universität freiburg

SAT Solving 2024

Beyond SAT

Bernhard Gstrein, Armin Biere

June 27, 2024

Overview

- MaxSAT, #SAT, Model enumeration, QBF, DQBF, CP
- Local search solvers
- Incremental SAT solving
- Projects/Theses

Local Search SAT Solvers

- Try to find a satisfying solution by local search
- Cannot prove UNSAT
- Are good for solving random benchmarks
  - But those are not considered interesting
- Used to find variable phases in stable mode
  - Stable mode: less restarts, better for SAT
  - Focused mode: more restarts, better for UNSAT
  - More on phase saving and target phases here

How to Local Search Solve

- Start with global assignment, e.g., random, then compute how many unsat clauses there are
- Put unsat clauses in working stack
- As long as this working stack is not empty
    - Pick one clause according to heuristic
    - Flip literals in that clause
- Which literal to flip?
    - One which would not break many other clauses
    - Sample on break value

State of the Art as of Today

- YalSAT is used in Kissat for rephasing

Local Search Using Continuous Energy Models

- Use fuzzy logic: literals $l_i \in [-1, 1]$ instead of $\{0, 1\}$
- Energy function: $E(\mathbf{l}) = \sum_{c \in C} \left(\prod_{i \in c} \frac{1-l_i}{2} \right)$
  - C: set of clauses
  - $\mathbf{l}$: continuous assignment
- Minimize energy using ODE: $\frac{d\mathbf{l}}{dt} = -\nabla E(\mathbf{l})$
- Algorithm:
  1. Initialize $\mathbf{l}$ (according to some heuristic)
  2. Solve ODE numerically until convergence
  3. Transform back: $x_i = \text{sign}(l_i) > 0$
- Leverage ODE solvers from and GPU computation
- Continuous optimization, possibly parallel computation

[Project/Thesis] Write an Energy-Based Local Search Solver

- Implement a local search solver based on continuous energy models
- Use ODE solvers and possibly GPU computation
- Compare performance to state-of-the-art local search solvers
- Reference implementation

MaxSAT

- Goal: find an assignment that maximizes the number of satisfied clauses in a CNF formula
- Weighed MaxSAT: maximize the sum of weights of satisfied clauses
- Partial MaxSAT
  - **Hard Clauses:** Clauses that must be satisfied
  - **Soft Clauses:** Clauses that we wish to satisfy but are not strictly necessary
- Weighed Partial MaxSAT: soft clauses are weighed

Solving MaxSAT

- Iteratively call a SAT solver on a modified version of the original formula
- In each iteration, try to satisfy as many clauses as possible
- If it fails, identify a subset of unsat clauses that are most likely to be the cause of unsat
  - This subset is called the ”unsat core”
- Add constraints to the formula that exclude the core and tries again
- Repeated until a satisfying assignment is found or no more cores can be identified

Example Weighed Partial MaxSAT Formula

```
c This is a comment
c Example 1...another comment
h 1 2 3 4 0
1 -3 -5 6 7 0
6 -1 -2 0
4 1 6 -7 0
```

[Project/Thesis] MaxSAT: Lookup-Table Network

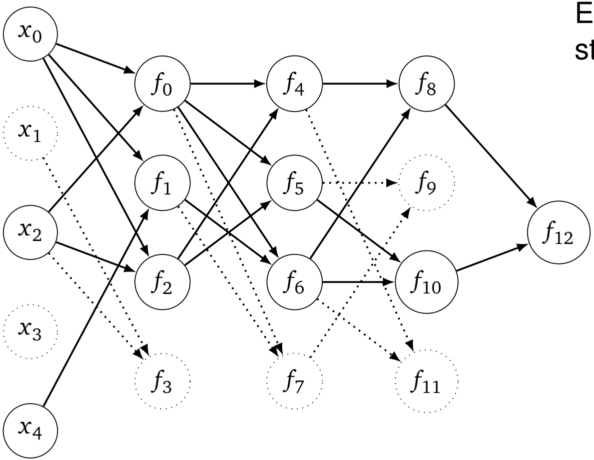
- Given: truth table of arity n, and consequently 2^n entries
  - $2^8 = 256$
  - $2^{16} = 65536$
  - $2^{784} \approx 10^{236}$, all but 60000 are don't-cares
- Given: skeleton of lookup-table (LUT) network
- Task: adjust LUT parameters such that LUT network output matches truth table

[Project/Thesis] MaxSAT: Lookup-Table Network

```
input ->  o
           \
            o <- lookup-table
           / \
input ->  o   o
           \ / \
            o   o
           / \ / \
input ->  o   o   o <- output
           \ / \ /
            o   o
           / \ /
input ->  o   o
           \ /
            o
           /
input ->  o
```

```
0 0 0 0 0  0
0 0 0 0 1  0
0 0 0 1 0  0
0 0 0 1 1  0
0 0 1 0 0  1
0 0 1 0 1  0
0 0 1 1 0  1
0 0 1 1 1  0
0 1 0 0 0  0
0 1 0 0 1  1
0 1 0 1 0  0
0 1 0 1 1  1
0 1 1 0 0  1
0 1 1 0 1  0
0 1 1 1 0  1
...
```

[Project/Thesis] Project LUT Network



Exact logic synthesis using SAT solvers is still an open question

[Project/Thesis] Predicting Optimal MaxSAT Encoding MaxSAT Using ML

- Goal: Develop a Machine Learning (ML) model to predict the best encoding for a MaxSAT solver based on WCNF features
- Importance: Proper feature extraction from WCNFs is crucial for the success of the ML model
  - Key paper: Feature Extraction for CNFs

[Project/Thesis] Predicting Optimal MaxSAT Encoding MaxSAT Using ML

1. **Feature Extraction**
    - Identify extractable features from WCNFs (literature review)
    - Implement feature extraction in a **compiled language**
2. **Benchmarking**
    - Run the solver on various benchmarks with different encodings to gather time data (staff task)
3. **ML Model Development**
    - Experiment with different ML models in Python to find the best performer
    - Integrate the best model into the MaxSAT solver
4. **Model Integration**
    - Use C++ bindings of PyTorch or ggml library for ML backend integration if needed

#SAT (SharpSAT)

- Goal: determine the number of satisfying assignments for a given Boolean formula
- While SAT is interested in finding a single solution (or knowing if one exists), #SAT wants to **count all** possible solutions
- Significantly harder than SAT
  - Exact counting: couple of hundred variables
  - Approximate counting: around 1000 variables

How to #SAT

- Knowledge Compilation
  - Transform input formula into a tractable form, such as d-DNNF (deterministic Decomposable Negation Normal Form)
  - From this form, counting the number of satisfying assignments becomes efficient
- DPLL-style Exhaustive Search
  - Systematically explore all possible assignments to variables
  - Pruning techniques like unit propagation and pure literal elimination to reduce search space
- Approximate Techniques
  - Randomized Algorithms: Use probabilistic methods to estimate the count
  - Markov Chain Monte Carlo (MCMC): Sample from the space of satisfying assignments to approximate the count
  - Hashing-based Methods: Use universal hashing to partition the solution space and count within each partition

#SAT Use Case: Probabilistic reasoning

Paper: *Characterization of Possibly Detected Faults by Accurately Computing their Detection Probability*

- Testing crucial for complex Very Large Scale Integration (VLSI) devices
- Commercial Automatic Test Pattern Generation (ATPG) tools struggle with faults involving unspecified input values
- Possibly detected faults may be over- or underestimated
- SAT-based algorithm computes the detection probability for faults marked as possibly detected

#SAT Use Case: Critical Infrastructure Reliability

Paper: *A Weighted Model Counting Approach for Critical Infrastructure Reliability*

- Model counting method for estimating network reliability

#SAT Extensions

- Weighed model counting
  - Each assignment is associated with a weight
  - Sum the weights of all satisfying assignments
- Projected model counting (#∃SAT)
  - Count assignments of a subset of variables that can be extended to a satisfying assignment of entire formula
  - Existentially quantify irrelevant variables
- Projected weighed model counting
  - Combination of the above

Model Enumeration

- Goal: finding all models (or satisfying assignments) for a given Boolean formula
- Example Problem Statement: Given a CNF formula, enumerate all satisfying assignments
- Very similar to #SAT, but the implementation will differ

Quantified Boolean Formula (QBF)

- Extension of SAT with universal and existential quantification of variables
- Key Concepts:
  - **Existential Quantification:** There exists an assignment for the variable that makes the formula true
  - **Universal Quantification:** For all assignments to the variable, the formula is true
- Theoretical complexity: PSPACE
- Notation is exponentially more succinct than SAT, but therefore the problems harder
- Example of QBF: $\forall x_1 \exists x_2 \forall x_3 \exists x_4 (x_1 \lor \neg x_2) \land (x_3 \lor x_4)$

QBF Example Formula

$\forall x_1 \exists x_2 \forall x_3 \exists x_4 (x_1 \vee \neg x_2) \wedge (x_3 \vee x_4)$

```
p cnf 4 2
a 1
e 2
a 3
e 4
1 -2 0
3 4 0
```

QBF Solving

- Several decision procedures for QBFs have been proposed and implemented
- Mostly based either on search or on variable elimination, or on a combination of the two

QBF Solving: Search

- QBF φ
- Left-most variable z
- Simplify φ to φ_z and (or, respectively) $\varphi_{\overline{z}}$ recursively
- Until either an empty clause (conflict) or the empty set of clauses (sat) are produced

QBF Solving: Variable Elimination

- Eliminate variables till the formula contains the empty clause or becomes empty
- For any QBF φ, $\exists x \varphi$ and $\forall y \varphi$ are logically equivalent to $(\varphi_x \vee \varphi_{\overline{x}})$ and $(\varphi_y \wedge \varphi_{\overline{y}})$
- Main problem: at each step, the formula can double its size
  - There are, however, several ways to address this

QBF Use Cases: Bounded Model Checking

- *A Survey on Applications of Quantified Boolean Formulas*
- Check if a system can reach a bad state within k steps
- Transition relation R^i
- Initial state I
- Bad state B

SAT encoding:

$$\bigvee_{i=0}^{k-1} (I \wedge R^i \wedge B)$$

QBF encoding:

$$\exists i \in [0, k-1] \; : \; (I \wedge R^i \wedge B)$$

Non-CNF Example of QBF

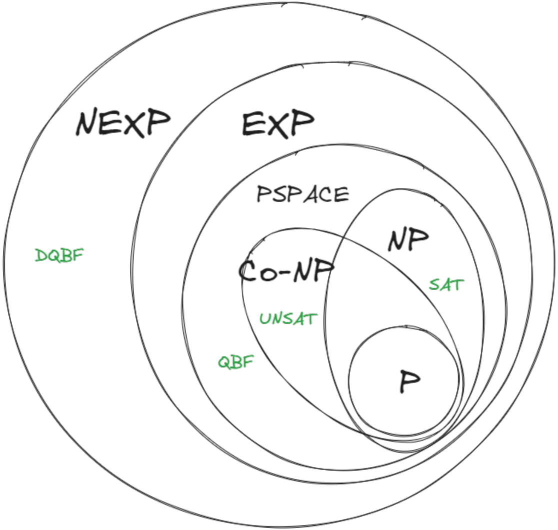
- We do not necessarily have to use CNF for QBF
- Solutions and counter-examples should be in the same format
  - But if you negate a CNF, it is not a CNF anymore
- See qpro

$$\forall a_2 \exists e_3 e_4 (e_4 \forall a_5 a_6 ((a_5 \wedge \neg a_6) \vee \exists e_7 e_8 e_9 e_{10} (e_7 \wedge e_8 \wedge e_9 \wedge e_{10})))$$

Dependent Quantified Boolean Formulas (DQBF)

- Generalization of QBF that includes dependencies among the quantified variables
- **Dependencies:** A variable's quantification can be dependent on other variables
- Theoretical Complexity: NEXPTIME (even more complex than PSPACE)
- An example of dQBF: $\forall x_1 \exists x_2(x_1) \forall x_3(x_1, x_2) \exists x_4(x_1, x_2, x_3)(x_1 \vee \neg x_2) \wedge (x_3 \vee x_4)$
  - Here, x_2 depends on x_1, x_3 depends on x_1 and x_2, and so on

Complexities



DQBF Use Cases

Dependency Quantified Boolean Formulas: An Overview of Solution Methods and Applications

Controller Synthesis

- Vector of present state bits $\mathbf{s}$, vector of next state bits $\mathbf{s}'$
- Uncontrollable primary inputs $\mathbf{x}$
- Controllable inputs $\mathbf{c}(\mathbf{s}, \mathbf{x})$
- Transition function $\Lambda(\mathbf{s}, \mathbf{x}, \mathbf{c})$
- Invariant properties $\mathrm{inv}(\mathbf{s}, \mathbf{x})$, must hold in any case
- Is there an implementation of the controller such that the resulting sequential circuit satisfies the invariant $\mathrm{inv}(\mathbf{s}, \mathbf{x})$?

$$\forall \boldsymbol{s}\, \forall \boldsymbol{s}' \forall \boldsymbol{x}\, \exists w(\boldsymbol{s})\; \exists w'(\boldsymbol{s}') \exists \boldsymbol{c}(\boldsymbol{s}, \boldsymbol{x}) :$$

$$\big(\mathrm{init}(\boldsymbol{s}) \Rightarrow w\big) \wedge \big(w \Rightarrow \mathrm{inv}(\boldsymbol{s}, \boldsymbol{x})\big) \wedge \big(\boldsymbol{s} \equiv \boldsymbol{s}' \Rightarrow w \equiv w'\big) \wedge$$

$$\Big(\big(w \wedge (\boldsymbol{s}' \equiv \boldsymbol{\Lambda}(\boldsymbol{s}, \boldsymbol{x}, \boldsymbol{c}))\big) \Rightarrow w'\Big)$$

Constraint Programming (CP)

- Express problems as variables and constraints
- Each variable has a domain of possible values
- Constraints define relationships between variables and restrict their possible values
- Constraint solvers aim to find values that satisfy all constraints through search and propagation
- Declarative, with the solver determining the solution process

Google OR-Tools

- Website
- Solving a CP Problem
- Knapsack Problem
- Google OR-Tools Github: SAT Solver

Why use CP and not SAT?!

- In CP, variables can be complex types (integers, sets, tuples)
- Constraints can be complex and flexible
- Interactive solving: add and remove constraints on the go
- In constrast, solving a problem via SAT requires low-level CNF
- Allegory: programming in assembly vs. C
    - SAT: assembly
    - CP: C

Pseudo-Boolean Solving

Taken from Pseudo-Boolean Solving Tutorial

- Pseudo-Boolean function: $f : \{0,1\}^n \rightarrow \mathbb{R}$
- Pseudo-Boolean constraint: $\sum_i a_i l_i \bowtie A$
- $\bowtie \in \{\geq, \leq, =, >, <, \}$
- $a_i, A \in \mathbb{Z}$
- Literals l_i: x_i or $\overline{x}_i$ (where $x_i + \overline{x}_i = 1$)
- Variables x_i take values $0 =$ false or $1 =$ true
- Example: $x_1 + x_2 + x_3 \geq 3$

How to PB Solving

- Conversion to disjunctive clauses
  - Lazy approach: learn clauses from PB constraints
  - Eager approach: re-encode to clauses and run CDCL
- Native reasoning with pseudo-Boolean constraints
  - RoundingSAT
- More in Pseudo-Boolean Solving Tutorial

Incremental SAT Solving

From *Practical SAT Solving*

- We often need to solve a sequence of similar SAT instances
  - for example planning as sat, sokoban, bounded model checking
  - the instances share most of the clauses with their neighbors
- Can we solve these sequences of instances more efficiently?
- What is incremental SAT solving?
  - Clauses can be added to and removed from the SAT solver
- Why not call the solver with the new formula every time?
  - The solver can remember learned clauses and other stuff (variable scores required for heuristics)
  - (de)initialization overheads removed

Incremental SAT Solving: Example

- System with three switches A, B, C which can either be ON or OFF

$$A \vee B \vee C$$

- New requirement: B cannot be ON at the same time as C
- Add:

$$\neg(B \wedge C)$$

The End
That’s it folks!

I wish you could learn something useful!

Bernhard Gstrein
`gstrein@cs.uni-freiburg.de`
+49 761 203 8147

Armin Biere
`biere@cs.uni-freiburg.de`
+49 761 203 8148