2022 Developer Survey is open! Take survey.

Computer Science Stack Exchange is a question and answer site for students, researchers and practitioners of computer science. It only takes a minute to sign up.

Sign up to join this community

Anybody can ask a question

Anybody can answer

The best answers are voted up and rise to the top

Encoding 1-out-of-n constraint for SAT solvers

Asked 8 years, 10 months ago Modified 1 year, 5 months ago Viewed 5k times

31

6

I'm using a SAT solver to encode a problem, and as part of the SAT instance, I have boolean variables $x_1, x_2, \ldots, x_n$ where it is intended that exactly one of these should be true and the rest should be false. (I've sometimes seen this described as a "one-hot" encoding.)

I want to encode the constraint "exactly one out of $x_1, \ldots, x_n$ must be true" in SAT. What is the best way to encode this constraint, to make the SAT solver run as efficiently as possible?

I can see many ways to encode this constraint:

- *Pairwise constraints.* I could add pairwise constraints $\neg x_i \vee \neg x_j$ for all i, j to ensure that at most one x_i is true, and then add $x_1 \vee x_2 \vee \cdots \vee x_n$ to ensure that at least one is true.

  This adds $\Theta(n^2)$ clauses and no extra boolean variables.

- *Binary encoding.* I could introduce $\lg n$ new boolean variables $i_1, i_2, \ldots, i_{\lg n}$ to represent (in binary) an integer i such that $1 \leq i \leq n$ (adding a few boolean constraints to ensure that i is in the desired range). Then, I can add constraints enforcing that x_i is true and that all

(adding a few boolean constraints to ensure that i is in the desired range). Then, I can add constraints enforcing that x_i is true and that all other x_j's are false. In other words, for each j, we add clauses enforcing that $i = j \Leftrightarrow x_j$.

This adds $\Theta(n \lg n)$ clauses and I don't know how many extra boolean variables.

- *Count the number of true values.* I could implement a tree of boolean adder circuits and require that $x_1 + x_2 + \cdots + x_n = 1$, treating each x_i as 0 or 1 instead of false or true, and use the Tseitin transform to convert the circuit to SAT clauses. A tree of half-adders suffices: constrain the carry output of each half-adder to be 0, and constrain the final output of the final half-adder in the tree to be 1. The tree can be chosen to be of any shape (balanced binary tree, or unbalanced, or whatever).

  This can be done in $\Theta(n)$ gates and thus adds $\Theta(n)$ clauses and $\Theta(n)$ new boolean variables.

  A special case of this approach is to introduce boolean variables $y_1, \ldots, y_n$, with the idea that y_i should contain the value of $x_1 \vee x_2 \vee \cdots \vee x_i$. This intent can be enforced by adding the clauses $y_i \vee \neg x_i$, $y_i \vee \neg y_{i-1}$, and $\neg y_i \vee x_i \vee y_{i-1}$ (where we treat y_0 as a synonym for false) for $i = 1, \ldots, n$. Next, we can add the restrictions $\neg y_i \vee \neg x_{i+1}$ for $i = 1, 2, \ldots, n-1$. This is basically equivalent to the Tseitin transform of a half-adder tree, where the tree has a maximally unbalanced shape.

- *Butterfly network.* I could build a butterfly network on n bits, constrain the n-bit input to be $000\cdots01$, constrain the n-bit output to be $x_1 x_2 \cdots x_n$, and treat each 2-bit butterfly gate as an independent gate that either swaps or does not swap its input with the decision of which to do based upon a fresh new boolean variable that is left unconstrained. Then, I can apply the Tseitin transform to convert the circuit to SAT clauses.

  This requires $\Theta(n \lg n)$ gates and thus adds $\Theta(n \lg n)$ clauses and $\Theta(n \lg n)$ new boolean variables.

Are there any other methods I have overlooked? Which one should I use? Has anyone tested this or tried them experimentally, or does anyone have any experience with any of these? Is the number of clauses and/or the number of new boolean variables a good stand-in metric for estimating the impact of this on SAT solver performance, or if not, what metric would you use?

I just noticed that this answer has some references on enforcing cardinality constraints for SAT, i.e., enforcing the constraint that exactly k out of the n variables are true. So, my question comes down to a special case where $k = 1$. Maybe the literature on cardinality constraints will help shed light on my question.

satisfiability sat-solvers applied-theory

Share Cite Improve this question Follow

edited Apr 13, 2017 at 12:48 asked Jul 9, 2013 at 20:56

Most modern SAT solvers support cardinality clauses and other special (non-CNF) constraints out of the box. – Dávid Horváth Mar 22, 2018 at 1:11

3 Answers

Sorted by: Highest score (default)

14

For the special case of k out of n variables true where k = 1, there is commander variable encoding as described in Efficient CNF Encoding for Selecting 1 to N Objects by Klieber and Kwon. Simplified: Divide the variables into small groups and add clauses that cause a commander variable's state to imply that a group of variables is either all false or all-but-one false. Then recursively apply the same algorithm to the commander variables. At the end of the process demand that the exactly one of the handful of final commander variables be true. The result is O(n) new clauses and O(n) new variables.

Given the ubiquity of two-watched-literals in DPLL based solvers, I think the O(n) clause growth is a decisive advantage over encoding schemes that would add more clauses.

Share Cite Improve this answer Follow

answered Jul 10, 2013 at 3:22

Kyle Jones
7,833 2 25 47

3 If "small group" has size two, then this is just binary addition, where "commander" is the result bit, and the carry is asserted to be false. Done recursively, this method is fully general (works for 1 of N) and indeed practically feasible. – d8d0d65b3f7cf42 Apr 23, 2015 at 11:51

6

A paper by Magnus Björk describes two techniques that could be worth trying.

For 1-out-of-n, one can use both one-hot *and* binary encoding simultaneously. Thus, we have $x_1, \ldots, x_n$ as a one-hot encoding, and $y_1, \ldots, y_b$ as a binary encoding, where $b = \lg n$. We can encode the "at least one" constraint easily, in a single clause: $(x_1 \vee \cdots \vee x_n)$. We can also force the two encodings to be consistent with $2 \lg n$ clauses: we simply add $x_i \longrightarrow \neg y_i$ or $x_i \longrightarrow y_i$, according to whether the ith bit

also force the two encodings to be consistent with $2 \lg n$ clauses. We simply add $x_i \longrightarrow y_j$ or $x_i \longrightarrow y_j$, according to whether the jth bit of the binary encoding of i is 0 or 1. Finally, the "at most one" constraint follows automatically. This also allows the rest of the SAT instance to use whichever encoding is more convenient.

For k-out-of-n, one can apply a sorting network to the input $x_1, \ldots, x_n$ to get the sorted output $y_1, \ldots, y_n$, and then add a clause requiring that y_k is true and y_{k+1} is false. There are a number of simple sorting networks that need only $O(n \lg^2 n)$ comparator units. Therefore, we get an encoding that uses $O(n \lg^2 n)$ clauses and temporary variables.

The paper is

Magnus Björk. Successful SAT Encoding Techniques. 25th July 2009.

The following paper has a detailed list of encodings for 1-out-of-n and k-out-of-n, with some experimental evaluation of each of them. The conclusions are not entirely clear (the command encoding looks pretty good in their experiments).

Alan M. Frisch, Paul A. Giannaros. SAT Encodings of the At-Most-k Constraint: Some Old, Some New, Some Fast, Some Slow. ModRef 2010.

Share Cite Improve this answer Follow

edited Jan 6, 2016 at 1:56

answered Jan 5, 2016 at 22:24

D.W. ♦
140k 18 189 403

0

Here is my implementation of Kyle's answer:

```
def sat_1_of_n(vs, names):
    ''' Returns the name of a variable that's true when one of the variables is true.
        Also returns clauses making sure that at most one variable is true. '''

    if len(vs) == 1:
        return vs[0], []

    elif len(vs) == 2:
        a, b = vs
        clauses = []

    elif len(vs) > 2:
        p = len(vs)//2
```

```
        a, clauses1 = sat_1_of_n(vs[:p], names)
        b, clauses2 = sat_1_of_n(vs[p:], names)
        clauses = clauses1 + clauses2

    c = names.setdefault((a, b, 'commander'), len(names)+1)

    # Make c true when "a or b"
    clauses.append(f'-{a} {c}')
    clauses.append(f'-{b} {c}')
    clauses.append(f'{a} {b} -{c}')

    # We also want one of them to be false
    clauses.append(f'-{a} -{b}')

    return c, clauses
```

Here `names` is a dictionary I use for keeping track of the names of my variables.

If you want exactly one to by true, rather than "less than two" as the above code, just add a single extra clause with all the variables.

Share Cite Improve this answer Follow

answered Dec 2, 2020 at 14:30

Thomas Ahle
186 5