Logic Solvers and Machine Learning
The Next Frontier

Vijay Ganesh
University of Waterloo, Canada

Computational Math Colloquium, Canada
Thursday Oct 29th, 2020
PART I

Context and Motivation

Why should you care about SAT and SMT Solvers?
Software Engineering and SAT/SMT Solvers
An Indispensable Tool for any Strategy
Program Specification

Program Reasoning Tool

Logic Formulas

SAT/SMT Solver

SAT/UNSAT

Program is correct?
or Generate Counterexamples (test cases)
SAT/SMT Solver Research Story
A 1000x+ Improvement

- Solver-based programming languages
- Compiler optimizations using solvers
- Solver-based debuggers
- Solver-based type systems
- Solver-based concurrency bug-finding
- Solver-based synthesis

- Concolic Testing
- Program Analysis
- Equivalence Checking
- Auto Configuration

- Bounded MC
- Program Analysis
- AI

1,000,000 Constraints
100,000 Constraints
10,000 Constraints
1,000 Constraints

Important Contributions
Solvers, Applications, and Other Work

Other important contributions
• Symbolic execution [CG+06]
• ML-based fuzzing [SMG20]
• Defense mechanisms for ML [PNG20]
• Attack-resistance [OBDBG17]
• Proof complexity of solvers [RKG18]
• Decidability and complexity results for theories over strings [GM+12]
PART II

Research Questions
The Problem
Why are Solvers Efficient?

- Why are SAT solvers efficient for many applications? After all, Boolean SAT is NP-complete, believed to be intractable. Open for over two decades. This question is very general, i.e., it applies to all formal reasoning systems, e.g., SMT solvers, first-order provers, QBF, MaxSAT, ILP, CSP, model checkers, … This question can be broken into the following sub-questions:

  - How do we best capture the essence of solvers via a simple yet powerful scientific design principle/abstraction?

  - Can such an abstraction also enable us to prove (parameterized) complexity-theoretic upper and lower bounds?
RQ1: What is a powerful solver abstraction that explains their efficiency and forms the foundation of an accompanying scientific design principle that simplifies and clarifies solver design?

1) Solver = proof system + ML for sequencing/selecting/initialization proof rules. Solvers are data-rich, ideal for ML
2) Led to the design of award-winning MapleSAT (and variants). 5 different ML-based heuristics (branching, splitting, restarts, initialization, and algorithm selection). Our work has been widely adopted, adapted, extended

RQ2: Can we express this understanding of solver efficiency in (parameterized) complexity-theoretic terms?

1) Extensive empirical work in understanding structure of real-world instances. Candidate parameter ‘merge’
2) Polynomial equivalence of SAT solvers and merge resolution, power of restarts, among many other theorems

RQ3: Can we leverage this understanding of the power of SAT solvers to extend them to novel applications?

1) Logic guided machine learning (LGML) and LogicGAN
2) CDCL(crypto)
PART III

SAT Solver Background
The Boolean Satisfiability (SAT) Problem

Basic Definitions

• The Boolean SAT problem: Given Boolean formulas in Conjunctive Normal Form (CNF), decide whether they are satisfiable. A SAT solver is a program that takes as input CNF formulas, and decides whether they are satisfiable.

\[(x_1 \lor \neg x_2 \ldots \lor \neg x_n)\]
\[(\neg x_1 \lor \neg x_2 \ldots \lor x_n)\]
\[\ldots\]
\[(\neg x_1 \lor x_2 \ldots \lor \neg x_n)\]

• The SAT problem is known to be NP-complete, believed to be intractable.

• SAT solvers are required to produce proofs of unsatisfiability for UNSAT instances and satisfying assignments for SAT instances.
Modern Conflict-Driven Clause-Learning (CDCL) SAT Solver
Overview

Input SAT Instance

- Propagate() (BCP)
  - Conflict?
    - All Vars Assigned?
      - Conflict Analysis()
        - Top-level Conflict?
          - Return SAT
          - Branch()
            - Learnt clause (x)
            - Learnt clause (neg(z) OR y)
          - Return UNSAT
          - Backjump()

GRASP Solver: [MS96]
ZChaff Solver: [MMZZM01]
PART IV

Contribution 1
Solver Design via Proof Systems + Machine Learning
What is a Branching Heuristic?
Prior Work

Question: What is a variable selection (branching) heuristic?

- A “dynamic” ranking function that ranks variables in an input formula in descending order
- Re-ranks variables of input formula at regular intervals throughout the run of a SAT solver
- This understanding of the famous VSIDS branching heuristic was unsatisfactory

Our experiments and results: [LG+15, LGPC16, LGPC+16, LGPC17, LGPC18]

- We studied 7 of the most well-known branching heuristics in detail
- Viewed branching as prediction engines that attempt to maximize global learning rate
- In turn led us to devise new ML-based branching, LRB, that for the first time matched VSIDS
CDCL with Deductive Feedback Loop

Reinforcement Learning

Partial Assignment

Agent

Decision Heuristic (ML)

Learnt Clause

Environment

Clause Learning (Proof System)
What is an Optimal Branching Sequence?
Defining a Good Objective/Reward

Global learning rate (GLR) = \frac{\# \text{conflicts}}{\# \text{decisions}}

# of lemmas
# of “cases”
**MULTI-ARMED BANDIT PROBLEM**

Sample average =

\[
\frac{1}{3} \times 4 + \frac{1}{3} \times 3 + \frac{1}{3} \times 1
\]

Exponential moving average =

\[
(1 - \alpha)^2 \times 4 + \alpha \times 3 + (1 - \alpha)^0 \times 1
\]
Connecting MAB and the Branching Problem
Applying Reinforcement Learning to Branching

Gambler/Agent

Reward
Time

\[ x_1 \]

\[ x_n \]

<table>
<thead>
<tr>
<th>Reward</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branching Heuristic

\[ \text{Reward} \]
\[ \text{Time} \]
LEARNING RATE EXAMPLE

\[
\text{sampled\_learning\_rate}(A) = \frac{2}{3}
\]

\[
\text{sampled\_learning\_rate}(B) = 0/3
\]
LEARNING-RATE BRANCHING (LRB) EXAMPLE

sampled_learning_rate(A) = \frac{2}{3}

\text{exponential moving average} = (1 - \alpha)^1 \times \frac{2}{3} + (1 - \alpha)^0 \times \frac{1}{3}

“Rewards”
Activity(A)
The bump is a constant
Every time a variable appears in a conflict analysis, its activity is additively bumped by a constant.

Exponential Moving Average (EMA)
performed for all variables at the same time
After each conflict, the activities of all variables are decayed.

The bump is not constant
Every time a variable appears in a conflict analysis, the numerator of its learning rate reward is incremented. After each conflict, the denominator of each assigned variable’s learning rate reward is incremented.

EMA performed only when variable goes from assigned to unassigned
When a variable is unassigned, the variable receives the learning rate reward, and the estimate Q is updated.
Most importantly, we understand why bumping certain variables and why performing multiplicative decay helps.
APPLE-TO-APPLE RESULTS
(MINISAT WITH VSIDS VS. CHB VS. LRB)
RESULT: GLOBAL LEARNING-RATE

- Global Learning Rate: \( \frac{\text{# of conflicts}}{\text{# of decisions}} \)

- Experimental setup: ran 1200+ application and hand-crafted instances on MapleSAT with VSIDS, CHB, LRB, Berkmin, DLIS, and JW with 5400 sec timeout per instance on StarExec

<table>
<thead>
<tr>
<th>Branching Heuristic</th>
<th>Global Learning Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRB</td>
<td>0.452</td>
</tr>
<tr>
<td>MVSIDS</td>
<td>0.410</td>
</tr>
<tr>
<td>CHB</td>
<td>0.404</td>
</tr>
<tr>
<td>CVSIDS</td>
<td>0.341</td>
</tr>
<tr>
<td>BERKMIN</td>
<td>0.339</td>
</tr>
<tr>
<td>DLIS</td>
<td>0.241</td>
</tr>
<tr>
<td>JW</td>
<td>0.107</td>
</tr>
</tbody>
</table>
Machine Learning for Branching Heuristics

Summary

• Branching heuristics aim to optimize sequencing of unit resolution rule applications. Reinforcement learning is a powerful way to model them. This lesson can be lifted to solvers of all kinds, since proof rule sequencing is a universally relevant optimization problem in automated deduction.

• The metric being optimized in the CDCL SAT context is Global Learning Rate. “Fail fast, Fail often, and learn from it”

Global Learning Rate (GLR) = (# of conflicts)/(# of decisions)

• A good way to maximize GLR is to branch on variables that were involved in ‘recent’ conflicts, until such time that they don’t give any additional conflicts. It is then time to move on.

• The Searchlight Analogy a la Exploitation vs. Exploration (multiplicative decay)
Machine Learning For Solvers

Summary

• Solvers = Proof System + ML-based Optimization Procedures. Very successful design principle. Adopted, adapted, and extended widely by leading solver researchers.

• We have developed the following ML-based heuristics [LG+15, LGPC16, LGPC+16, LGPC17, LGPC18]:
  • Splitting heuristics for parallel SAT solvers
  • Restart policy for SAT solvers
  • Initialization heuristics for sequential SAT solvers
  • Algorithm selection for SMT solvers
  • Tactic selection for SMT solvers

• Researchers have attempted pure ML-based approaches for SAT. They haven’t worked. It seems combining ML with deduction is the way to go.
PART V

Contribution 2

Towards a Complexity-theoretic Understanding of Solvers
Towards Complexity Theory of Solvers

The Questions

**Question 1:** Which proof system exactly characterizes CDCL SAT solvers? (i.e., polynomial equivalence between a proof system and CDCL, thus lifting lower bounds from proof complexity theory setting to solvers)

**Question 2:** Can we establish parameterized upper bound on the runtime of CDCL SAT solver for UNSAT and/or SAT instances? The parameter must be meaningful in theory and practice

  **Question 2.1:** Parameterized upper bound on proof size

  **Question 2.2:** Parameterized upper bound for proof search (automatizability)

**Question 3:** By switching various heuristics ON/OFF, we get a variety of solver configurations and corresponding proof systems. We can pose the above questions for upper and lower bounds for each of these configurations. Similarly, we can extend this program to SMT, QBF, MC,…
Proof Systems
Parameterized Proof-complexity of Solvers

General resolution
The rule is form of modus ponens. Proof is a directed acyclic graph (DAG).

\[
\frac{(x_1 \lor \cdots \lor x_n) \ (\neg x_n \lor y_1 \ldots \lor y_m)}{(x_1 \lor \cdots \lor x_{n-1} \lor y_1 \ldots \lor y_m)}
\]

Merge resolution
Biased towards deriving clauses that share literals. (In the clauses below, \(\alpha\) is a disjunction of literals.)

\[
\frac{(\alpha \lor \cdots \lor x_n) \ (\neg x_n \lor \cdots \lor \alpha)}{(x_1 \lor \cdots \lor x_{n-1})}
\]

Unit resolution
One clause must be unit. Proof is a DAG.

\[
\frac{(x_n) \ (\neg x_n \lor y_1 \ldots \lor y_m)}{(y_1 \lor \cdots \lor y_m)}
\]

Tree resolution
Same rules as general resolution. Proof is a tree. Not allowed to reuse lemmas unlike DAG proofs.
RESOLUTION AND MERGES
MERGEABILITY AND CDCL SAT SOLVERS

\[
\frac{(\alpha \lor x_n) \ (\neg x_n \lor \beta)}{(\alpha \lor \beta)} \quad \frac{(\gamma \lor y_n) \ (\neg y_n \lor \delta)}{(\gamma \lor \delta)}
\]

\[
\ ...
\]

\[
\frac{(x_i \lor x_j) \ (\neg x_j \lor x_i)}{(x_i)} \quad \frac{(\neg x_i \lor x_k) \ (\neg x_k \lor \neg x_i)}{(\neg x_i)}
\]

\[
\frac{(x_i) \ (\neg x_i)}{\bot}
\]
MANY PROPOSED COMPLEXITY-THEORETIC PARAMETERS

Motivation: Industrial instances must have structure that is exploited by SAT solvers

Question: Which metrics have the best explanatory power, across a wide variety of benchmarks?
MERGEABILITY
NUMBER OF MERGES VS. LEARNT CLAUSE SIZE RESULT

Merges vs Average Learned Clause Size

Merges vs Average Learned Clause Size (SAT)

Merges vs Average Learned Clause Size (UNSAT)
MERGEABILITY
NUMBER OF MERGES VS. SOLVER RUNTIME RESULT
Proof Complexity-theoretic Results
- CDCL solvers are \( p \)-equivalent to merge resolution, building on previous result on CDCL \( p \)-equivalent to general resolution

- Every 1-UIP clause requires a merge. Non-merge resolution steps cannot derive empowering clauses (also by [BBCGKV13])

- Separation result for CDCL with and without restarts for SAT instances, long-standing open problem

- \( \text{Res}(T) \) and \( \text{Res}^*(T) \) proof system model for SMT solvers [RKG18]

Relevant Empirical Results
- Number of merges in random as well as industrial formulas correlate well with solver runtime

- Scaling study on random instances: As we increase the number of merges for random \( k \)-SAT instances, solver runtime falls (esp. for UNSAT), learnt clause length decreases (for all instances)
PART VI

Contribution 3

Logic Solvers for Machine Learning
Logic Guided Machine Learning

Results: Using LGML we were able to learn expressions for the Pythagorean theorem and the Sine function from data [SPG20]
PART VII

Conclusions and Takeaway
ML for Solvers and Solvers for ML
Corrective Feedback between ML and Deduction

Agent

Decision Heuristic (ML)

Partial Assignment

Environment

Clause Learning (Proof System)

Learnt Clause
RQ1: What is a powerful solver abstraction that explains their efficiency and forms the foundation of an accompanying scientific design principle that simplifies and clarifies solver design?

1) Solver = proof system + ML for sequencing/selecting_INITIALIZATION proof rules. Solvers are data-rich, ideal for ML
2) Led to the design of award-winning MapleSAT (and variants). 5 different ML-based heuristics (branching, splitting, restarts, initialization, and algorithm selection). Our work has been widely adopted, adapted, extended

RQ2: Can we express this understanding of solver efficiency in (parameterized) complexity-theoretic terms?

1) Extensive empirical work in understanding structure of real-world instances. Candidate parameter ‘merge’
2) Polynomial equivalence of SAT solvers and merge resolution, power of restarts, among many other theorems

RQ3: Can we leverage this understanding of the power of SAT solvers to extend them to novel applications?

1) Logic guided machine learning (LGML) and LogicGAN
2) CDCL(crypto)
Future Work
ML for Solvers and Solvers for ML

1. Extended resolution SAT solvers
   - Use ML to select pairs of variables that appear most often together and replace with new variable

2. Interference-based proof systems and SDCL solvers
   - Use MAB-based method to determine when to add interference-based pruning clause

3. Logic Generative Adversarial Networks (LogicGANs)
   - Use solvers to identify why discriminator rejects, provide corrective feedback to the generator
Future Work
Applications and Complexity of Solvers

1. Applications of solvers to cryptanalysis, a la, CDCL(crypto)
   - We have already developed a solver-based algebraic fault injection attack (AFA) tool
   - Working on a solver-based differential cryptanalysis tool

2. Merge-based upper bound on solver runtime
   - Parameterized automatizability result on solver runtime using merge
   - Parameterized upper-bound on proof size

3. Proof complexity of SMT, MCSAT, … solvers
   - Already characterized SMT solvers as Res(T) and Res*(T) proof systems [RKG18]. Extend such proof complexity-theoretic results to MCSAT, QBF, model checkers,…
## FEW RECENT CONTRIBUTIONS

<table>
<thead>
<tr>
<th>Name</th>
<th>Key Concept</th>
<th>Impact</th>
<th>Pubs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maple Series of SAT Solvers, and understanding SAT</td>
<td>Machine learning for solver heuristics, and community structure</td>
<td>Won SAT Competition 2016</td>
<td>AAAI 2016, SAT 2016/17/18 SAT 2014&lt;sup&gt;3&lt;/sup&gt;</td>
</tr>
<tr>
<td>Z3str3 String and Integer Solver&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Novel techniques for string + integer combination</td>
<td>Analysis of Web Apps (Part of Z3 codebase)</td>
<td>FSE 2013, CAV 2015&lt;sup&gt;3&lt;/sup&gt; FMSD 2017</td>
</tr>
<tr>
<td>MathCheck Conjecture Verifier</td>
<td>CAS+SAT combination a la DPLL(CAS)</td>
<td>Solved problem open since 1944</td>
<td>AAAI 2019, IJCAI 2016, JAR 2017, CADE 2015&lt;sup&gt;3&lt;/sup&gt;</td>
</tr>
<tr>
<td>Undecidability results for theories over strings</td>
<td>Boundary between decidability and undecidability</td>
<td>Solved problems open since early 2000's</td>
<td>HVC 2012, RP2018</td>
</tr>
<tr>
<td>Attack-resistance</td>
<td>A new approach to formally establishing the efficacy of security defenses</td>
<td>Mathematical guarantee of software trustworthiness even in the presence of bugs</td>
<td>Euro S&amp;P 2017 PLAS 2015</td>
</tr>
</tbody>
</table>

1. 100+ research projects use STP, HAMPI, and Z3str2.
2. STP won the SMTCOMP 2006/2010 and second in 2011/2014 competitions for bit-vector solvers
3. Best paper awards/honors at various conferences including SAT, DATE, SPLC, CAV, and CADE
# IMPORTANT PAST CONTRIBUTIONS

<table>
<thead>
<tr>
<th>Name</th>
<th>Key Concept</th>
<th>Impact</th>
<th>Pubs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STP</strong> Bit-vector &amp; Array Solver(^1,2)</td>
<td>Abstraction-refinement for solving</td>
<td>Concolic Testing</td>
<td>CAV 2007</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CCS 2006</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TISSEC 2008</td>
</tr>
<tr>
<td><strong>HAMPI</strong> String Solver(^1)</td>
<td>App-driven bounding for solving</td>
<td>Analysis of Web Apps</td>
<td>ISSTA 2009(^3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TOSEM 2012</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CAV 2011</td>
</tr>
<tr>
<td><strong>Taint-based Fuzzing</strong></td>
<td>Data flow is cheaper than concolic</td>
<td>Scales better than concolic</td>
<td>ICSE 2009</td>
</tr>
<tr>
<td><strong>Automatic Input Rectification</strong></td>
<td>Automatic security envelope</td>
<td>New security approach</td>
<td>ICSE 2012</td>
</tr>
</tbody>
</table>

1. 100+ research projects use STP, HAMPI, and Z3str2.
2. STP won the SMTCOMP 2006/2010 and second in 2011/2014 competitions for bit-vector solvers
3. Best paper awards/honors at various conferences including SAT, DATE, SPLC, CAV, and CADE
ML for Solvers and Solvers for ML
Corrective Feedback between ML and Deduction

Agent

Decision Heuristic (ML)

Partial Assignment

Learnt Clause

Environment

Clause Learning (Proof System)
ACKNOWLEDGEMENTS

• Students
  • Jimmy Liang
  • Ed Zulkoski
  • Joe Scott
  • Ian Li
  • Saeed Nejati
  • Curtis Bright
  • Murphy Berzish

• Collaborators
  • Krzysztof Czamecki
  • Christoph Wintersteiger
  • Ruben Martins
  • David L. Dill
  • Robert Robere
  • Antonina Kolokolova
  • Yunhui Zheng
• [WGS03] Williams, R., Gomes, C.P. and Selman, B. Backdoors to typical case complexity. IJCAI 2003
• [BSG09] Dilkina, B., Gomes, C.P. and Sabharwal, A. Backdoors in the context of learning. SAT 2009
• [LGPC16] Liang, J.H., Ganesh, V., Poupart, P., and Czarnecki, K. Learning Rate Based Branching Heuristic for SAT Solvers. SAT 2016
• [RKG18] Robere, R., Kolkolova, A., Ganesh, V. Proof Complexity of SMT Solvers. CAV 2018