Constrained Sampling and Counting

Moshe Y. Vardi

Rice University

Joint work with **Kuldeep S. Meel**, **Supratik Chakraborty**, **Daniel Fremont**, **Rakesh Mistry**, **Sanjit Seshia**.

Boolean Satisfiability

Boolean Satisfiability (SAT); Given a Boolean expression, using "and" ($\wedge$) "or", ($\vee$) and "not" ($\neg$), *is there a satisfying solution* (an assignment of 0's and 1's to the variables that makes the expression equal 1)?

Example:

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

Solution: $x_1 = 0,\ x_2 = 0,\ x_3 = 1,\ x_4 = 1$

Complexity of Boolean Reasoning

History:

- William Stanley Jevons, 1835-1882: "I have given much attention, therefore, to lessening both the manual and mental labour of the process, and I shall describe several devices which may be adopted for saving trouble and risk of mistake."

- Ernst Schröder, 1841-1902: "Getting a handle on the consequences of any premises, or at least the fastest method for obtaining these consequences, seems to me to be one of the noblest, if not the ultimate goal of mathematics and logic."

- Cook, 1971, Levin, 1973: Boolean Satisfiability is NP-complete.

P vs. NP

- P: efficient *discovery* of solutions
- NP: efficient *checking* of solutions

The Big Question: Is $P = NP$ or $P \neq NP$?

- Is *checking* really easier than *discovering*?

Intuitive Answer: Of course, *checking* is easier than *discovering*, so $P \neq NP$!!!

- **Metaphor**: finding a needle in a haystack
- **Metaphor**: Sudoku
- **Metaphor**: mathematical proofs

Alas: We do not know how to *prove* that $P \neq NP$.

$P = NP$

S. Aaronson, MIT: "If $P = NP$, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in 'creative leaps,' no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss."

Consequences:

- Can solve efficiently numerous important problems.
- RSA encryption is not safe.

Question: Is it really possible that $P = NP$?

Answer: Yes! It'd require discovering a very clever algorithm, but it took 40 years to prove that LinearProgramming is in P.

Sharpening The Problem

NP-**Complete Problems**: *hardest problems* in NP

- Boolean Satisfiability is NP-complete! [Cook-Levin]

  Corollary: $P = NP$ if and only if Boolean Satisfiability is in P

There are *thousands* of NP-complete problems. To resolve the $P = NP$ question, it'd suffice to prove that *one* of them is or is not in P.

Convetional Wisdom: NP-complete problems are hard!

Algorithmic Boolean Reasoning: Early History

- Newell, Shaw, and Simon, 1955: "Logic Theorist"
- Davis and Putnam, 1958: "Computational Methods in The Propositional calculus", unpublished report to the NSA
- Davis and Putnam, JACM 1960: "A Computing procedure for quantification theory"
- Davis, Logemman, and Loveland, CACM 1962: "A machine program for theorem proving"

DPLL Method: Propositional Satisfiability Test

- Convert formula to conjunctive normal form (CNF)
- Backtracking search for satisfying truth assignment
- Unit-clause preference: If there is a clause p (resp., $\neg p$), assign p 1 (resp, 0).

Modern SAT Solving

CDCL = conflict-driven clause learning

- Backjumping
- Smart unit-clause preference
- Conflict-driven clause learning
- Smart decision heuristic (brainiac vs. speed demon)
- Randomized Restarts

Key Tools: GRASP, 1996; Chaff, 2001

Current capacity: *millions* of variables - *wide industrial usage*!

SAT Heuristic – Backjumping

Backtracking: go up one level in the search tree when both Boolean values for a variable have been tested.

Backjumping [Stallman-Sussman, 1977]: jump back in the search tree, if jump is safe – use highest node to jump to.

Key: Distinguish between

- *Decision variable*: Variable is that chosen and then assigned first c and then $1 - c$.
- *Implication variable*: Assignment to variable is forced by a unit clause.

Implication Graph: directed acyclic graph describing the relationships between decision variables and implication variables.

Smart Unit-Clause Preference

Boolean Constraint Propagation (BCP): propagating values forced by unit clauses.

- *Empirical Observation*: BCP can consume up to 80% of SAT solving time!

Requirement: identifying unit clauses

- *Naive Method*: associate a counter with each clause and update counter appropriately, upon assigning and unassigning variables.
- *Two-Literal Watching* [Moskewicz-Madigan-Zhao-Zhang-Malik, 2001]: "watch" two un-false literals in each unsatisfied clause – no overhead for backjumping.

SAT Heuristic – Clause Learning

Conflict-Driven Clause Learning: If assignment $\langle l_1, \ldots, l_n \rangle$ is bad, then add clause $\neg l_1 \vee \ldots \vee \neg l_n$ to block it.

Marques-Silva&Sakallah, 1996: This would add very long clauses! Instead:

- Analyze implication graph for chain of reasoning that led to bad assignment.
- Add a short clause to block said chain.
- The "learned" clause is a *resolvent* of prior clauses.

Consequence:

- Combine search with inference (*resolution*).
- Algorithm uses exponential space; "forgetting" heuristics required.

Smart Decision Heuristic

Crucial: Choosing decision variables wisely!

Dilemma: brainiac vs. speed demon

- *Brainiac*: chooses very wisely, to maximize BCP – decision-time overhead!
- *Speed Demon*: chooses very fast, to minimize decision time – many decisions required!

VSIDS [Moskewicz-Madigan-Zhao-Zhang-Malik, 2001]: *Variable State Independent Decaying Sum* – prioritize variables according to recent participation on conflicts – compromise between Brainiac and Speed Demon.

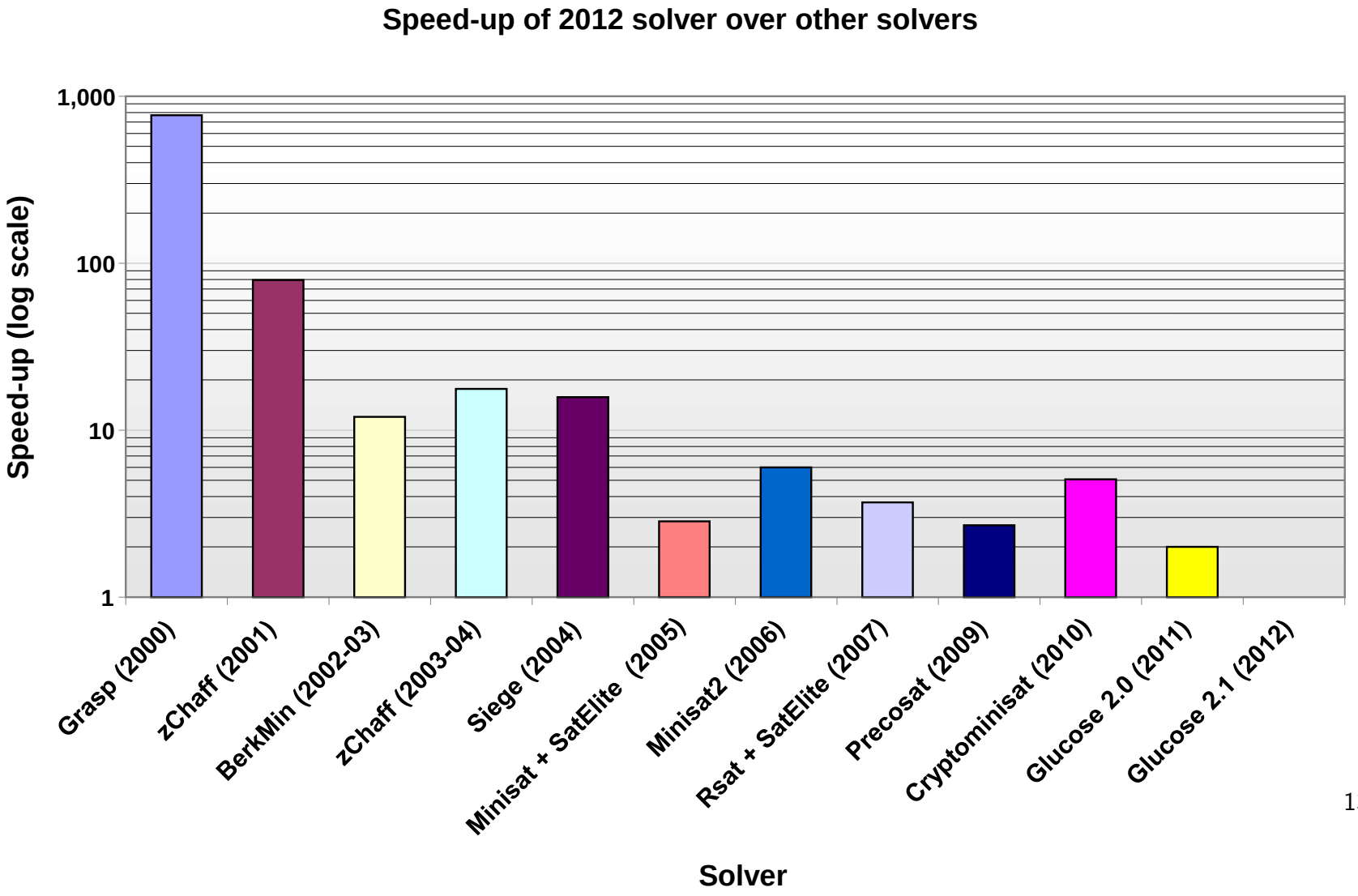
Randomized Restarts

Randomize Restart [Gomes-Selman-Kautz, 1998]

- Stop search
- Reset all variables
- Restart search
- *Keep* learned clauses

Aggressive Restarting: restart every $\sim$50 backtracks.

Some Experience with SAT Solving



Reflection on P vs. NP

Old Cliché "What is the difference between theory and practice? In theory, they are not that different, but in practice, they are quite different."

P vs. NP in practice:

- P=NP: Conceivably, NP-complete problems can be solved in polynomial time, but the polynomial is $n^{1,000}$ – *impractical*!

- P≠NP: Conceivably, NP-complete problems can be solved by $n^{\log\log\log n}$ operations – *practical*!

Conclusion: No guarantee that solving P vs. NP would yield *practical* benefits.

Are NP-Complete Problems Really Hard?

- When I was a graduate student, SAT was a "scary" problem, not to be touched with a 10-foot pole.
- Indeed, there are SAT instances with a few hundred variables that cannot be solved by any extant SAT solver.
- But today's SAT solvers, which enjoy wide industrial usage, routinely solve real-life SAT instances with millions of variables!

Conclusion: We need a richer and broader complexity theory, a theory that would explain both the difficulty and the easiness of problems like SAT.

Question: Now that SAT is "easy" in practice, how can we leverage that?

Verification of HW/SW systems

HW/SW Industry: \$0.75T per year!

Major Industrial Problem: *Functional Verification* – ensuring that computing systems satisfy their intended functionality: Verification consumes the majority of the development effort!

Two Major Approaches, both Computer Aided:

- *Formal Verification*: Constructing mathematical models of systems under verification and analyzing them mathematically: $\leq 10\%$ of verification effort – *CAV*!

- *Dynamic Verification*: simulating systems under different testing scenarios and checking the results: $\geq 90\%$ of verification effort – *not CAV*!

Dynamic Verification

- Dominant approach!
- Design is simulated with input test vectors.
- Test vectors represent different verification scenarios.
- Results compared to intended results.
- **Challenge**: Exceedingly large test space!

Motivating Example: HW FP Divider

$z = x/y$: x, y, z are 128-bit floating-point numbers

Question How do we verify that circuit works correctly?

- Try for all values of x and y?
- 2^{256} possibilities
- Sun will go nova before done! *Not scalable!*

Test Generation

Classical Approach: *manual* test generation - capture intuition about problematic input areas

- Verifier can write about 20 test cases per day: *not scalable!*

Modern Approach: *random-constrained* test generation

- Verifier writes *constraints* describing problematic inputs areas (based on designer intuition, past bug reports, etc.)

- Uses *constraint solver* to solve constraints, and uses solutions as test inputs – rely on industrial-strength constraint solvers!

- Proposed by Lichtenstein+Malka+Aharon (IBM), 1994: de-facto industry standard today!

Random Solutions

Major Question: How do we generate solutions *randomly* and *uniformly?*

- *Randomly*: We should not reply on solver internals to chose input vectors; we do not know where the errors are!

- *Uniformly*: We should not prefer one area of the solution space to another; we do not know where the errors are!

Uniform Generation of SAT Solutions: Given a SAT formula, generate solutions uniformly at random, while scaling to industrial-size problems.

Constrained Sampling: Applications

Many Applications:

- Constrained-random Test Generation: discussed above
- Personalized Learning: automated problem generation
- Search-Based Optimization: generate random points of the candidate space
- *Probabilistic Programming*: Sample after conditioning
- ...

Constrained Sampling – Prior Approaches, I

Theory:

- Jerrum+Valiant+Vazirani, 1986: *Random generation of combinatorial structures from a uniform distribution*: uniform generation in random polynomial time with a Σ_2^p oracle.

- Bellare+Goldreich+Petrank, 2000: *Uniform generation of NP-witnesses using an NP-oracle* – uniform generation in random polynomial time with an NP oracle.

We *implemented* the BPG Algorithm: did not scale above 16 variables!

Constrained Sampling – Prior Work, II

Practice:

• *BDD-based*: Yuan-Aziz-Pixley-Albin, 2004: *Simplifying Boolean constraint solving for random simulation-vector generation*

– Compute number of paths to 1 from each node, weight edges accordingly.

– Take a random walk on BDD from root to 1 according to weights.

– Scalability: hundreds of variables

• *Heuristics approaches*:

– Randomized solvers – good scalability, poor uniformity

– MCMC-based – good scalability, poor uniformity

Markov Chain Monte Carlo Methods

Basic Idea:

• Construct a Markov chain that has the desired distribution as its *stationary distribution*.

• Take a random walk on the chain until stationary distribution is achieved and then sample.

Shortcomings:

• Hard to construct Markov chain with uniform distribution over solutions of a Boolean formula.

• Exponentially long time to reach stationary distribution.

Almost-Uniform Generation of Solutions

New Algorithm – UniGen: Chakraborty–Meel–V, CAV'13+DAC'14:

- Almost-uniform generation in randomized polynomial time algorithms with a SAT oracle.
- Based on *universal hashing*.
- Uses an *SMT solver*.
- Scales to 100Ks of variables.
- Enables parallel generation of independent solutions after preprocessing.

Uniformity vs Almost-Uniformity

- Input formula: φ; Solution space: $Sol(\varphi)$
- Solution-space size: $\kappa = |Sol(\varphi)|$
- Uniform generation: for every assignment y: Prob$[Output = y]$=$1/\kappa$
- AImost-Uniform Generation: for every assignment y:
  $\frac{(1/\kappa)}{(1+\varepsilon)} \leq Prob[Output = y] \leq (1/\kappa) \times (1+\varepsilon)$

The Basic Idea

1. Partition $Sol(\varphi)$ into "roughly" equal small cells of appropriate size.

2. Choose a random cell.

3. Choose at random a solution in that cell.

You got random solution almost uniformly!

Question: How can we partition $Sol(\varphi)$ into "roughly" equal small cells without knowing the distribution of solutions?

Answer: *Universal Hashing* [Carter-Wegman 1979, Sipser 1983]

Universal Hashing

Hash function: maps $\{0,1\}^n$ to $\{0,1\}^m$

- Random inputs: All cells are roughly equal (in expectation)

Universal family of hash functions: Choose hash function *randomly* from family

- For *arbitrary* distribution on inputs: All cells are roughly equal (in expectation)

Strong Universality

Universal Family: Each input is hashed *uniformly*, but different inputs might not be hashed *independently*.

$H(n, m, r)$: Family of r-*universal* hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$ such that every r elements are mapped *independently*.

- *Higher* r: Stronger guarantee on *range of sizes* of cells
- r-*wise universality*: Polynomials of degree $r-1$

Strong Universality

Key: Higher universality $\Rightarrow$ higher complexity!

- **BGP**: n-universality $\Rightarrow$ all cells are small $\Rightarrow$ uniform generation
- **UniGen**: 3-universality $\Rightarrow$ a random cell is small w.h.p $\Rightarrow$ almost-uniform generation

From tens of variables to 100Ks of variables!

XOR-Based 3-Universal Hashing

- Partition $\{0,1\}^n$ into 2^m cells.
- *Variables*: $X_1, X_2, \ldots X_n$
- Pick every variable with probability $1/2$, XOR them, and equate to 0/1 with probability $1/2$.
  - E.g.: $X_1 + X_7 + \ldots + X_{117} = 0$ (splits solution space in half)
- m XOR equations $\Rightarrow 2^m$ cells
- *Cell constraint*: a conjunction of CNF and XOR clauses

SMT: Satisfiability Modulo Theory

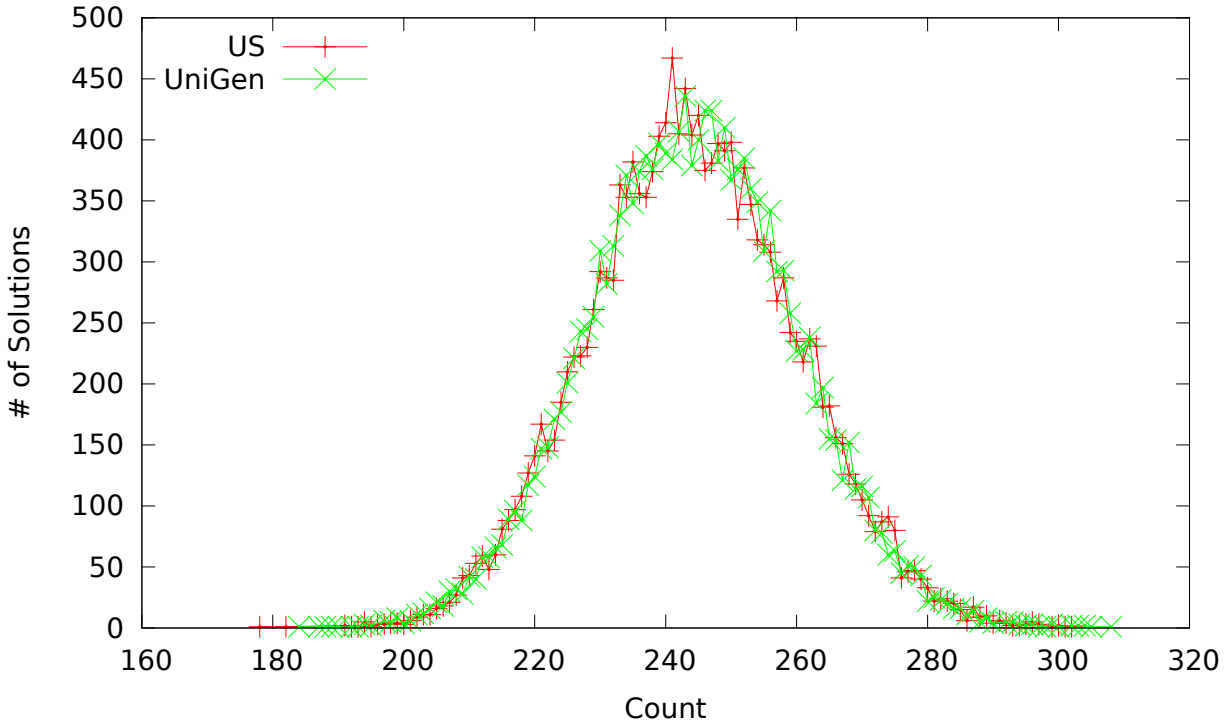
SMT Solving: Solve Boolean combinations of constraints in an underlying theory, e.g., linear constraints, combining SAT techniques and domain-specific techniques.

- Tremendous progress since 2000!

CryptoMiniSAT: M. Soos, 2009

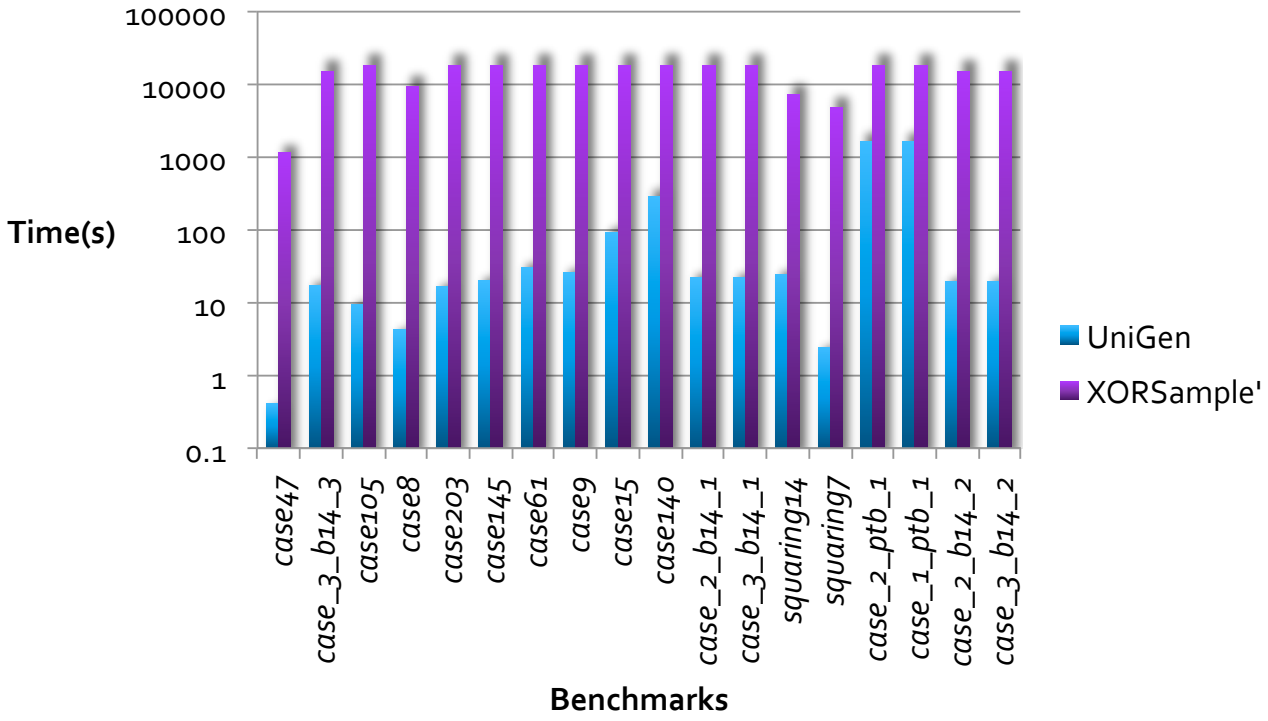
- Specialized for combinations of CNF and XORs
- Combine SAT solving with Gaussian elimination

UniGen Performance: Uniformity



Uniformity Comparison: UniGen vs Uniform Sampler

UniGen Performance: Runtime



Runtime Comparison: UniGen vs XORSample'

From Independence to Almost Independence

Question: Each hash-value cell has many solutions, but we keep only one. Why?

Answer: Because solutions in the same cell are not *independent*!

Question: Why do we need independent solutions?

Answer: We need independence to be able to *multiply probabilities*, to increase confidence of hitting bugs.

Relaxing Independence: If we relax independence, we need *more* solutions to reach confidence level, but we use *fewer SAT calls*!

Chakraborty-Fremont-Meel-Seshia-V, 2015: From 100Ks of variables to millions of variables!

From Sampling to Counting

- Input formula: φ; Solution space: $Sol(\varphi)$
- **#SAT Problem**: Compute $|Sol(\varphi)|$
  - $\varphi = (p \vee q)$
  - $Sol(\varphi) = \{(0,1),(1,0),(1,1)\}$
  - $|Sol(\varphi)| = 3$

Fact: $\#SAT$ is complete for $\#P$ – the class of counting problems for decision problems in NP [Valiant, 1979].

Constrained Counting

A wide range of applications!

- Coverage in random-constrained verification
- Probabilistic inference
- Planning with uncertainty
- ...

But: $\#SAT$ is really a hard problem! In practice, quite harder than SAT.

Approximate Counting

Probably Approximately Correct (PAC):

- *Formula*: φ, *Tolerance*: ε, *Confidence*: $0 < \delta < 1$
- $|Sol(\varphi)| = \kappa$
- $Prob[\frac{\kappa}{(1+\varepsilon)} \leq \text{Count} \leq \kappa \times (1+\varepsilon) \geq \delta$
- Introduced in [Stockmeyer, 1983]
- [Jerrum+Sinclair+Valiant, 1989]: BPP^{NP}
- No implementation so far.

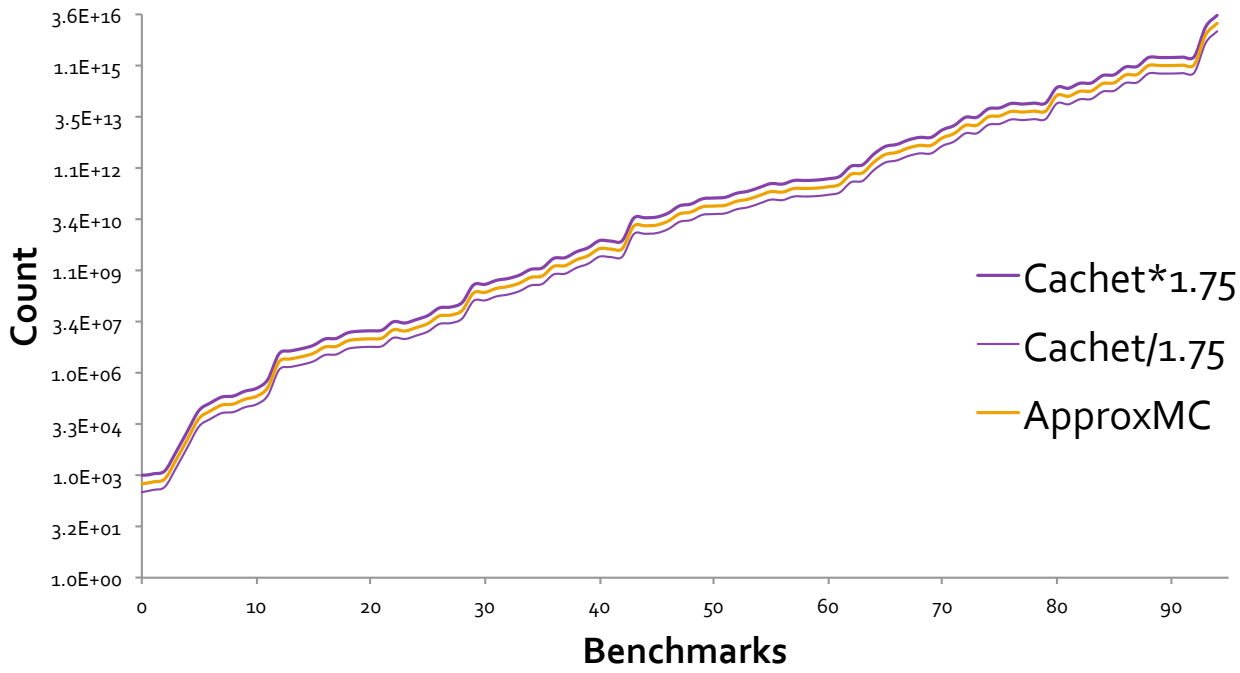
From Sampling to Counting

ApproxMC: [Chakraborty+Meel+V., 2013]

- Use m random XOR clauses to select at random an appropriately small cell.
- Count number of solutions in cell and multiply by 2^m to obtain estimate of $|Sol(\varphi)|$.
- Iterate until desired confidence is achieved.

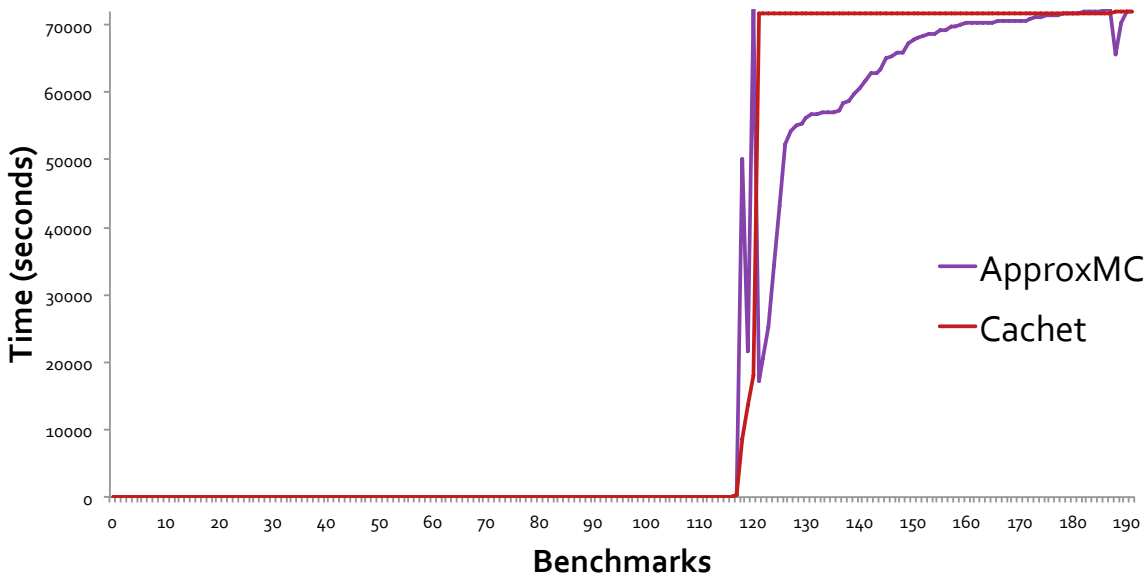
ApproxMC runs in time polynomial in $|\varphi|$, ε^{-1}, and $\log(1-\delta)^{-1}$, relative to SAT oracle.

ApproxMC Performance: Accuracy



Accuracy: ApproxMC vs Cachet (exact counter)

ApproxMC Performance: Runtime



Runtime Comparison: ApproxMC vs Cachet'

Scaling Performance – from Bits to Bit Vectors

Observation: Benchmarks are often word-level.

[Chistikov-Dimitrova-Majumdar, 2015]: Lift ApproxMC to word-level benchmarks, via bit blasting.

Why Bit Blasting? Because of XOR-based hash function.

Lifting Hashing [Chakraborty+Meel+Mistry+V., 2016]:

- XOR constraints are linear equations over the finite field F_2.
- We can lift to F_p – for prime p – by using linear equations.

Word-Level Approximate Counting

SMTApproxMC: Word-level approximate constrained counter, built on top of Boolector [Chakraborty+Meel+Mistry+V., 2016]

Empirical Evaluation:

- Usually outperforms CDM.
- Underperforms CDM in benchmarks with large variety of bitvector widths.

To be done: Enhancing Boolector with *Gaussian elimination*.

The Core of UniGen and ApproxMC

Before we can sample cells, we need to know how many cells are needed, e.g., how many hash bits are needed.

- For cells to be small in expectation, number of cells should be proposortional to size of solution space.

- But we do not know the size of the solution space!!!

- Run a search to find appropriate number of hash bits for cell to be small.

Improving the Core

[Chakraborty-Meel-V., 2016]:

- **Old Thinking**: Search probes must be independent, so search must be sequential.
- **New Thinking**: Search probes can be *weakly* independent, so search can be logarithmic!

ApproxMC2 [Chakraborty+Meel+V., 2016]:

- Uses logarithmic rather than sequential search in core algorithm.
- About 10X performance improvement over ApproxMC.

- Applicable to other hasning-based algorithms.

In Conclusion

- The improvement in the performance of SAT solvers over the past 20 years is *revolutionary*!
  - Better marketing: Deep Solving
- SAT solving is an enabler, e.g., approximate sampling and counting
- When you have a big hammer, look for nails!!!
  - Example: Statistical analysis of systems
- Scalability is an ongoing challenge!