün, I. P. Gent, C. Jefferson, I. Miguel, and P. Spracklen, "Automatically improving constraint models in Savile Row," vol. 251, pp. 35–61.

[10] M. J. Heule, M. J. Järvisalo, M. Suda, *et al.*, *Proceedings of SAT Competition 2018.* Department of Computer Science, University of Helsinki.

[11] M. J. H. Heule, M. Järvisalo, and M. Suda, "Proceedings of SAT Race 2019 : Solver and Benchmark Descriptions," vol. B-2019-1.

[12] M. Davis and H. Putnam, "A computing procedure for quantification theory," vol. 7, no. 3, pp. 201–215.

[13] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," vol. 5, no. 7, pp. 394–397.

[14] J. M. Silva and K. Sakallah, "GRASP-A new search algorithm for satisfiability," in *Proceedings of International Conference on Computer Aided Design*, pp. 220–227, IEEE.

[15] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Leningrad Seminar on Mathematical Logic*, pp. 115–125.

[16] D. Kroening and O. Strichman, *Decision Procedures.* Springer.

[17] N. Beldiceanu, S. Demassey, M. Carlsson, and J.-X. Rampon, "Global Constraint Catalog."

[18] A. M. Frisch and P. A. Giannaros, "Sat encodings of the at-most-k constraint. some old, some new, some fast, some slow," in *Proc. of the Tenth Int. Workshop of Constraint Modelling and Reformulation*, p. 36.

[19] M. Bofill, J. Coll, J. Suy, and M. Villaret, "SAT encodings of pseudo-boolean constraints with at-most-one relations," in *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pp. 112–128, Springer.

[20] O. Bailleux and Y. Boufkhad, "Efficient CNF encoding of boolean cardinality constraints," in *Principles and Practice of Constraint Programming – CP 2003* (F. Rossi, ed.), pp. 108–122, Springer Berlin Heidelberg.

[21] S. Joshi, R. Martins, and V. Manquinho, "Generalized totalizer encoding for pseudo-boolean constraints," in *Principles and Practice of Constraint Programming* (G. Pesant, ed.), pp. 200–209, Springer International Publishing.

[22] O. Bailleux, Y. Boufkhad, and O. Roussel, "New encodings of pseudo-boolean constraints into CNF," in *Theory and Applications of Satisfiability Testing - SAT 2009* (O. Kullmann, ed.), pp. 181–194, Springer Berlin Heidelberg.

[23] L. Kotthoff, "Algorithm selection for combinatorial search problems: A survey," in *Data Mining and Constraint Programming: Foundations of a Cross-Disciplinary Approach* (C. Bessiere, L. De Raedt, L. Kotthoff, S. Nijssen, B. O'Sullivan, and D. Pedreschi, eds.), pp. 149–190, Springer International Publishing.

[24] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based algorithm selection for SAT," vol. 32, pp. 565–606.

[25] B. Hurley, "Exploiting machine learning for combinatorial problem solving and optimisation."

[26] M. Soos, R. Kulkarni, and K. S. Meel, "CrystalBall: Gazing in the black box of SAT solving," in *International Conference on Theory and Applications of Satisfiability*, vol. 22, pp. 371–387.