Encodings into SAT

Combinatorial Problem Solving (CPS)

Enric Rodríguez-Carbonell

May 29, 2020
What is an encoding?

- Language of SAT solvers: CNF propositional formulas

- To solve combinatorial problems with SAT solvers, constraints have to be represented in this language

- An **encoding** of a constraint $C$ into SAT is a CNF $F$ that expresses $C$, so that there is a bijection

  \[
  \text{solutions to } C \iff \text{models of } F
  \]
Examples: AMO constraints

- An **AMO constraint** is of the form $x_0 + \ldots + x_{n-1} \leq 1$
  where each $x_i$ is 0-1
  *(At Most One of the variables can be true)*

- **Quadratic encoding.**
  - Variables: the same $x_0, \ldots, x_{n-1}$
  - Clauses: for $0 \leq i < j < n$, $\overline{x_i} \lor \overline{x_j}$
  - Requires $\binom{n}{2} = O(n^2)$ clauses

- Other encodings try to use fewer clauses, at the cost of introducing new variables
Examples: AMO constraints

- **Logarithmic encoding.** Let $m = \lceil \log_2 n \rceil$. Then:
  
  - Variables: the $x_i$ and new variables $y_0, y_1, \ldots, y_{m-1}$
  - Clauses: for $0 \leq i < n$, $0 \leq j < m$
    
    - $\overline{x_i} \lor y_j$ if the $j$-th digit in binary of $i$ is 1
    - $\overline{x_i} \lor \overline{y_j}$ otherwise

  - Requires $O(\log n)$ new variables, $O(n \log n)$ clauses

- **Heule encoding.**

  - If $n \leq 3$, the encoding is the quadratic encoding.
  - If $n \geq 4$, introduce an auxiliary variable $y$ and encode (recursively)
    
    $x_0 + x_1 + y \leq 1$ and $x_2 + \cdots + x_{n-1} + \overline{y} \leq 1$.

  - Requires $O(n)$ new variables, $O(n)$ clauses

- Other encodings exist (see next)
Consistency and Arc-Consistency

- Let us consider an encoding of a constraint $C$ such that there is a correspondence between maps of the variables of $C$ to their domains, and partial assignments of the boolean variables of the encoding.

- The encoding is consistent if whenever $M$ is not compatible with any solution to $C$, unit propagation on the boolean assignment of $M$ leads to a conflict.

- The encoding is arc-consistent if
  - it is consistent, and
  - unit propagation discards arc-inconsistent values (i.e., values without a support)

- These are good properties for encodings: SAT solvers are very good at unit propagation!
Consistency and Arc-Consistency

- In the case of an AMO constraint $x_0 + \ldots + x_{n-1} \leq 1$:

- **Consistency** $\equiv$ if there are two true vars $x_i$ in $M$ or more, then unit propagation should give a conflict

- **Arc-consistency** $\equiv$ Consistency + if there is one true var $x_i$ in $M$, then unit propagation should set all others $x_j$ to false

- The quadratic, logarithmic and Heule encodings are all arc-consistent
Cardinality Constraints

- A cardinality constraint is of the form $x_1 + \ldots + x_n \bowtie k$
  where each $x_i$ is 0-1 and $\bowtie \in \{\leq, <, \geq, >, =\}$
- AMO are a particular case of card. constraints where $k = 1$ and $\bowtie$ is $\leq$
- Without loss of generality we may assume $\bowtie$ is $<$, i.e.,
  $$x_1 + \ldots + x_n < k$$

- Naive encoding.
  - Variables: the same $x_1, \ldots, x_n$
  - Clauses: for all $1 \leq i_1 < i_2 < \ldots < i_k \leq n$,
    $$\overline{x_{i_1}} \lor \overline{x_{i_2}} \lor \ldots \lor \overline{x_{i_k}}$$
  - This is $\binom{n}{k}$ clauses!
Adders

- Again, other encodings try to use fewer clauses, at the cost of introducing new variables

- Adder encoding.

Build an adder circuit by using bit-adders as building blocks:

\[
\begin{align*}
\text{s} \leftrightarrow & \ \text{XOR}(x, y, z) \\
\text{c} \leftrightarrow & \ (x \land y) \lor (x \land z) \lor (y \land z)
\end{align*}
\]

where \(\text{XOR}(x, y, z)\) is short for

\[
(x \land \overline{y} \land \overline{z}) \lor (\overline{x} \land y \land \overline{z}) \lor (\overline{x} \land \overline{y} \land z) \lor (x \land y \land z)
\]
Adders

- Encodings of this kind are not arc-consistent.

- Consider $x + y + z \leq 0$.
  Then unit propagation should propagate $\overline{x}, \overline{y}, \overline{z}$.

- Let us encode the constraint with a full adder

- The encoding is the Tseitin transformation of $\overline{s}, \overline{c}$ and

  \[
  s \leftrightarrow \text{XOR}(x, y, z) \\
  c \leftrightarrow (x \land y) \lor (x \land z) \lor (y \land z)
  \]

  - Note that

  \[
  \overline{s} \rightarrow (\overline{x} \lor y \lor z) \land (x \lor \overline{y} \lor z) \land (x \lor y \lor \overline{z}) \land (\overline{x} \lor \overline{y} \lor \overline{z}) \\
  \overline{c} \rightarrow (\overline{x} \lor \overline{y}) \land (\overline{x} \lor \overline{z}) \land (\overline{y} \lor \overline{z})
  \]

  - Unit propagation cannot propagate anything!
Sorting Networks

- Sorting Network encoding.

Pass $x_1, \ldots, x_n$ as inputs to a circuit that sorts (say, decreasingly) $n$ bits.

Let $y_1, \ldots, y_n$ be the outputs of this circuit.

Then if the constraint to be encoded is

- $\sum_{i=1}^{n} x_i \geq k$, then add clause $y_k$
- $\sum_{i=1}^{n} x_i \leq k$, then add clause $\overline{y_{k+1}}$
- $\sum_{i=1}^{n} x_i = k$, then add clauses $y_k, \overline{y_{k+1}}$
Sorting Networks

- How to build such a sorting circuit?
- A possibility is to implement mergesort
- In what follows: so-called odd-even sorting networks
- The basic block of odd-even sorting networks are 2-comparators
2-comparators

- A 2-comparator is a sorting network of size 2:
  - it has 2 input variables \( x_1 \) and \( x_2 \)
  - it has 2 output variables \( y_1 \) and \( y_2 \)
  - \( y_1 \) is true if and only if at least one of the input variables is true (i.e., it is the maximum or disjunction)
  - \( y_2 \) is true if and only if both two input variables are true (i.e., it is the minimum or conjunction)
2-comparators

- **Clauses:**

\[ x_1 \leftarrow y_2, \quad x_2 \leftarrow y_2, \quad x_1 \lor x_2 \leftarrow y_1, \]
\[ x_1 \rightarrow y_1, \quad x_2 \rightarrow y_1, \quad x_1 \land x_2 \rightarrow y_2 \]

- **Graphical representation:**

\[ \begin{array}{c}
    x_1 \quad x_2 \\
    \quad y_1 \quad y_2
\end{array} \]

- **Some simplifications are possible:**

- For \( \geq \) constraints: top three clauses suffice
- For \( \leq \) constraints: bottom three clauses suffice
- For \( = \) constraints: all clauses needed
2-comparators

- **Clauses:**

\[
\begin{align*}
    x_1 & \leftarrow y_2, & x_2 & \leftarrow y_2, & x_1 \lor x_2 & \leftarrow y_1, \\
    \overline{x}_1 & \leftarrow \overline{y}_1, & \overline{x}_2 & \leftarrow \overline{y}_1, & \overline{x}_1 \lor \overline{x}_2 & \leftarrow \overline{y}_2
\end{align*}
\]

- **Graphical representation:**

\[
\begin{array}{c}
    x_1 \quad y_1 \\
    x_2 \quad y_2
\end{array}
\]

- **Some simplifications are possible:**

- For \( \geq \) constraints: **top three clauses suffice**
- For \( \leq \) constraints: **bottom three clauses suffice**
- For \( = \) constraints: **all clauses needed**
Merge Networks

- From now on we assume that $n$ is a power of two (if not, pad with variables set to false)

- A **merge network** takes as input two ordered sets of variables of size $n$ and produces an ordered output of size $2n$.

- Let $(x_1, \ldots, x_n)$ and $(x'_1, \ldots, x'_n)$ be the inputs. We recursively define a merge network as follows:

- If $n = 1$, a merge network is a 2-comparator:

$$\text{Merge}(x_1; x'_1) := 2\text{-Comp}(x_1, x'_1).$$
For $n > 1$: Let us define

\[
\begin{align*}
(z_1, z_3, \ldots, z_{2n-1}) &= \text{Merge}(x_1, x_3, \ldots, x_{n-1}; x'_1, x'_3, \ldots x'_{n-1}), \\
(z_2, z_4, \ldots, z_{2n}) &= \text{Merge}(x_2, x_4, \ldots, x_n; x'_2, x'_4, \ldots, x'_n), \\
(y_2, y_3) &= \text{2-Comp}(z_2, z_3), \\
(y_4, y_5) &= \text{2-Comp}(z_4, z_5), \\
&\vdots \\
(y_{2n-2}, y_{2n-1}) &= \text{2-Comp}(z_{2n-2}, z_{2n-1})
\end{align*}
\]

Then,

\[
\text{Merge}(x_1, x_2, \ldots, x_n; x'_1, x'_2, \ldots, x'_n) := (z_1, y_2, y_3, \ldots, y_{2n-1}, z_{2n})
\]
Merge Networks

\[ \text{Merge}_{n=2} \]

\[ \text{Merge}_{n=2} \]
Merge Networks

Sketch of the proof of correctness of Merge:

By IH: \{x_1, x_3, \ldots, x_{n-1}, x'_1, x'_3, \ldots, x'_{n-1}\} = \{z_1, z_3, \ldots, z_{2n-1}\}

By IH: \{x_2, x_4, \ldots, x_n, x'_2, x'_4, \ldots, x'_n\} = \{z_2, z_4, \ldots, z_{2n}\}

Hence \{x_1, x_2, \ldots, x_n, x'_1, x'_2, \ldots, x'_n\} = \{z_1, z_2, \ldots, z_{2n}\}

And

\((y_2, y_3) = 2\text{-Comp}(z_2, z_3)\) implies \{y_2, y_3\} = \{z_2, z_3\}

\((y_4, y_5) = 2\text{-Comp}(z_4, z_5)\) implies \{y_4, y_5\} = \{z_4, z_5\}

\ldots

\((y_{2n-2}, y_{2n-1}) = 2\text{-Comp}(z_{2n-2}, z_{2n-1})\) implies \{y_{2n-2}, y_{2n-1}\} = \{z_{2n-2}, z_{2n-1}\}

So \{x_1, x_2, \ldots, x_n, x'_1, x'_2, \ldots, x'_n\} = \{z_1, y_2, y_3, \ldots, y_{2n-2}, y_{2n-1}, z_{2n}\}
Let us prove outputs are sorted decreasingly. For $1 \leq i < n - 1$ let us see:

- $z_{2i} \geq z_{2(i+1)+1}$:
  - Let us see $z_{2(i+1)+1} = 1$ implies $z_{2i} = 1$
  - If $z_{2(i+1)+1} = z_{2i+3} = z_{2(i+2)-1} = 1$ there are $i + 2$ 1’s in odd $x, x'$
  - Let $p$ be the number of 1’s in odd $x$
  - Let $q$ the number of 1’s in odd $x'$
  - Then $p + q = i + 2$
  - As $x, x'$ is ordered decreasingly,
  - there are $p - 1$ 1’s in even $x$, $q - 1$ 1’s in even $x'$
  - So altogether there are $(p - 1) + (q - 1) = p + q - 2 = i$ 1’s in even $x, x'$
  - Hence $z_{2i} = 1$
Let us prove outputs are sorted decreasingly. For $1 \leq i < n - 1$ let us see:

- $z_{2i} \geq z_{2(i+1)+1}$: proved
Merge Networks

Let us prove outputs are sorted decreasingly. For $1 \leq i < n - 1$ let us see:

- $z_{2i} \geq z_{2(i+1)+1}$: proved
- $z_{2i} \geq z_{2(i+1)}$: by IH
Let us prove outputs are sorted decreasingly. For $1 \leq i < n - 1$ let us see:

- $z_{2i} \geq z_{2(i+1)+1}$: proved
- $z_{2i} \geq z_{2(i+1)}$: by IH
- $z_{2i+1} \geq z_{2(i+1)+1}$: by IH
Merge Networks

Let us prove outputs are sorted decreasingly. For $1 \leq i < n - 1$ let us see:

- $z_{2i} \geq z_{2(i+1)+1}$: proved
- $z_{2i} \geq z_{2(i+1)}$: by IH
- $z_{2i+1} \geq z_{2(i+1)+1}$: by IH
- $z_{2i+1} \geq z_{2(i+1)}$: similar to above

So $\min(z_{2i}, z_{2i+1}) \geq \max(z_{2(i+1)}, z_{2(i+1)+1})$

But $y_{2i+1} = \min(z_{2i}, z_{2i+1})$ and $y_{2(i+1)} = \max(z_{2(i+1)}, z_{2(i+1)+1})$

So $y_{2i+1} \geq y_{2(i+1)}$

And $y_{2i} \geq y_{2i+1}$ for being outputs of 2-Comp

Altogether $z_1, y_2, y_3, \ldots, y_{2n-2}, y_{2n-1}, z_{2n}$ is sorted decreasingly
A sorting network of size $n$ takes an input of size $n$ and sorts it (decreasingly).

We can build a sorting network by successively applying merge networks (as in mergesort).

Let $x_1, \ldots, x_n$ be the inputs. We recursively define a sorting network as follows:

If $n = 2$, a sorting network is a 2-comparator:

$$\text{Sorting}(x_1, x_2) := \text{2-Comp}(x_1, x_2)$$
For $n > 2$: Let us define

\[
(z_1, z_2, \ldots, z_{n/2}) = \text{Sorting}(x_1, x_2, \ldots, x_{n/2}),
\]
\[
(z_{n/2+1}, z_{n/2+2}, \ldots, z_n) = \text{Sorting}(x_{n/2+1}, x_{n/2+2}, \ldots, x_n),
\]
\[
(y_1, y_2, \ldots, y_n) = \text{Merge}(z_1, z_2, \ldots, z_{n/2}; z_{n/2+1}, \ldots, z_n)
\]

Then,

\[
\text{Sorting}(x_1, x_2, \ldots, x_n) := (y_1, y_2, \ldots, y_n)
\]
Sorting Networks

\[ SN_{n=4} \rightarrow Merge_{n=4} \]

\[ x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8 \]

\[ z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8 \]

\[ y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8 \]
Sorting Networks

- This encoding of cardinality constraints is arc-consistent
- It uses $O(n \log^2 n)$ new variables and $O(n \log^2 n)$ clauses
- Several improvements are possible:
  - Only the first $k$ outputs suffice: cardinality networks use $O(n \log^2 k)$ vars and clauses
  - No need to assume that $n$ is a power of two: merges can be defined for inputs of different sizes
Bibliography


