Algo may_22

Inputs

1. A *CNF C* : (conjunction of a) set of *clauses*, where each clause is (a disjunction of) a set of *literals*, each literal is either a *positive* or a *negative* of a *variable*, and variables are represented as natural numbers.

No literal/variable is repeated in a clause, and no clause is repeated in the CNF.

2. A set *X* of variables to be existentially quantified: A vector of positive integers, where each number represents a variable.

Outcome

A set of BDDs $\{B_1, B_2, ... B_N\}$ whose conjunction is an over-approximation of $\exists_X C$, i.e., $\wedge_i \{B_i\} \Leftarrow \exists_X C$.

Discussion

A brief summary of the MUST algorithm might prove useful in order to explain the rest of this algorithm. MUST is a tool for online listing of minimal unsatisfiable subsets of clauses from a set of CNF clauses. To keep track of its search space, it maintains a record of "explored sets", which is initially empty. It then uses a heuristic to start with an *unexplored* set of clauses σ called the "seed" and checks for satisfiability. If satisfiable, starts with a fresh seed. If σ is unsatisfiable, it re-tests for satisfiability after dropping each clause one at a time. If there is a clause $c \in \sigma$ such that $\sigma \backslash \{c\}$ is unsatisfiable, then MUST continues the search with $\sigma \backslash \{c\}$ as the seed. If there is no such $c \in \sigma$, then σ is returned as a Minimal Unsatisfiable Set (MUS). At every satisfiability check for any set σ, if σ is satisfiable then all subsets of σ are marked as *explored*, and if σ is unsatisfiable, then all super-sets of σ are marked as *explored*.

For the purpose of our algorithm, the MUST tool needs to be modified to produce MUS-es with two properties:

P1: For every generated MUS μ, there is an assignment of $\overline{X}$ variables in ϕ which allows μ to be preserved in ψ, i.e., $assignment(\mu) \neq \perp$.

To satisfy P1, modify the *exploration set* to disable exploring $\{c_1, c_2\}$ together for all clauses c_1, c_2 in ψ such that there is a variable $x \in \overline{X}$ that satisfies $x \wedge assignment(c_1) = \perp$ and $\neg x \wedge assignment(c_2) = \perp$ (i.e., $assignment(c_1)$ has $\neg x$ and $assignment(c_2)$ has x). This ensures that no two clauses in a seed can result in an inconsistent assignment. Note that this has a potential to blow up into a quadratic. If we had to prioritize, we should focus on disjoint pairs as that would cut out larger portions in the search space.

P2: For every generated MUS μ, $assignment(\mu)$ should satisfy the factor graph solution S_0 .

One way to guarantee P2 would be to put a final check on any discovered MUS μ before reporting it, and if $assignment(\mu)$ doesn't satisfy S_0, then discard μ, and continue searching from a fresh seed. This satisfiability check, of course, is hard, since S_0 is represented as a set of to-be-conjoined BDDs, and therefore may require the application of a SAT solver following a Tseytin transformation.

The satisfiability check with S_0 may also be applied on a starting seed σ, because of the following lemma:

$$(S_0 \wedge assignment(\sigma) \neq \perp) \Rightarrow (\forall \sigma' \subset \sigma, (S_0 \wedge assignment(\sigma') \neq \perp))$$

In other words, if a set σ of clauses is consistent with S_0, then all subsets of σ are consistent with S_0 as well. This property can be utilized during the shrinking process: if a set is consistent with S_0, then no other subset needs to be tested. The converse however is not true: if a set σ is inconsistent with S_0, then subsets of σ cannot be assumed as inconsistent with S_0. Furthermore, if a set σ is found to be inconsistent with S_0, then even though it is known that its super-sets are inconsistent with S_0, there is no benefit of marking them as such, since all super-sets would be marked as *explored* owing to the fact that σ is unsatisfiable.

In this algorithm, we choose to compute an inexact solution for the consistency with S_0, by testing fixed number random assignments. If a satisfying assignment is found then σ is proved to be consistent with S_0, along with all subsets of σ. However, if no satisfying assignment is found, we consider σ to be inconsistent with S_0. This does not affect the correctness of our algorithm, but it does affect how the factors in our factor graph are merged, and hence affects the memory $\Leftrightarrow$ time $\Leftrightarrow$ exactness tradeoff.

We also use a heuristic count of the consistency of clause sets with S_0 to drive the order in which subsets $\sigma \backslash \{c_i\}$ of an unsatisfiable clause $\sigma = \{c_1, c_2, \ldots c_n\}$ are explored. Let S_0 be represented as (a conjunction of) a set of bdds $\{b_1, b_2, \ldots b_m\}$. Then a heuristic estimate of *how consistent* $\sigma \backslash \{c_k\}$ is with $S_0 = \wedge_i \{b_i\}$ is $\prod_i satratio(b_i \wedge assignment(\sigma \backslash \{c_k\}))$ where *satratio* gives the ratio of satisfying assignments of a bdd, and is easily computable. We explore the *most consistent* subset first.

High level algorithm

Let λ be a high level parameter controlling how large we allow the support sets of individual BDDs to be.

Create a factor graph out of *C*. Merge the factors and variables using some weights determined by how many common neighbours they have, while respecting λ. Compute an over-approximation $S_0 \Leftarrow \exists_X C$.

Let ϕ be the clauses in *C* which have *X* literals, and let $\bar{\phi}$ be the rest of the clauses, i.e., the clauses with literals only on $\overline{X}$.

For each clause *c* in ϕ, $\forall_X c$ is the disjunction of the literals in *c* on the variables in $\overline{X}$.

Let ψ be a set of clauses generated by dropping all $\overline{X}$ literals in ϕ. For duplicate clauses, add unique fake variables that would get projected out without harming the final result of the algorithm. For every clause c in ψ let $orig(c)$ be the corresponding clause in ϕ. Let $assignment(c)=\neg(\forall_X orig(c))$ denote the assignments of $\overline{X}$ variables which allows c to be preserved in $\psi(X)$. Similarly, for a set of clauses $S=\{c_1,c_2,..c_N\}$, let $assignment(S)=\wedge_i\{assignment(c_i)\}$ denote the assignments of $\overline{X}$ variables which allows all the clauses in S to be preserved in $\psi(X)$.

Initialize a MUST solver on ψ. For all $x\in\overline{X}$, for all c_1, c_2 in ψ such that $assignment(c_1)$ has $\neg x$ and $assignment(c_2)$ has x, modify the *exploration set* of the solver to disable exploring $\{c_1,c_2\}$ together. Also modify the Clause Set Reduction algorithm to prefer to first drop the clause c from a set σ which maximizes the *chance* of $assignment(\sigma\backslash\{c\})$ satisfying $S_0=\{b_1\wedge b_2\wedge...\wedge b_n\}$, which is defined as:

$$chance(\sigma\backslash\{c\},S_0)=\prod_i chance(\sigma\backslash\{c\},b_i)$$

$$chance(x,b)=satratio(assignment(x)\wedge b)$$

$satratio(b)$ is the ratio of rows in truth table of b which are $\top$

Use the MUST solver to generate an MUC $\mu_1=\{d_1,d_2,...d_M\}$ such that $assignment(\mu_1)$ satisfies S_0. (To check for satisfiability, use a Tseytin transformation on S_0.) Undo the grouping of factors and variable in the factor graph. Add $\neg assignment(\mu_1)$ as a factor to the factor graph. Increase the grouping weights of $(orig(d_i),orig(d_j))$ for all $d_i,d_j\in\mu_1$. Then re-group and compute the factor graph approximation S_1.

Re-iterate the above algorithm to generate μ_{i+1} using S_i and then generate S_{i+1} using μ_{i+1}. Keep doing this until you either run out of MUCs or the program time limit.