From Model-Driven Computer Science to Data-Driven Computer Science and Back

Moshe Y. Vardi

Rice University

Is Computer Science Fundamentally Changing?

Formal Science vs Data Science

- *Common perception*: A Kuhnian paradigm shift!

– "Throw out the old, bring in the new!"

- *In reality*: new scientific theories *refine* old ones.

– After all, we went to the moon with Newtonian Mechanics!

- *My Thesis*: Data science *refines* formal science!

This Talk: Two personal examples: database query languages, Boolean satisfiability solving

Database Query Languages

Basic Framework Codd, 1970:

- *Fixed Schema*: e.g., EMP-DPT, DPT-MGR
- Standard database query languages (e.g., SQL 2.0) are essentially syntactically sugared 1st-order logic (FOL).

Beyond FOL:

- Aho&Ullman, 1979: 1st-order languages are weak – add *recursion*
- Gallaire&Minker,1978: add recursion via *logic programs*
- SQL 3.0, 1999: recursion added

Datalog

Datalog [Maier&Warren, 1988]:

- Function-free logic programs
- Select-project-join-union-recurse queries

Example: *Transitive Closure*

$$Path(x, y) \ :- \ Edge(x, y)$$

$$Path(x, y) \ :- \ Path(x, z), Path(z, y)$$

Example: Impressionable Shopper

$$Buys(x, y) \ :- \ Trendy(x), Buys(z, y)$$

$$Buys(x, y) \ :- \ Likes(x, y)$$

Query Containment, I

Query Optimization: Given Q, find Q' such that:

- $Q \equiv Q'$
- Q' is "easier" than Q

Query Containment: $Q_1 \sqsubseteq Q_2$ if $Q_1(B) \subseteq Q_2(B)$ for all databases B.

Fact: $Q \equiv Q'$ iff $Q \sqsubseteq Q'$ and $Q' \sqsubseteq Q$

Consequence: Query containment is a *key* database problem.

Query Containment, II

Decidability of Query Containment:

- *SQL*: undecidable
  - Folk Theorem (unsolvability of FO)
  - Poor theory and practice of optimization
- *SPJU Queries*: decidable
  - Chandra&Merlin, 1977, Sagiv&Yannakakis,- 1982
  - Rich theory and practice of optimization

Select-Project-Join-Union Queries:

- Covers the vast majority of real-life database queries

Example: $Triangle(x, y) \ :- \ Edge_1(x, y), Edge_1(y, z), Edge_(z, x)$
$Triangle(x, y) \ :- \ Edge_2(x, y), Edge_2(y, z), Edge_2(z, x)$

Query Containment, III

Datalog Containment:

- Complexity: undecidable
  - Shmueli, 1987: easy reduction from CFG containment
- Difficult theory and practice of optimization

Unfortunately, most decision problems involving Datalog are undecidable - very few interesting, well-behaved fragments.

Reminder: Datalog=SPJU+Recursion

Question: Can we limit recursion to recover decidability?

1990s: Graph Databases

WWW: *Nodes, Edges Labels*

Graph Data: WWW, SGML documents, library catalogs, XML documents, meta-data, ...

Graph Databases: *No fixed Schema* – (D, E, λ)

- D - nodes
- $E \subseteq D^2$ - edges
- $\lambda : E \to \Lambda$ – edge labels (more general than node labels)

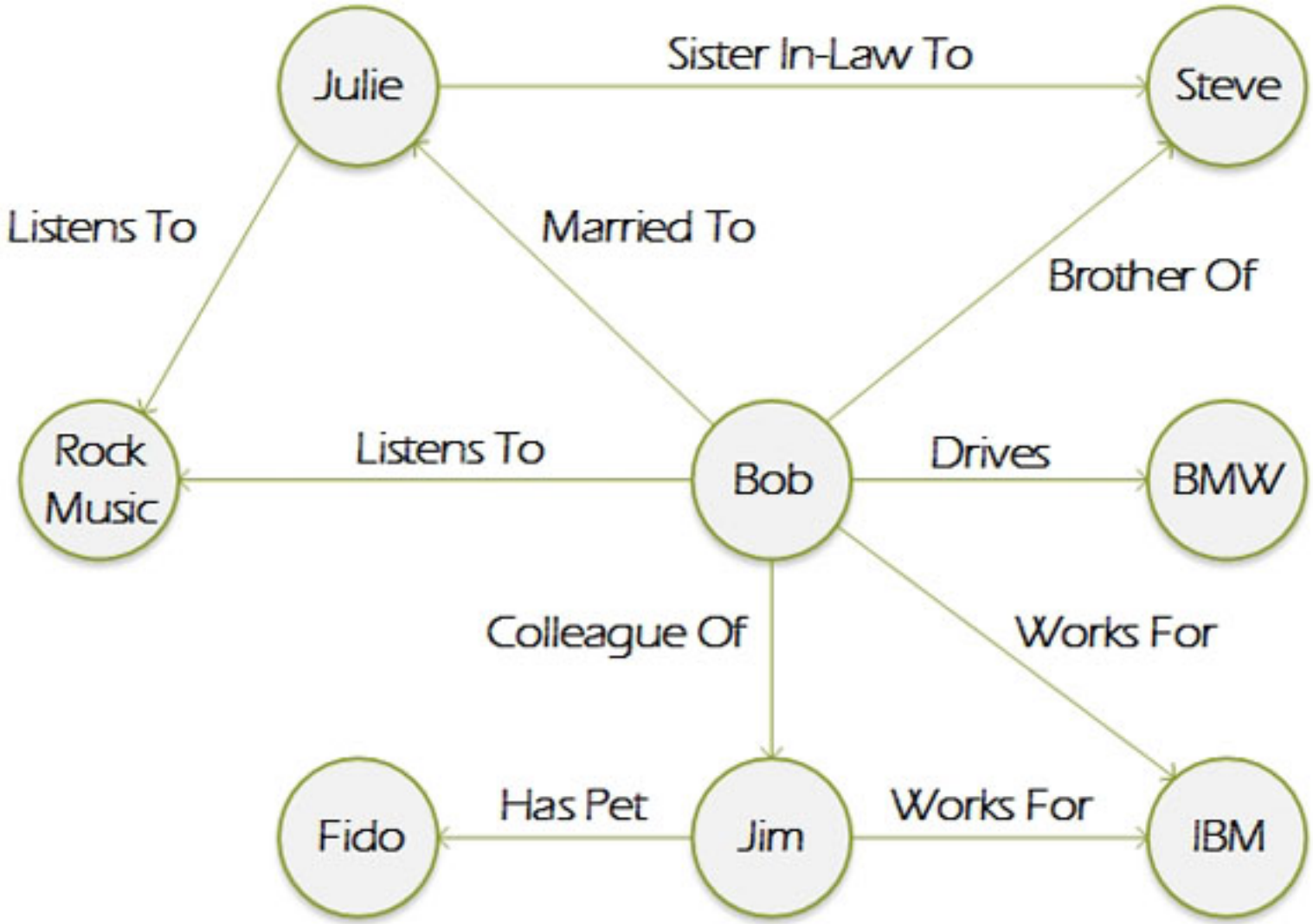


Figure 1: Graph Database

Path Queries

Active Research Topic: What is the right query language for graph databases? ("No SQL")

Basic Element of all proposals: *path queries*

- $Q(x, y) \ :- \ x \ L \ y$
- L: formal language over labels
- $a \cdot \underline{\ l_1\ } \cdots \underline{\ l_k\ } \cdot b$
- $Q(a, b)$ holds if $l_1 \cdots l_k \in L$

Example: *Regular Path Query*

$$Q(x, y) \ :- \ x \ (Wing \cdot Part^{+} \cdot Nut) \ y$$

Regular Path Queries

Observation:

- A fragment of binary Datalog
  - *Concatenation*: $E(x,y) \ :- \ E_1(x,z), E_2(z,y)$
  - *Union*: $E(x,y) \ :- \ E_1(x,y)$
    $E(x,y) \ :- \ E_2(x,y)$
  - *Transitive Closure*: $P(x,y) \ :- \ E(x,y)$
    $P(x,y) \ :- \ E(x,z), E(z,y)$

Path-Query Containment

$$Q_1(x, y) \ :- \ x \ L_1 \ y, \qquad Q_2(x, y) \ :- \ x \ L_2 \ y$$

Language-Theoretic Lemma 1:

$$Q_1 \sqsubseteq Q_2 \text{ iff } L_1 \subseteq L_2$$

Proof: Consider a database

$$a \cdot \underline{\quad l_1 \quad} \cdots \underline{\quad l_k \quad} \cdot b \text{ with } l_1 \cdots l_k \in L_1$$

Corollary: Path-Query Containment is

- undecidable for context-free path queries
- PSPACE-complete for regular path queries.

Two-Way RPQs

Extended Alphabet: $\Lambda^- = \{a^- \ : \ a \in \Lambda\}$, $\Lambda' = \Lambda \cup \Lambda^-$

Inverse Roles:

$Part(x, y)$: y part of x
$Part^-(x, y)$: x part of y

Example: $(1/2)^*$ Siblings

$Q(x, y) \ :-$
$x \ \ [(father^- \cdot father) + (mother^- \cdot mother)]^+ \ \ y$

[Calvanese-De Giacomo-Lenzerini-V., 2000]: 2RPQ containment is PSPACE-complete.

Closing 2RPQs under $\cap$ and $\cup$

Intersection:

- Regular languages are closed under intersection and union.
- Intersection adds succinctness: RE($\cap$)$<$RE

Intersection vs. Conjunction:

$Q_1(x, y) \ : -(x(E_1 \cap E_2)y)$
$Q_2(x, y) \ : -(xE_1y)\&(xE_2y)$

Conclusion: Intersection$\neq$Conjunction for graph databases!

UC2RPQ: Closure of 2RPQs under union and conjunction

UC2RPQ

UC2RPQ: Core of all graph query languages

$$Q(x_1, \ldots, x_n) \ :- \ y_1 E_1 z_1, \ldots, y_m E_m z_m$$

- E_i – UC2RPQ

Intuition:

- UC2RPQs are obtained from SPJU by replacing atoms with REs over Λ'.
- UC2RPQs are Select-Project-Union-"Regular Join" queries.

Example:

$$\begin{aligned} Q(x, y) \ :- \ & z \ \ (Wing \cdot Part^+ \cdot Nut) \ \ x, \\ & z \ \ (Wing \cdot Part^+ \cdot Nut) \ \ y \end{aligned}$$

UC2RPQ Containment

Difficulty: Earlier techniques do not apply

- Database techniques cannot handle transitive closure.
- No language-theoretic lemma to reduce to automata.

Solution: combine database-theoretic and automata-theoretic techniques:

[Calvanese-De Giacomo-V., 2000&2003]: UC2RPQ containment is EXPSPACE-complete.

Regular Queries

UC2RPQs:

- *Elements*: disjunction, conjunction, and transitive closure
- *Closure*: disjunction, conjunction

Example: Not in UC2RPQ!
$Q(x,y) \ :-(xE_1z)\&(zE_2y)\&(xE_3y)$
$Answe(x,y) \ :-(xQ^*y)$

RQ: closure under disjunction, conjunction, *and* transitive closure (TC)
Essentially: *Replace recursion by TC*.

RQ Containment: 2EXPSPACE-complete [Reutter&Romero&V., 2015]

Question: Practical?

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'.

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

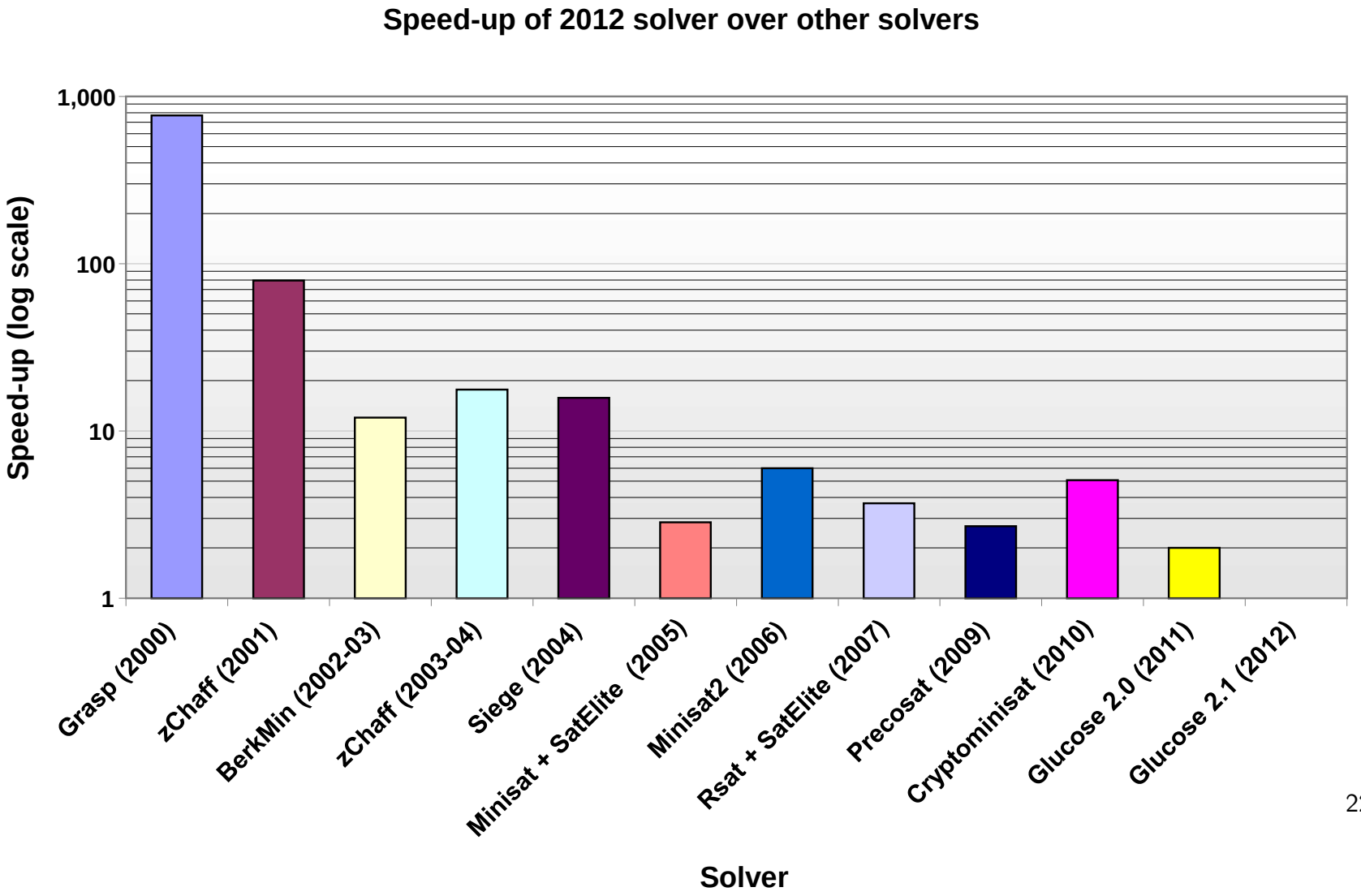
CDCL = conflict-driven clause learning

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

Key Tools: GRASP, 1996; Chaff, 2001

Current capacity: *millions* of variables

Some Experience with SAT Solving



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 analzying 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 rely 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 in $BPP^{\Sigma_2^p}$

- Bellare+Goldreich+Petrank: *Uniform generation of NP-witnesses using an NP-oracle*, 2000 – uniform generation in BPP^{NP}.

But: 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 in BPP^{NP} (randomized polynomial time algorithms with a SAT oracle)
- Based on *universal hashing*.
- Uses an *SMT solver*.
- Scales to 100,000s of variables.

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)

XOR-Based 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

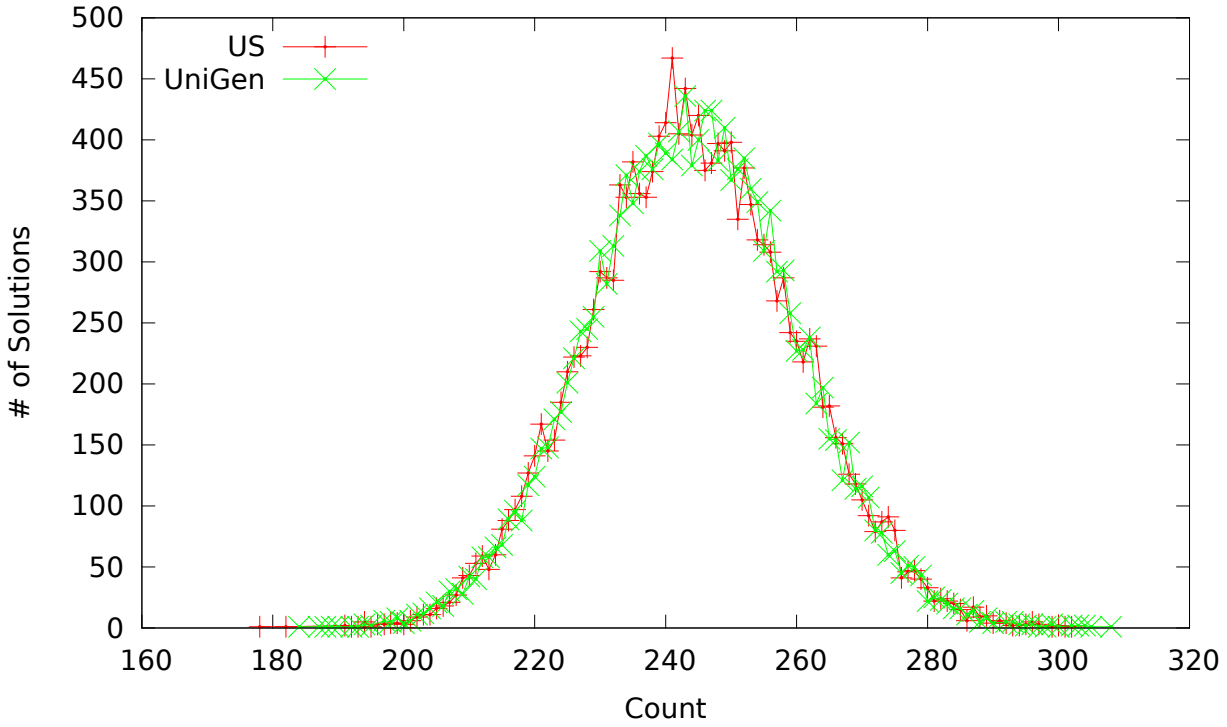
SMT Solving: Solve Boolean combinations of constraints in an underlying theory, e.g., linear constraints, combining SAT techniques and domain-specific techniques.

- Tremendous progress since 2000!

CryptoMiniSAT: M. Soos, 2009

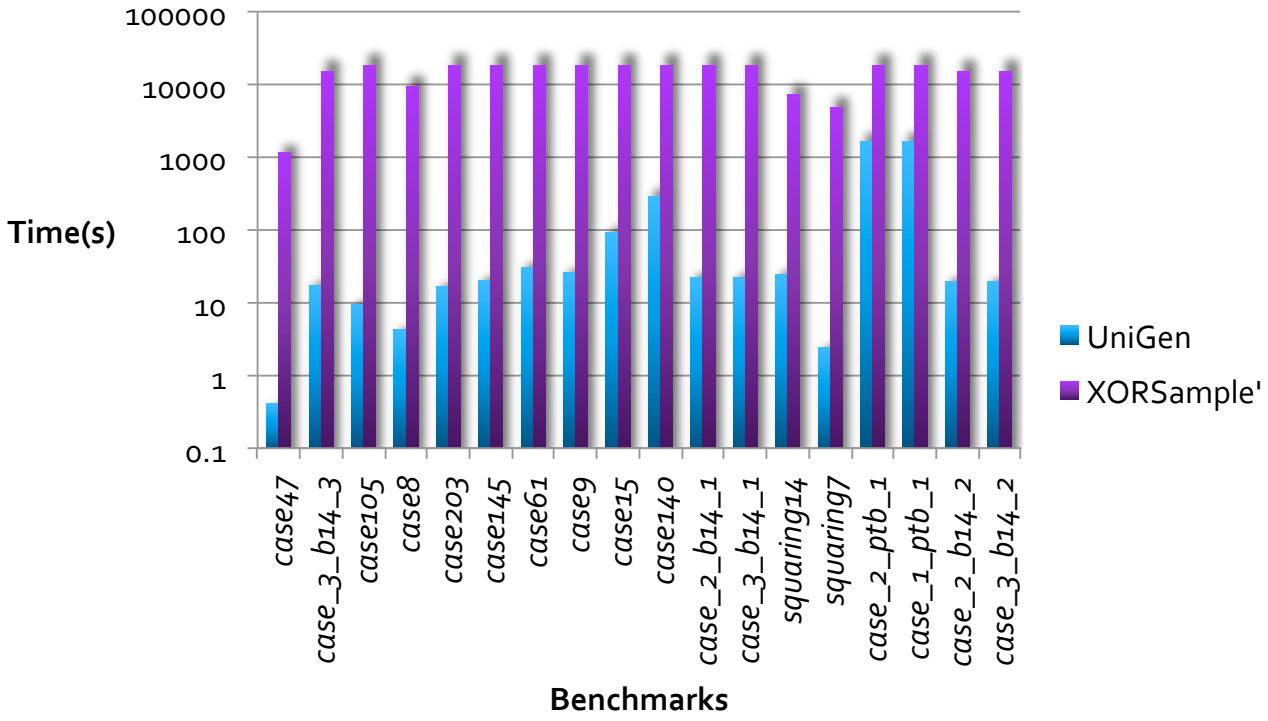
- Specialized for combinations of CNF and XORs
- Combine SAT solving with Gaussian elimination

UniGen Performance: Uniformity



Uniformity Comparison: UniGen vs Uniform Sampler

UniGen Performance: Runtime



Runtime Comparison: UniGen vs XORSample'

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?

- If not worst-case complexity, then what?

From Model-Driven Computer Science to Data-Driven Computer Science and Back

In Summary:

- It is a *paradigm glide*, not *paradigm shift*.
- Data-driven CS *refines* model-driven CS, it does *not* replace it.
- Physicists still teach Mechanics, Electromagnetism, and Optics.
- We should still teach Algorithms, Logic, and Formal Languages.