Solving Optimization Problems using Satisfiability Solvers

Hendrik Bierlee[1,2] Peter J. Stuckey[1,2] Jip J. Dekker[1,2] Guido Tack[1,2]

Monash University, Department of Data Science and AI

{hendrik.bierlee,peter.stuckey,jip.dekker,guido.tack}@monash.edu

ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA)

Abstract

Over the past twenty years, **Boolean satisfiability (SAT) solvers** have dramatically improved their performance on solving large, hard Boolean formulas. However, most interesting problems are *not* purely Boolean, but instead involve **integer variables and complex constraints**. Such non-Boolean problems need to be reformulated as a Boolean formula before they can be solved by SAT solvers. There are many ways to **encode** the same problem, and the choices made can impact solve time even more than the strength of the SAT solver. In other words, even a good SAT solver cannot solve a bad encoding.

In my PhD project, we advance the state of the art in SAT encoding along the following key objectives:

- Design and evaluate new SAT encoding techniques
- Implement and release the **MiniZinc-SAT** library
- Formalize an integer-based encoding specification language

Coupling Different Integer Encodings for SAT [1]

In **scheduling** problems, the aim is to find for a set of n tasks, the starting time x_i of each task i with duration d_i. For a weighted earliness/tardiness objective, each task also has a deadline and priority (low or high) which determine its initial domain $D(x_i)$ (i.e., the set of all possible starting times). For low-priority tasks, $D(x_i)$ is large, and for high-priority it is small.

To use a SAT solver to solve this problem, the multiple values in $D(x_i)$ are represented by multiple Boolean variables $b_{i,0}, b_{i,1}, \ldots$. The SAT solver will assign each $b_{i,j}$ to true or false. The interpretation of each $b_{i,j}$ w.r.t x_i depends on the chosen **variable encoding**:

- $x_i{:}\mathbb{O}$, where $b_{i,5} \Leftrightarrow$ task x_i starts at time 5 or later
- $x_i{:}\mathbb{B}$, where $b_{i,5} \Leftrightarrow$ task x_i starts at time 2^5 or later

Encoding $x_i{:}\mathbb{O}$ has stronger theoretical solving properties but requires $O(|D(x_i)|)$ variables, while $x_i{:}\mathbb{B}$ only requires $O(\log_2 |D(x_i)|)$. Generally speaking, $x_i{:}\mathbb{O}$ is better for small domains and $x_i{:}\mathbb{B}$ is better for large.

Since tasks vary in domain size, a natural idea is to use *both* variable encodings. Figure 1 shows this indeed improves solver performance.

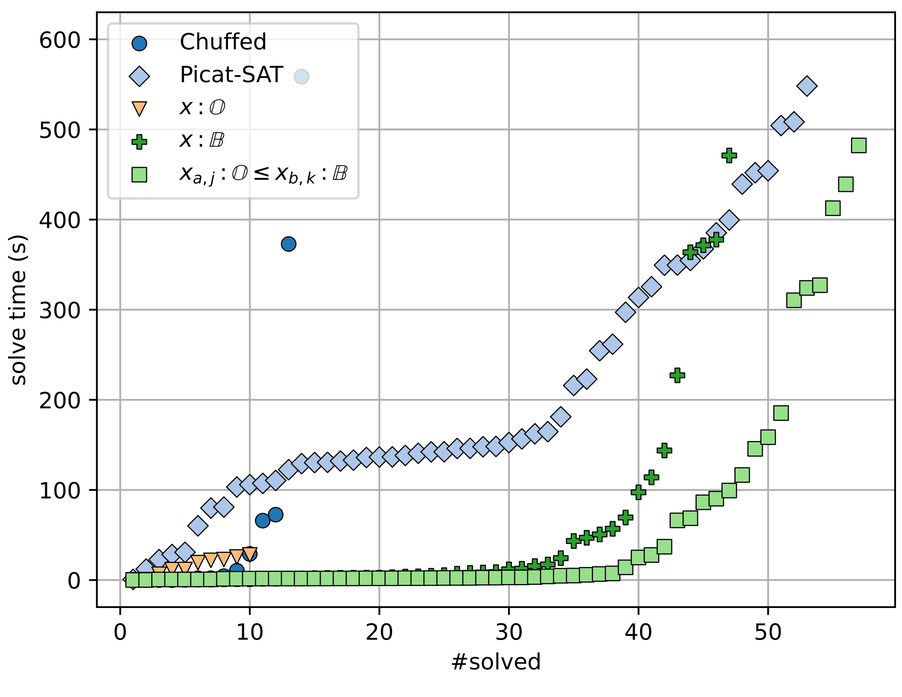

Figure 1:Cactus plot of the solve times of various SAT encodings

The MiniZinc-SAT library

The main challenge of using both variable encodings is that the variable encoding changes the **constraint encoding**. For example, the precedence constraint should ensure that task j starts only after task i has finished. Specialized constraint encodings are needed to account for every combination of variable encoding of x_i and x_j. The mixed case (for low-priority task i and high-priority task j) is denoted as $x_i{:}\mathbb{O} + d_i \leq x_j{:}\mathbb{B}$.

MiniZinc is a solver-independent modelling language, which translates a high-level user model for any of its supported target solvers. Our MiniZinc-SAT library allows the user to annotate a variable's encoding. The appropriate constraint encoding is automatically dispatched. In the integer variable array declaration below, we choose between $x_i{:}\mathbb{O}$ and $x_i{:}\mathbb{B}$ based on whether the domain `size` is smaller than the `CUTOFF` value.

```
array[tasks] of var int: x =
  [ let {
      ann: enc = if dom[i] < CUTOFF then O else B endif;
      var dom[i]: x_i::enc;
    } in x_i | i in n ];
```

Decoding existing SAT encodings

The integer-based encoding specification language can also formally describe existing SAT encodings, such as the Generalized Totalizer (GT) encoding of a **pseudo-Boolean** constraint, by decomposition, e.g.:

$$2b_1 + 3b_2 + 5b_3 \leq 6 \qquad b_1, b_2, b_3 \in \{0, 1\}$$
$$x_{1,1} + x_{1,2} + x_{1,3} \leq 6 \qquad x_{1,1} \in \{0, 2\}, x_{1,2} \in \{0, 3\}, x_{1,3} \in \{0, 5\}$$
$$x_{1,1} + x_{1,2} \leq x_{2,1} \wedge x_{1,3} + x_{2,1} \leq 6 \qquad x_{2,1} \in \{0, 2, 3, 5\}$$

If encoded using $x_i{:}\mathbb{O}$, then GT is equivalent to:

$$\bigwedge_{i=1}^{\log n} \bigwedge_{j=1, j \bmod 2 \equiv 1}^{|x_i|} (x_{i,j} + x_{i,j+1} \leq x_{i+1,\frac{j+1}{2}})$$

References

[1] Hendrik Bierlee, Graeme Gange, Guido Tack, Jip J. Dekker, and Peter J. Stuckey. Coupling different integer encodings for SAT. In Pierre Schaus, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 19th International Conference, CPAIOR 2022, Los Angeles, CA, USA, June 20-23, 2022, Proceedings*, volume 13292 of *Lecture Notes in Computer Science*, pages 44–63. Springer, 2022.

Acknowledgements

This research was partially funded by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Project ID IC200100009.