The Automated-Reasoning Revolution: from Theory to Practice and Back

Moshe Y. Vardi

Rice University

Is This Time Different? The Opportunities and Challenges of Artificial Intelligence

Jason Furman, Chair, Council of Economic Advisers, July 2016:

"Even though we have not made as much progress recently on other areas of AI, such as logical reasoning, the advancements in deep learning techniques may ultimately act as at least a partial substitute for these other areas."

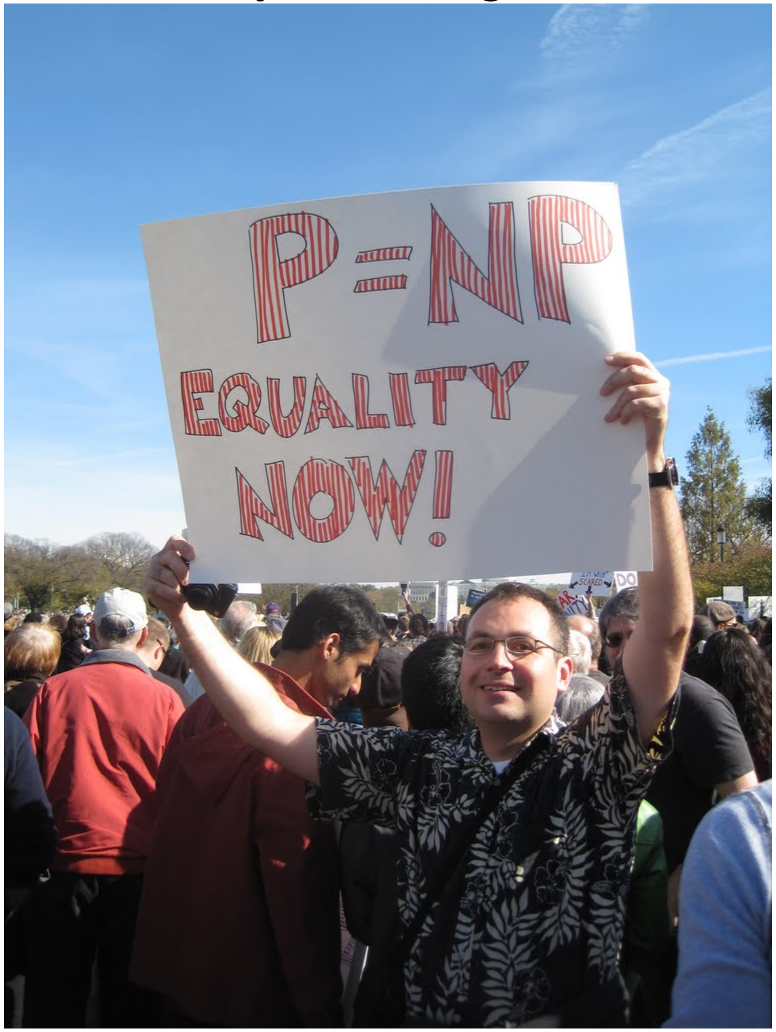
P $vs.$ NP: An Outstanding Open Problem

Does $P = NP$?

- *The* major open problem in theoretical computer science
- A major open problem in mathematics
  - A Clay Institute Millennium Problem
  - Million dollar prize!

What is this about? It is about computational complexity – how hard it is to solve computational problems.

Rally To Restore Sanity, Washington, DC, October 2010



Computational Problems

Example: *Graph* – $G = (V, E)$

- V – set of nodes
- E – set of edges

Two notions:

- Hamiltonian Cycle: a cycle that visits every *node* exactly once.
- Eulerian Cycle: a cycle that visits every *edge* exactly once.

Question: How hard it is to find a Hamiltonian cycle? Eulerian cycle?

Figure 1: The Bridges of Königsberg

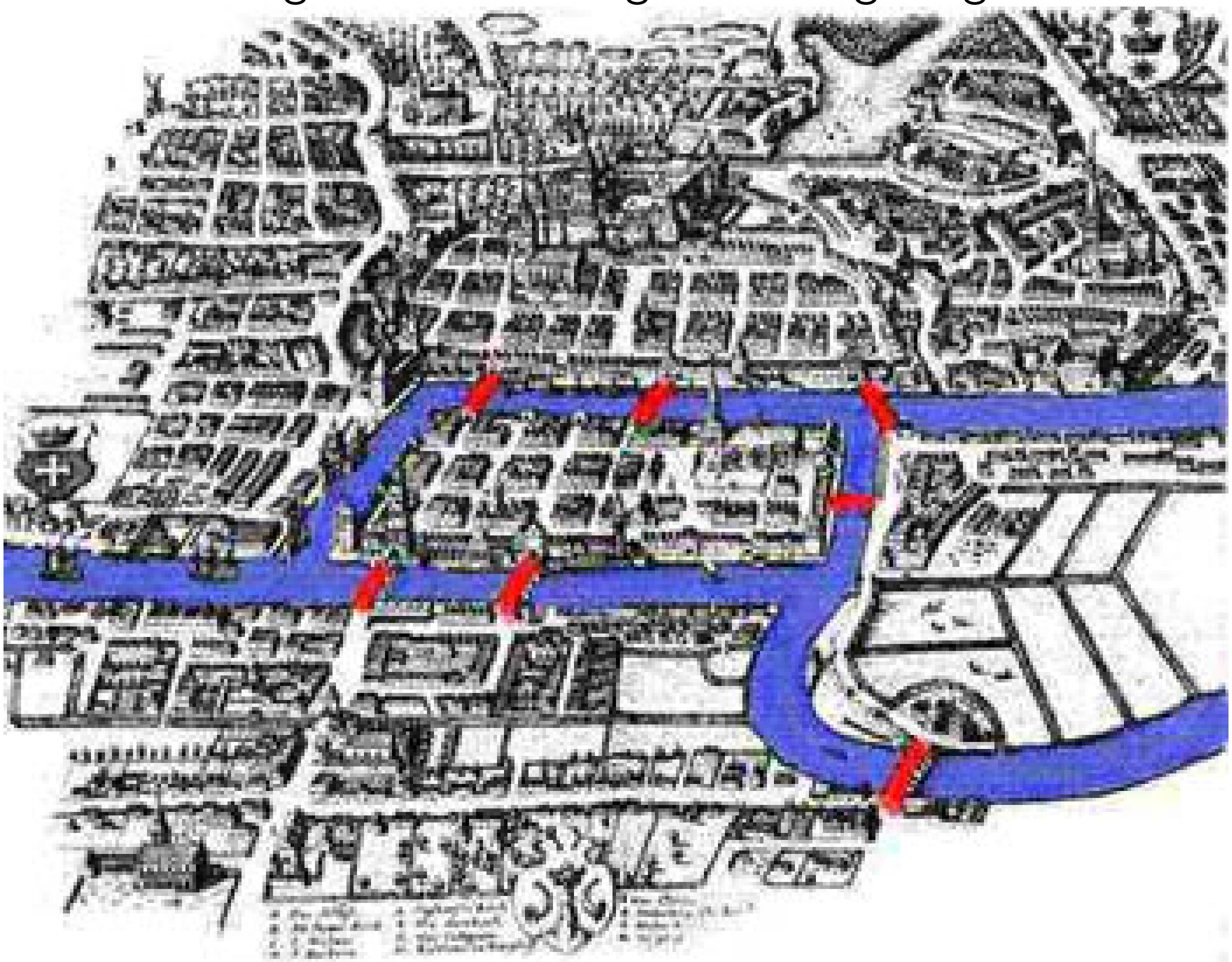


Figure 2: The Graph of The Bridges of Königsberg

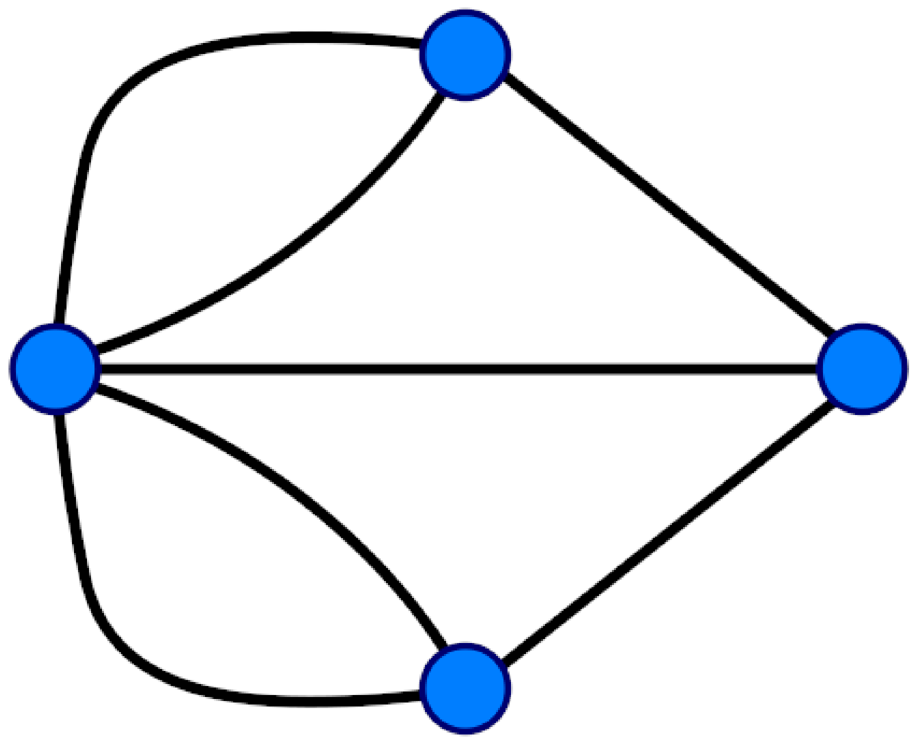
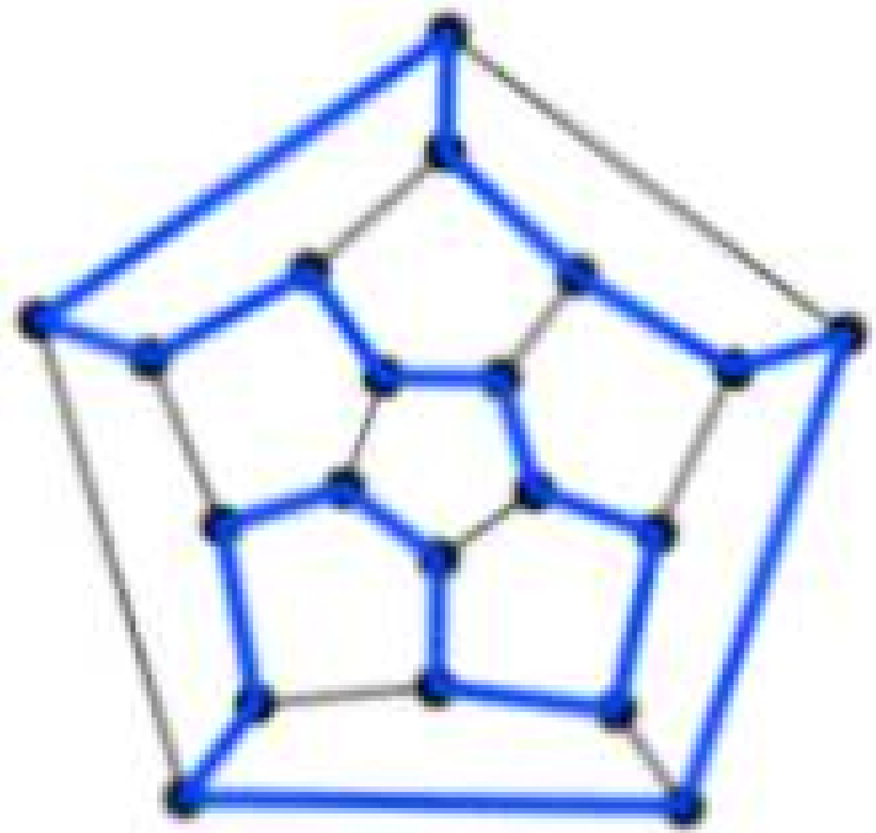


Figure 3: Hamiltonian Cycle



Computational Complexity

Measuring complexity: How many (Turing machine) operations does it take to solve a problem of size n?

- *Size* of (V, E): number of nodes plus number of edges.

Complexity Class P: problems that can be solved in *polynomial time* – n^c for a *fixed* c

Examples:

- Is a number even?
- Is a number square?
- Does a graph have an Eulerian cycle?

What about the Hamiltonian Cycle Problem?

Hamiltonian Cycle

- **Naive Algorithm**: Exhaustive search – run time is $n!$ operations
- **“Smart” Algorithm**: Dynamic programming – run time is 2^n operations

Note: The universe is much younger than 2^{200} Planck time units!

Fundamental Question: Can we do better?

- Is HamiltonianCycle in P?

Checking Is Easy!

Observation: Checking if a *given* cycle is a Hamiltonian cycle of a graph $G = (V, E)$ is *easy*!

Complexity Class NP: problems where solutions can be *checked* in polynomial time.

Examples:

- HamiltonianCycle
- Factoring numbers

Significance: Tens of thousands of optimization problems are in NP!!!

- CAD, flight scheduling, chip layout, protein folding, ...

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 $P \neq NP$.

$$P \neq NP$$

Consequences:

- Cannot solve efficiently numerous important problems
- RSA encryption may be safe.

Question: Why is it so important to *prove* $P \neq NP$, if that is what is commonly believed?

Answer:

- If we cannot prove it, we do not really understand it.
- May be $P = NP$ and the "enemy" proved it and broke RSA!

$P = NP$

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 is NP

- HamilatonianCycle is NP-complete! [Karp, 1972]

Corollary: $P = NP$ if and only if HamiltonianCycle 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.

History

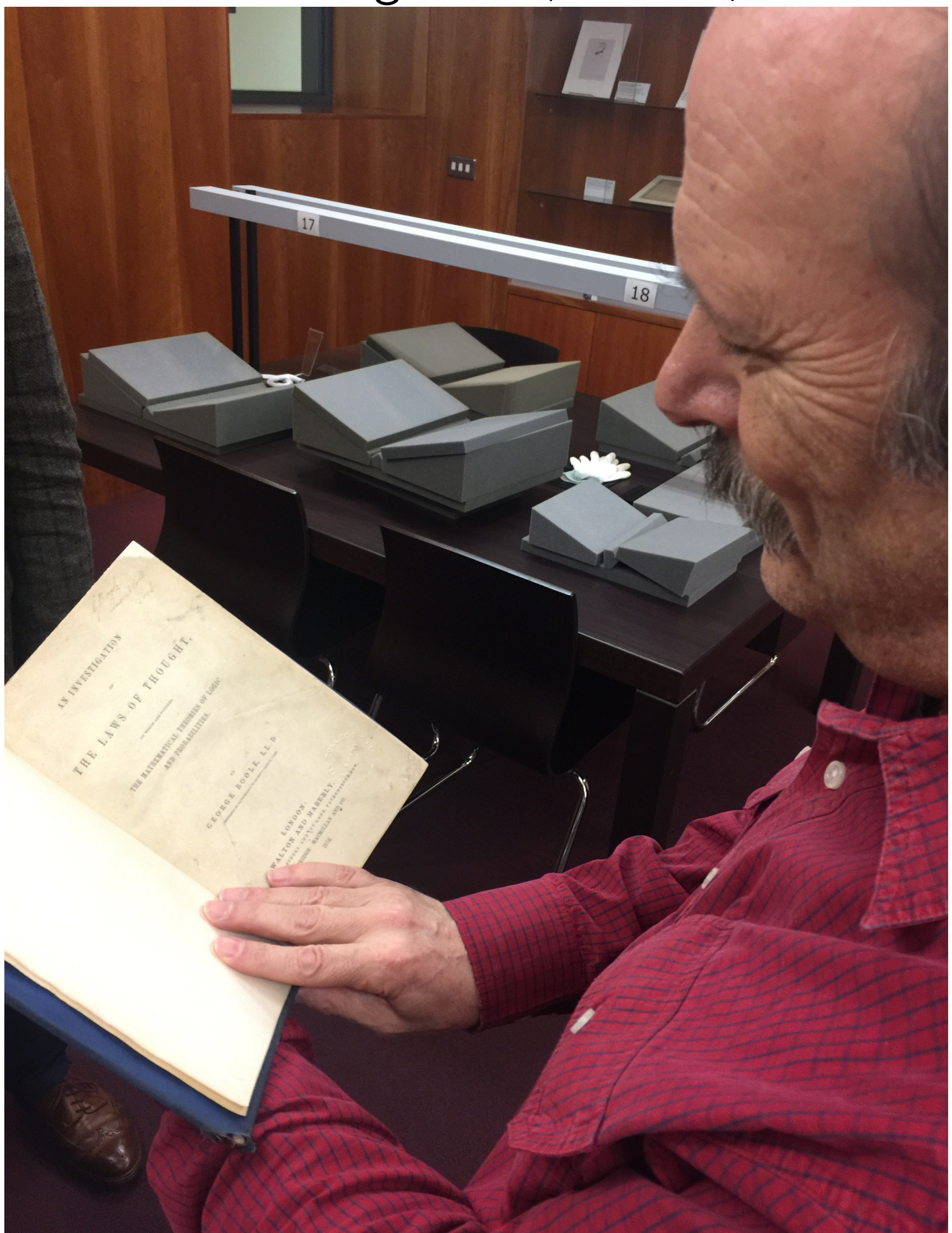
- 1950-60s: Futile effort to show hardness of search problems.
- Stephen Cook, 1971: Boolean Satisfiability is NP-complete.
- Richard Karp, 1972: 20 additional NP-complete problems– 0-1 Integer Programming, Clique, Set Packing, Vertex Cover, Set Covering, Hamiltonian Cycle, Graph Coloring, Exact Cover, Hitting Set, Steiner Tree, Knapsack, Job Scheduling, ...
  – *All* NP-complete problems are polynomially equivalent!
- Leonid Levin, 1973 (independently): Six NP-complete problems
- M. Garey and D. Johnson, 1979: "Computers and Intractability: A Guide to NP-Completeness" - hundreds of NP-complete problems!
- Clay Institute, 2000: $1M Award!

Boole's Symbolic Logic

Boole's insight: Aristotle's syllogisms are about *classes* of objects, which can be treated *algebraically*.

> "If an adjective, as 'good', is employed as a term of description, let us represent by a letter, as y, all things to which the description 'good' is applicable, i.e., 'all good things', or the class of 'good things'. Let it further be agreed that by the combination xy shall be represented that class of things to which the name or description represented by x and y are simultaneously applicable. Thus, if x alone stands for 'white' things and y for 'sheep', let xy stand for 'white sheep'.

Vardi at Univ. College Cork, Ireland, March 2017



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.

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

Modern SAT Solving

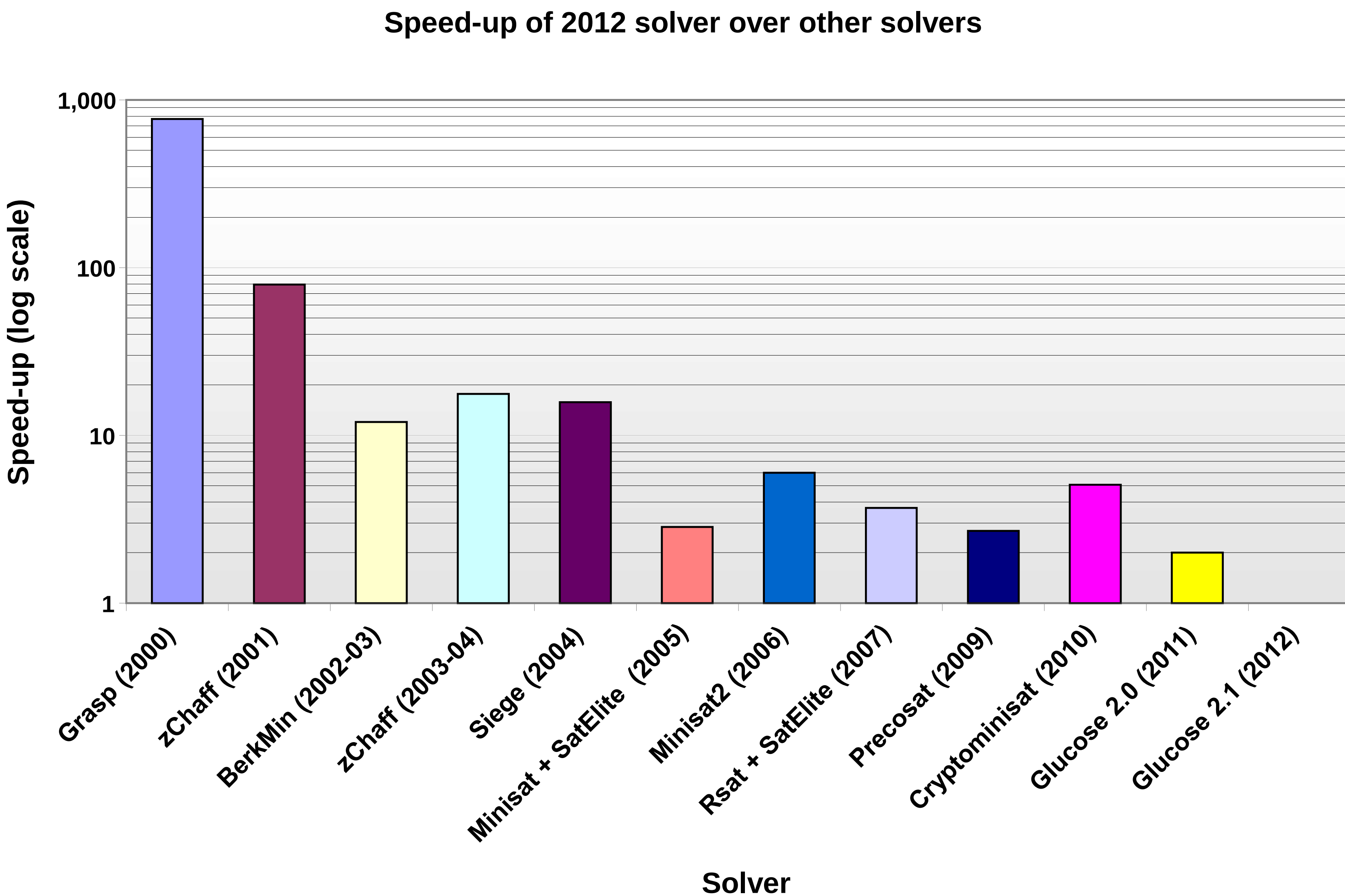
CDCL = conflict-driven clause learning

- Backjumping
- Smart unit-clause preference
- Conflict-driven clause learning (and forgetting!)
- Smart choice heuristic (brainiac vs speed demon)
- Restarts

Key Tools: GRASP, 1996; Chaff, 2001

Current capacity: *millions* of variables

Some Experience with SAT Solving



Knuth Gets His Satisfaction

SIAM News, July 26, 2016: "Knuth Gives Satisfaction in SIAM von Neumann Lecture"

Donald Knuth gave the 2016 John von Neumann lecture at the SIAM Annual Meeting. The von Neumann lecture is SIAM's most prestigious prize.

Knuth based the lecture, titled "Satisfiability and Combinatorics", on the latest part (Volume 4, Fascicle 6) of his The Art of Computer Programming book series. He showed us the first page of the fascicle, aptly illustrated with the quote "I can't get no satisfaction," from the Rolling Stones. In the preface of the fascicle Knuth says "The story of satisfiability is the tale of a triumph of software engineering, blended with rich doses of beautiful mathematics".

Applications of SAT Solving in SW Engineering

Leonardo De Moura+Nikolaj Björner, 2012: Applications of Z3 at Microsoft

- Symbolic execution
- Model checking
- Static analysis
- Model-based design
- ...

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:

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

- *Dynamic Verification*: simulating systems under different testing scenarios and checking the results: $\geq$ 90% of verification effort

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, 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 Inference*: Sample after conditioning
- . . .

Constrained Sampling – Prior Approaches, I

Theory:

- Jerrum+Valiant+Vazirani: *Random generation of combinatorial structures from a uniform distribution*, TCS 1986 – uniform generation by a randomized polytime algrithm with an Σ_2^p oracle.
- Bellare+Goldreich+Petrank: *Uniform generation of NP-witnesses using an NP-oracle*, 2000 – uniform generation by a randomized polytime algorithm 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: *Simplifying Boolean constraint solving for random simulation-vector generation*, 2004 – poor scalability
- *Heuristics approaches*: MCMC-based, randomized solvers, etc. – good scalability, poor uniformity

Almost Uniform Generation of Solutions

New Algorithm – UniGen: Chakraborty, Fremont, Meel, Seshia, V, 2013-15:

- almost uniform generation by a randomized polytime algorithms with a SAT oracle.
- Based on *universal hashing*.
- Uses an *SMT solver*.
- Scales to millions of variables.
- Enables parallel generation of 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$[Output = y]$=$1/\kappa$
- Almost-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 millions 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

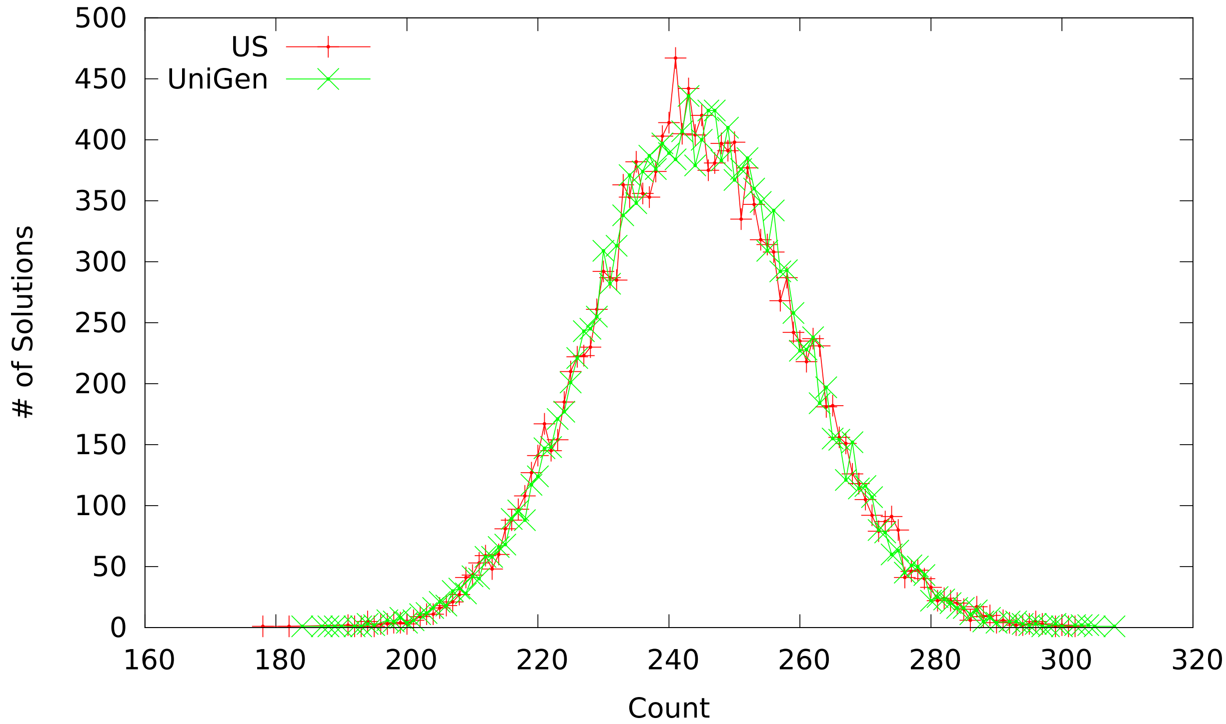
SMT Solving: Solve Boolean combinations of constraints in an underlying theory, e.g., linear constraints, combining SAT techniques and domain-specific techniques.

- Tremendous progress since 2000!

CryptoMiniSAT: M. Soos, 2009

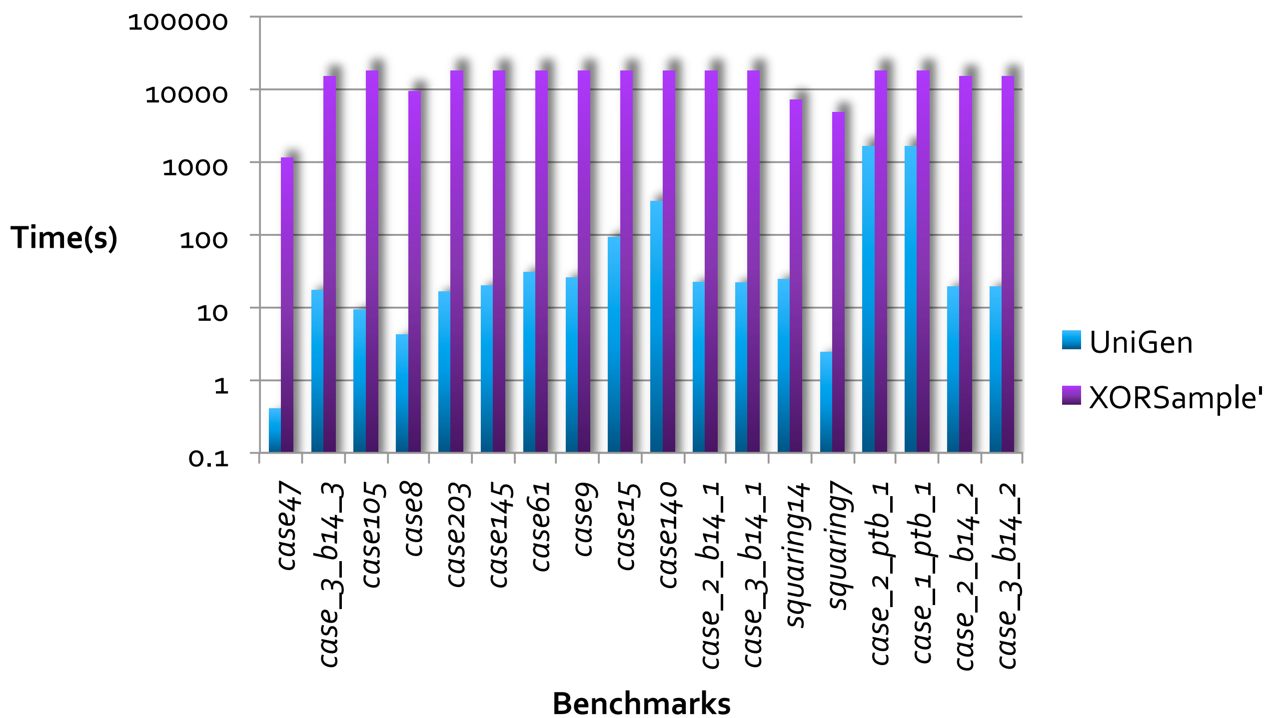
- Specialized for combinations of CNF and XORs
- Combine SAT solving with Gaussian elimination

UniGen Performance: Uniformity



Uniformity Comparison: UniGen vs Uniform Sampler

UniGen Performance: Runtime



Runtime Comparison: UniGen vs XORSample'

From 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
- Bayesian 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.

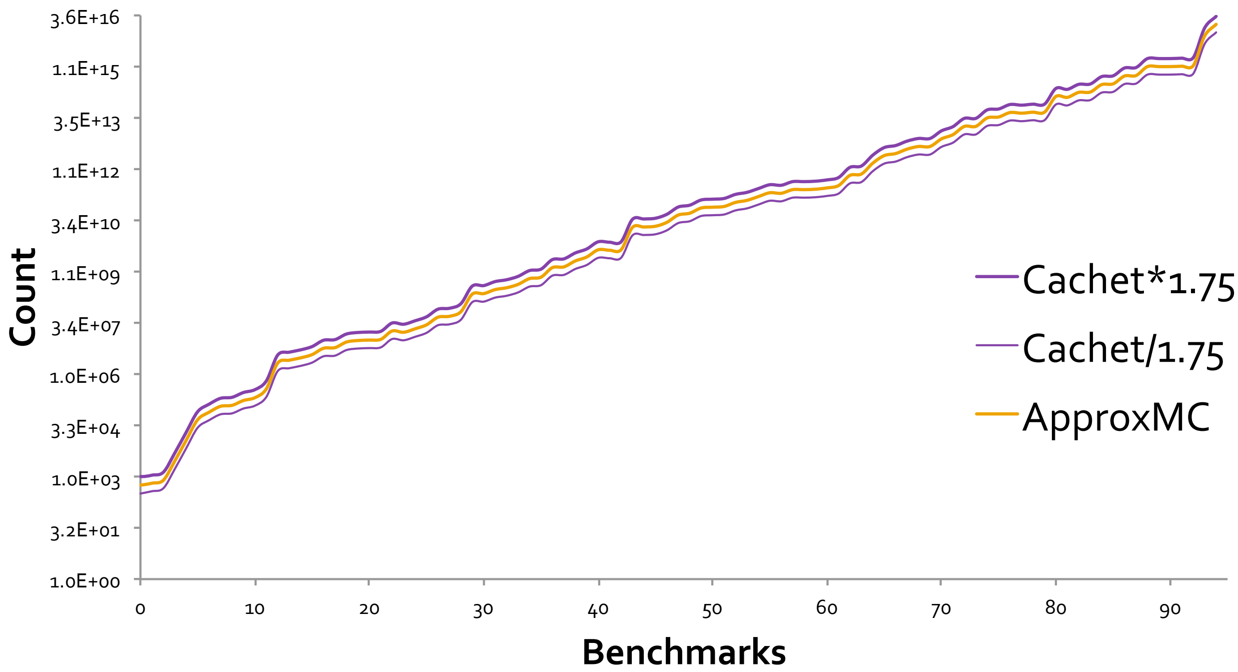
From Sampling to Counting

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.

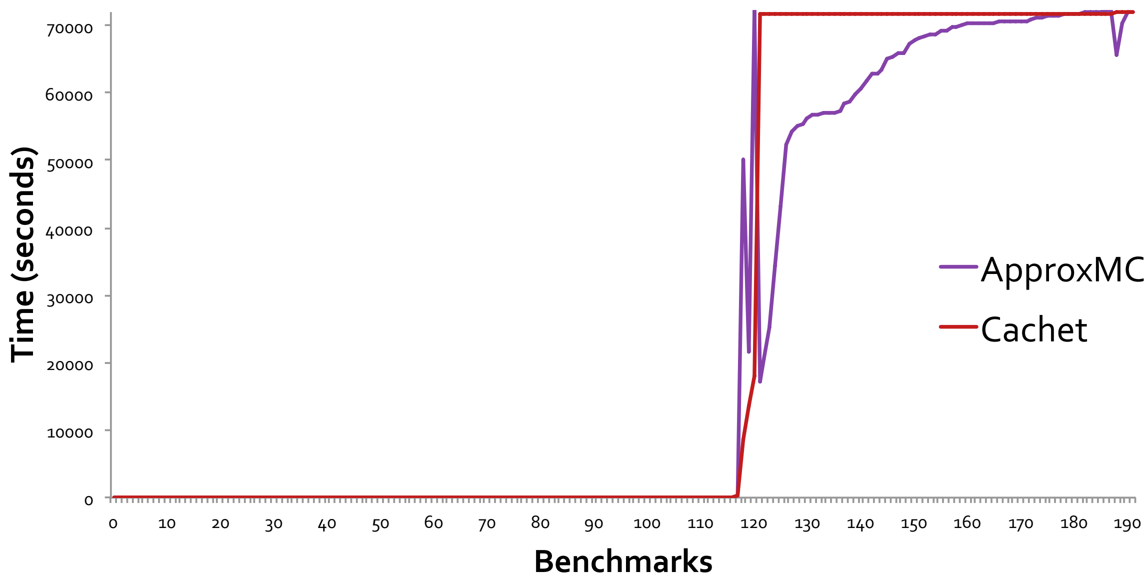
ApproxMC runs in time polynomial in $|\varphi|$, ε^{-1}, and $\log(1-\delta)^{-1}$, relative to SAT oracle.

ApproxMC Performance: Accuracy



Accuracy: ApproxMC vs Cachet (exact counter)

ApproxMC Performance: Runtime



Runtime Comparison: ApproxMC vs Cachet'

SAT Solving

- 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!!!
- Scalability is an ongoing challenge!

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?

- We showed how to leverage for sampling and counting. What else?
- Is BPP^{NP} the "new" $PTIME$?