Discrete Sampling and Integration for the AI Practitioner

Supratik Chakraborty, IIT Bombay
Kuldeep S. Meel, Rice University
Moshe Y. Vardi, Rice University

Agenda

Part 1: Boolean Satisfiability Solving (Vardi)

Part 2(a): Applications (Chakraborty)

Coffee Break

Part 2(b): Prior Work (Chakraborty)

Part 3: Hashing-based Approach (Meel)

Discrete Sampling and Integration for the AI Practitioner
Part I: Boolean Satisfiability Solving

Supratik Chakraborty, **IIT Bombay**
Kuldeep S. Meel, **Rice University**
Moshe Y. Vardi, **Rice University**

Boolean Satisfiability

Boolean Satisfiability (SAT); Given a Boolean expression φ, 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)? That is, is $Sol(\varphi)$ nonempty?

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

Discrete Sampling and Integration

Discrete Sampling: Given a Boolean formula φ, sample from $Sol(\varphi)$ uniformly at random?

Discrete Integration: Given a Boolean formula φ, compute $|Sol(\varphi)|$.

Weighted Sampling and Integration: As above, but subject to a weight function $w: \ Sol(\varphi) \mapsto R^+$

Basic Theoretical Background

Discrete Integration: #SAT

Known:

1. #SAT is #P-complete.

2. In practice, #SAT is quite harder than SAT.

3. If you can solve #SAT, then you can sample uniformly using self-reducibility.

Desideratum: Solve discrete sampling and integration using a SAT solver.

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."

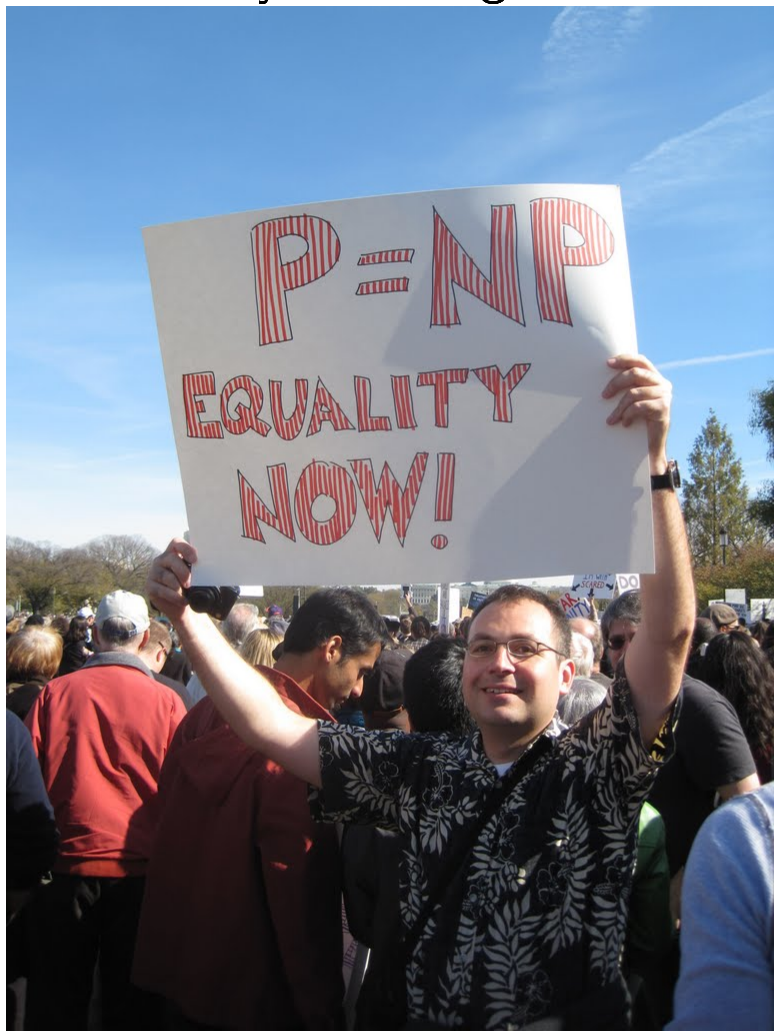
P $vs.$ NP: An Outstanding Open Problem

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

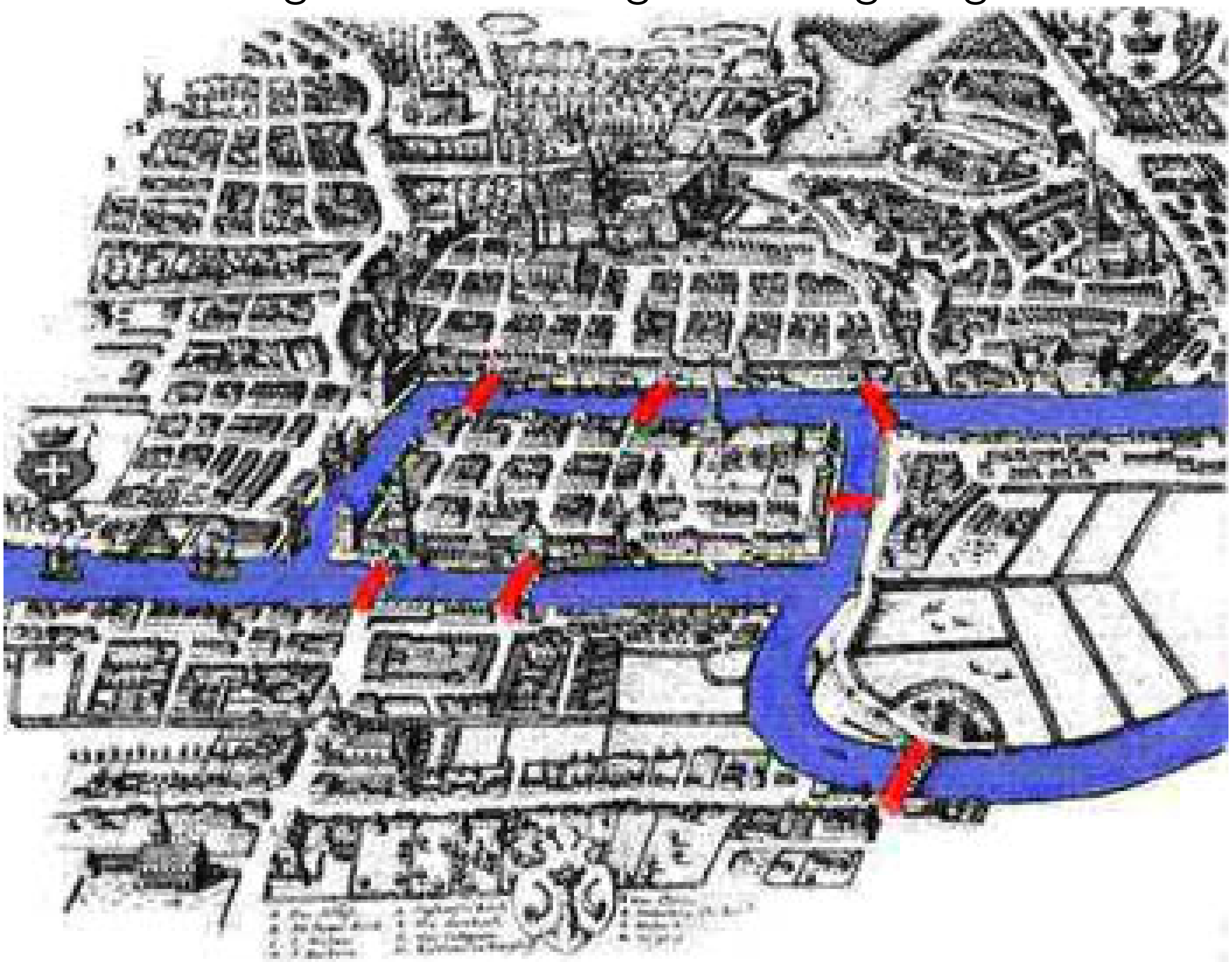


Figure 2: The Graph of The Bridges of Königsburg

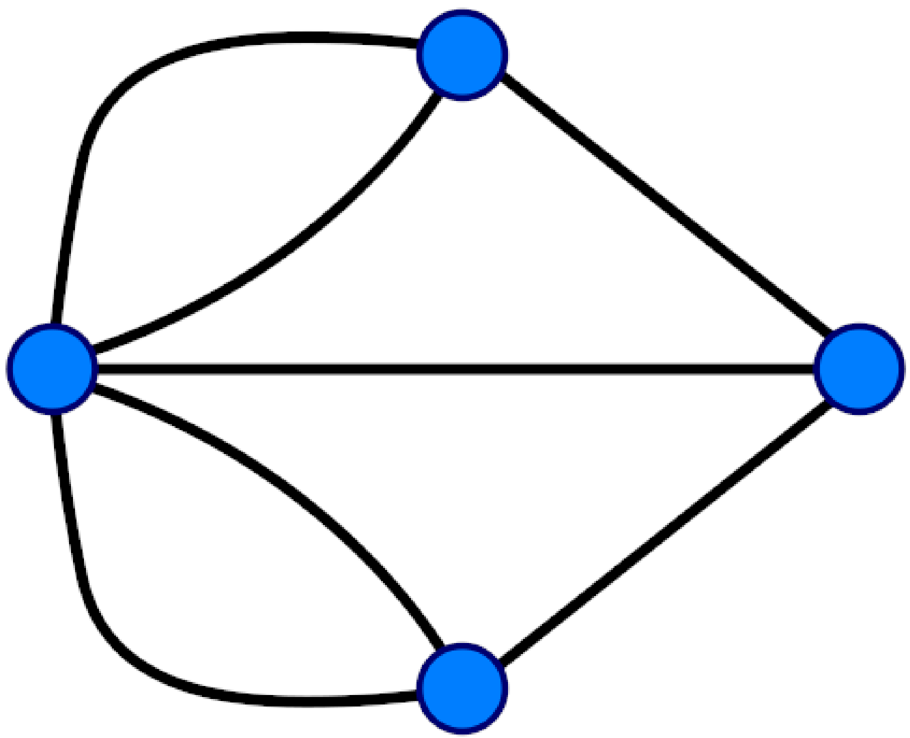
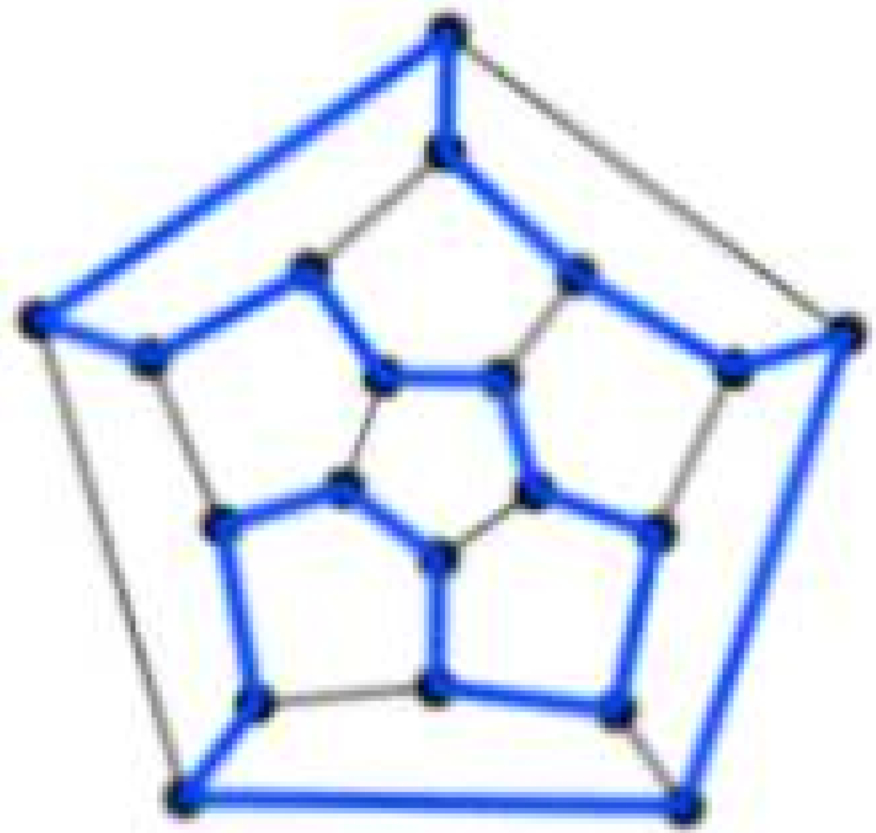


Figure 3: Hamiltonian Cycle



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: *Perebor Project* – 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'.

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

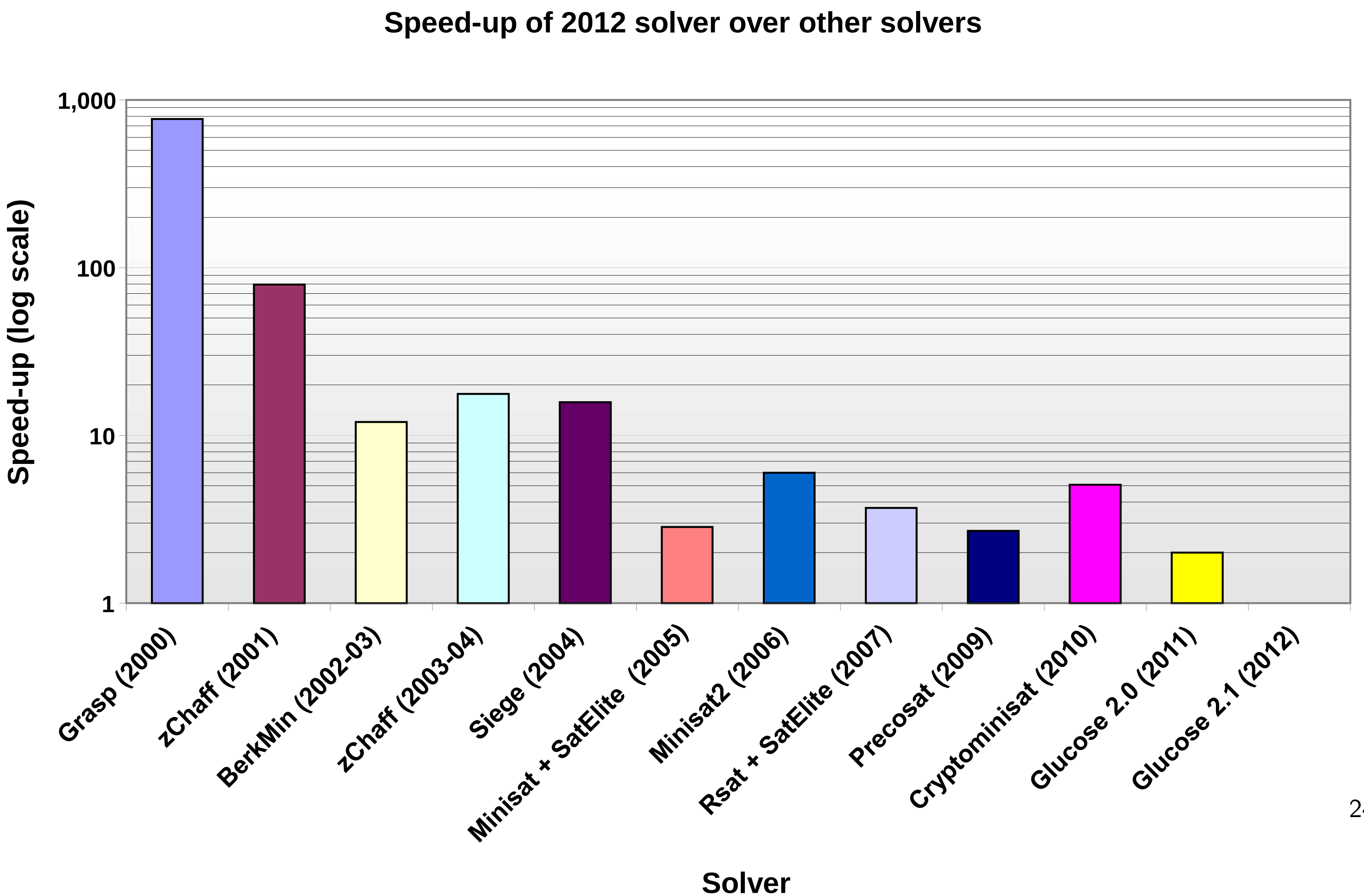
CDCL = conflict-driven clause learning

- 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



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”.

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.

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!

Example: SMTLA
$(x > 10) \wedge [((x > 5) \vee (x < 8)]$

Sample Application: *Bounded Model Checking of Verilog programs* – SMT(BV).

SMT Solving

General Approach: combine SAT-solving techniques with theory-solving techniques

- Consider formula as Boolean formula ove theory atoms.
- Solve Boolean formula; obtain conjunction of theory atoms.
- Use theory solver to check if conjunction is satisfiable.

Crux: Interaction between SAT solver and theory solver, e.g., *conflict-clause learning* – convert unsatisfiable theory-atom conjection to a new Boolean clause.

Applications of SAT/SMT Solving in SW Engineering

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

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

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?

- Is BPP^{NP} the "new" $PTIME$?

Notation

- Given
  - $X_1, \dots X_n$: variables with finite discrete domains $D_1, \dots D_n$
  - Constraint (logical formula) φ over $X_1, \dots X_n$
  - Weight function $W: D_1 \times \dots D_n \rightarrow Q^{\geq 0}$

  Sol(φ) : set of assignments of $X_1, \dots X_n$ satisfying φ

  - Determine $W(\varphi) = \sum_{y \in Sol(\varphi)} W(y)$
    If $W(y) = 1$ for all y, then $W(\varphi) = | Sol(\varphi) |$

Discrete Integration (Model Counting)

  - Randomly sample from Sol(φ) such that Pr[y is sampled] $\propto$ W(y)
    If $W(y) = 1$ for all y, then uniformly sample from Sol(φ)

Discrete Sampling

For this tutorial: Initially, D_i's are {0,1} – Boolean variables

Later, we'll consider D_i's as $\{0, 1\}^n$ – Bit-vector variables

Closer Look At Some Applications

- Discrete Integration
  - Probabilistic Inference
  - Network (viz. electrical grid) reliability
  - Quantitative Information flow
  - And many more …
- Discrete Sampling
  - Constrained random verification
  - Automatic problem generation
  - And many more …

Application 1: Probabilistic Inference

- An **alarm** rings if it's in a working state when an **earthquake** happens or a **burglary** happens
- The **alarm** can malfunction and ring without **earthquake** or **burglary** happening

- Given that the **alarm** rang, what is the likelihood that an **earthquake** happened?

- Given conditional dependencies (and conditional probabilities) calculate **Pr[event | evidence]**
  - What is **Pr [Earthquake | Alarm]** ?

Probabilistic Inference: Bayes' Rule

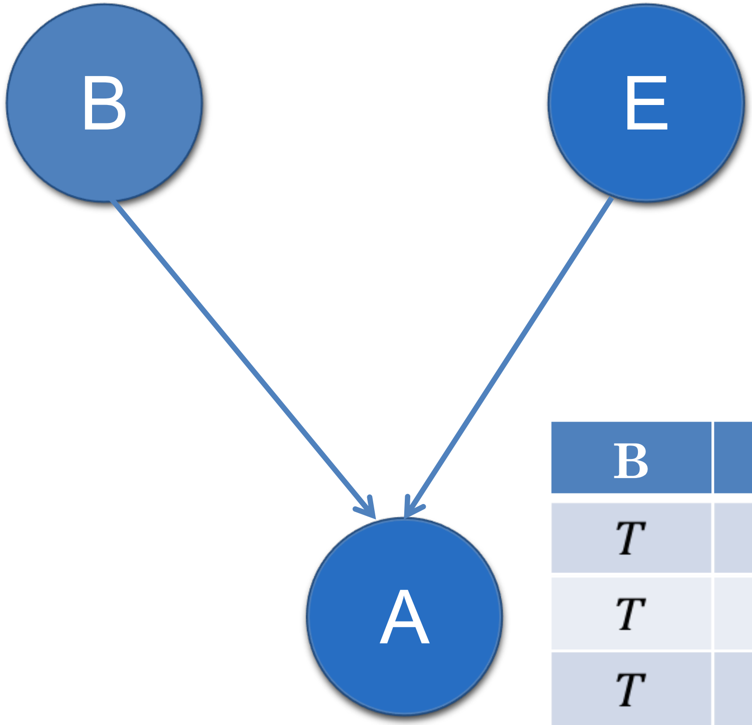
$$\Pr[event_i \mid evidence] = \frac{\Pr[event_i \cap evidence]}{\Pr[evidence]} = \frac{\Pr[event_i \cap evidence]}{\sum_j \Pr[event_j \cap evidence]}$$

$$\Pr[event_j \cap evidence] = \Pr[evidence \mid event_j] \times \Pr[event_j]$$

How do we represent conditional dependencies efficiently, and calculate these probabilities?

Probablistic Inference: Graphical Models

B	Pr
T	0.8
F	0.2



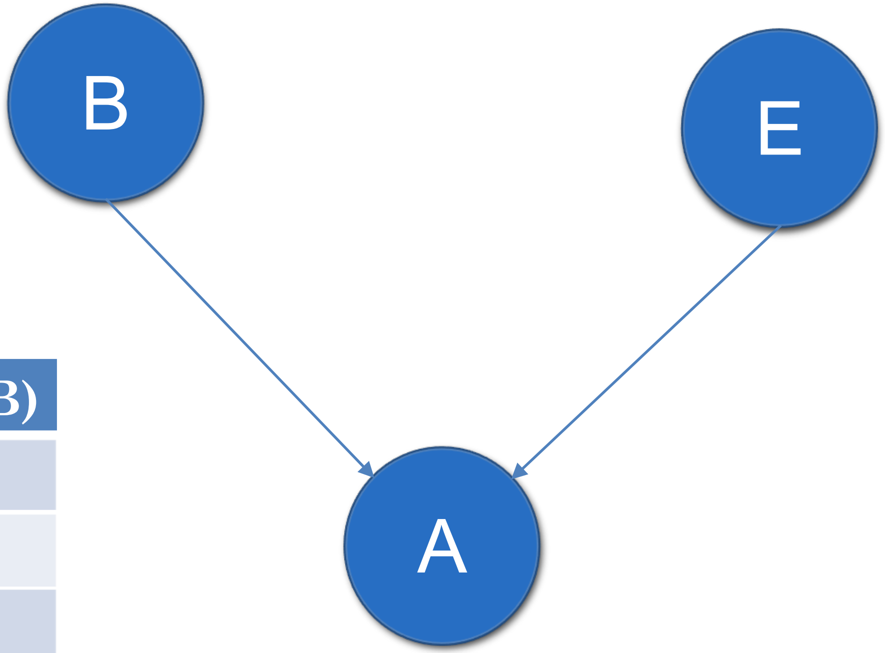
E	Pr
T	0.1
F	0.9

B	E	A	Pr(A | E,B)
T	T	T	0.3
T	T	F	0.7
T	F	T	0.4
T	F	F	0.6
F	T	T	0.2
F	F	F	0.8
F	F	T	0.1
F	F	F	0.9

Conditional Probability Tables (CPT)

Probabilistic Inference: First Principle Calculation

B	Pr
T	0.8
F	0.2



E	Pr
T	0.1
F	0.9

B	E	A	Pr(A | E,B)
T	T	T	0.3
T	T	F	0.7
T	F	T	0.4
T	F	F	0.6
F	T	T	0.2
F	F	F	0.8
F	F	T	0.1
F	F	F	0.9

$$
\begin{aligned}
\Pr[E \cap A] = \\
&\Pr[E] * \Pr[\neg B] * \Pr[A \mid E, \neg B] \\
&+ \Pr[E] * \Pr[B] * \Pr[A \mid E, B]
\end{aligned}
$$

Probabilisitc Inference: Logical Formulation

$V = \{v_A, v_{\sim A}, v_B, v_{\sim B}, v_E, v_{\sim E}\}$ Prop vars corresponding to events

$T = \{t_{A|B,E}\ ,\ t_{\sim A|B,E}\ ,\ t_{A|B,\sim E}\ \ldots\}$ Prop vars corresponding to CPT entries

Formula encoding probabilistic graphical model (φ_{PGM}):

$(v_A \oplus v_{\sim A}) \wedge (v_B \oplus v_{\sim B}) \wedge (v_E \oplus v_{\sim E})$ Exactly one of v_A and $v_{\sim A}$ is true

$$\wedge$$

$(t_{A|B,E} \Leftrightarrow v_A \wedge v_B \wedge v_E) \wedge (t_{\sim A|B,E} \Leftrightarrow v_{\sim A} \wedge v_B \wedge v_E) \wedge \ldots$

If v_A , v_B , v_E are true, so must $t_{A|B,E}$ and vice versa

Probabilistic Inference: Logic and Weights

$V = \{v_A, v_{\sim A}, v_B, v_{\sim B}, v_E, v_{\sim E}\}$

$T = \{t_{A|B,E}\ ,\ t_{\sim A|B,E}\ ,\ t_{A|B,\sim E}\ \ldots\}$

$W(v_{\sim B}) = 0.2,\ W(v_B) = 0.8$ Probabilities of indep events are weights of +ve literals

$W(v_{\sim E}) = 0.1,\ W(v_E) = 0.9$

$W(t_{A|B,E}) = 0.3,\ W(t_{\sim A|B,E}) = 0.7,\ \ldots$ CPT entries are weights of +ve literals

$W(v_A) = W(v_{\sim A}) = 1$ Weights of vars corresponding to dependent events

$W(\neg v_{\sim B}) = W(\neg v_B) = W(\neg\, t_{A|B,E}) \ldots = 1$ Weights of -ve literals are all 1

Weight of assignment $(v_A = 1,\ v_{\sim A} = 0,\ t_{A|B,E} = 1,\ \ldots) = W(v_A) * W(\neg v_{\sim A}) * W(\ t_{A|B,E}) * \ldots$

Product of weights of literals in assignment

Probabilistic Inference: Discrete Integration

$V = \{v_A, v_{\sim A}, v_B, v_{\sim B}, v_E, v_{\sim E}\}$

$T = \{t_{A|B,E}\ ,\ t_{\sim A|B,E}\ ,\ t_{A|B,\sim E}\ \cdots\}$

Formula encoding combination of events in probabilistic model

(Alarm and Earthquake) $F = \varphi_{PGM} \wedge v_A \wedge v_E$

Set of satisfying assignments of F:

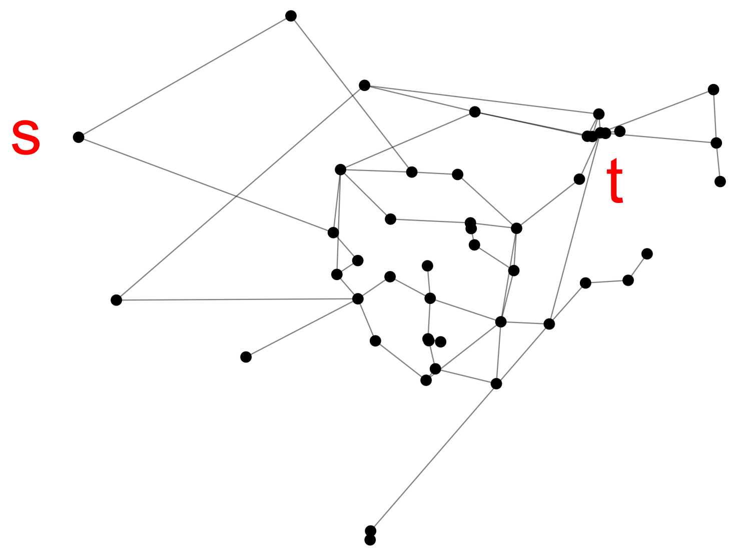
$R_F = \{ (v_A = 1, v_E = 1, v_B = 1, t_{A|B,E} = 1, \text{all else } 0), (v_A = 1, v_E = 1, v_{\sim B} = 1, t_{A|\sim B,E} = 1, \text{all else } 0) \}$

Weight of satisfying assignments of F:

$$W(R_F) = W(v_A) * W(v_E) * W(v_B) * W(t_{A|B,E}) + W(v_A) * W(v_E) * W(v_{\sim B}) * W(t_{A|\sim B,E})$$

$$= 1 * Pr[E] * Pr[B] * Pr[A \mid B,E] + 1 * Pr[E] * Pr[\sim B] * Pr[A \mid \sim B,E] = Pr[A \cap E]$$

Application 2: Network Reliability



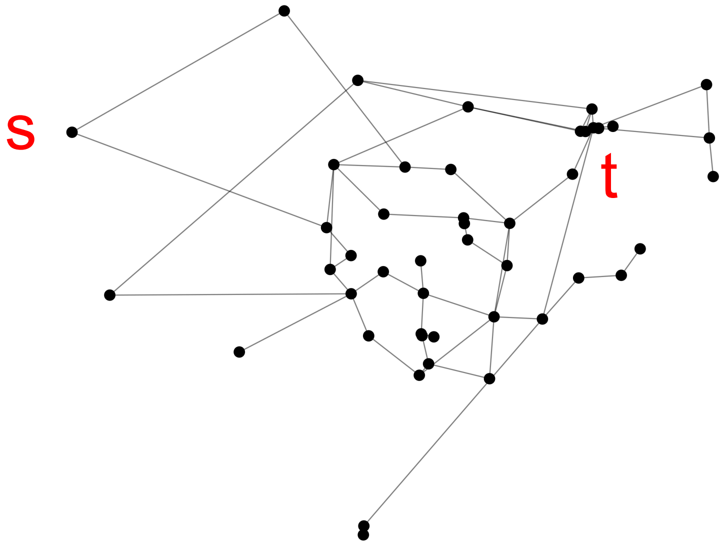
Graph G = (V, E) represents a (power-grid) network

- Nodes (V) are towns, villages, power stations
- Edges (E) are power lines
- Assume each edge e fails with prob $g(e) \in [0,1]$
- Assume failure of edges statistically independent
- What is the probability that **s** and **t** become disconnected?

Network Reliability: First Principles Modeling

$\pi : E \rightarrow \{0, 1\}$ … configuration of network

-- $\pi(e) = 0$ if edge e has failed, 1 otherwise



Prob of network being in configuration π

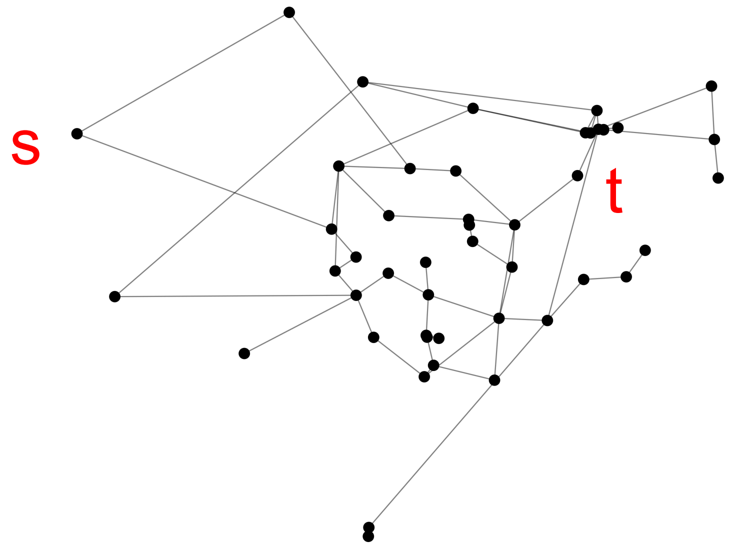
$$\Pr[\pi] = \prod_{e:\ \pi(e) = 0} g(e) \times \prod_{e:\ \pi(e) = 1} (1 - g(e))$$

Prob of s and t being disconnected

$$P^d_{s,t} = \sum_{\pi\ :\ s,\ t \text{ disconnected in } \pi} \Pr[\pi]$$

May need to sum over numerous ($> 2^{100}$) configurations

Network Reliability: Discrete Integration



- p_v: Boolean variable for each v in V
- q_e: Boolean variable for each e in E
- $\varphi_{s,t}$ (p_{v1}, … p_{vn}, q_{e1}, … q_{em}) : Boolean formula such that sat assignments σ of $\varphi_{s,t}$ have 1-1 correspondence with configs π that disconnect s and t
  - $W(\sigma) = \Pr[\pi]$

$$P^d_{s,t} = \sum_{\pi\,:\,s,\,t \text{ disconnected in } \pi} \Pr[\pi] = \sum_{\sigma \models \varphi_{s,t}} W(\sigma) = W(\varphi)$$

Application 3: Quantitative Information Flow

- A password-checker PC takes a secret password (SP) and a user input (UI) and returns “Yes” iff SP = UI [Bang et al 2016]
  - Suppose passwords are 4 characters (‘0’ through ‘9’) long

```
PC1 (char[] SP, char[] UI)  {
 for (int i=0; i<SP.length(); i++) {
   if(SP[i] != UI[i]) return “No”;
 }
 return “Yes”;
}
```

```
PC2 (char[] H, char[] L)  {
 match = true;
 for (int i=0; i<SP.length(); i++) {
   if (SP[i] != UI[i]) match=false;
   else match = match;
 }
 if match return “Yes”;
 else return “No”;
}
```

Which of PC1 and PC2 is more likely to leak information about the secret key through side-channel observations?

QIF: Some Basics

- Program P receives some “high” input (H) and produces a “low” (L) output
  - Password checking: **H is SP**, **L is time taken to answer “Is SP = UI?”**
  - Side-channel observations: memory, time …
- Adversary may infer partial information about H on seeing L
  - E.g. in password checking, infer: **1st char is password is not 9**.
- Can we quantify “leakage of information”?

  “initial uncertainty in H” = “info leaked” + “remaining uncertainty in H” [Smith 2009]

- Uncertainty and information leakage usually quantified using information theoretic measures, e.g. Shannon entropy

QIF: First Principles Approach

- Password checking: Observed time to answer “Yes”/“No”
  - Depends on # instructions executed
- E.g. SP = 00700700

  UI = N2345678, $N \neq 0$

  PC1 executes for loop once

  UI = 02345678

  PC1 executes for loop at least twice

```
PC1 (char[] SP, char[] UI)  {
 for (int i=0; i<SP.length(); i++) {
   if(SP[i] != UI[i]) return “No”;
 }
 return “Yes”;
}
```

Observing time to “No” gives away whether 1st char is not N, $N \neq 0$

In 10 attempts, 1st char can of SP can be uniquely determined.

In max 40 attempts, SP can be cracked.

QIF: First Principles Approach

- Password checking: Observed time to answer “Yes”/“No”
  - Depends on # instructions executed
- E.g. SP = 00700700

  UI = N2345678, $N \neq 0$

  PC1 executes for loop 4 times

  UI = 02345678

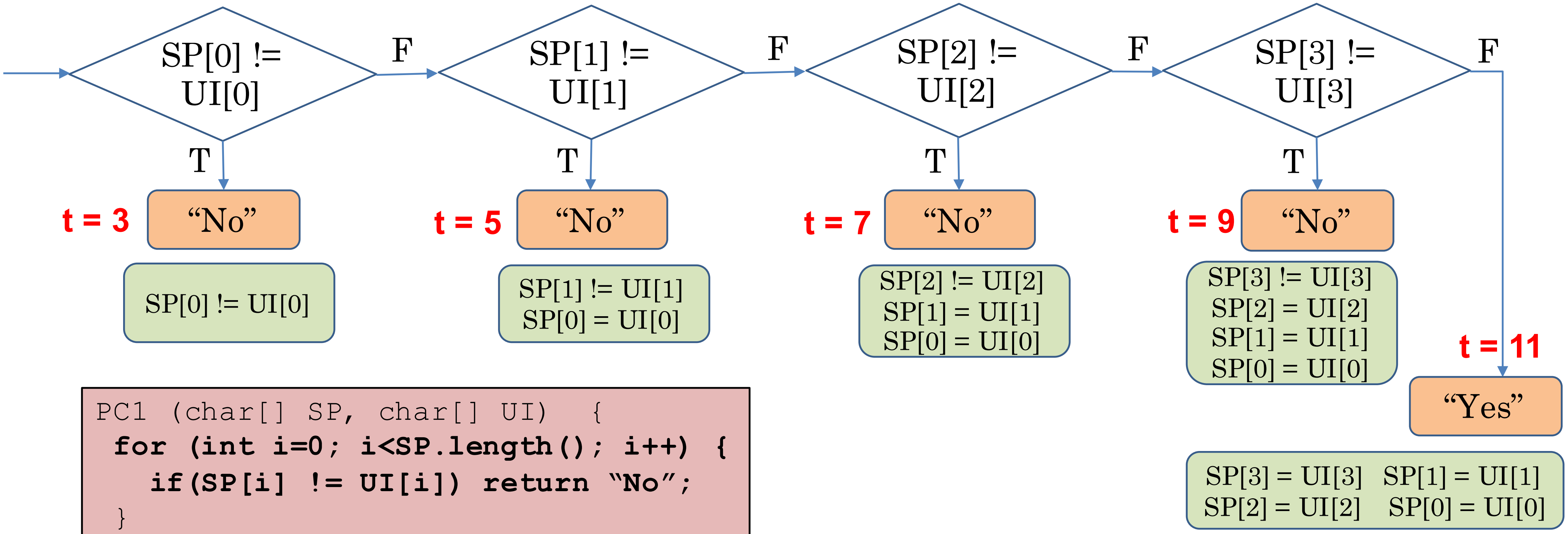
  PC1 executes for loop 4 times

```
PC2 (char[] H, char[] L)  {
 match = true;
 for (int i=0; i<SP.length(); i++) {
   if (SP[i] != UI[i]) match=false;
   else match = match;
 }
 if match return "Yes";
 else return "No";
}
```

Cracking SP requires max 10^4 attempts !!! (“less leakage”)

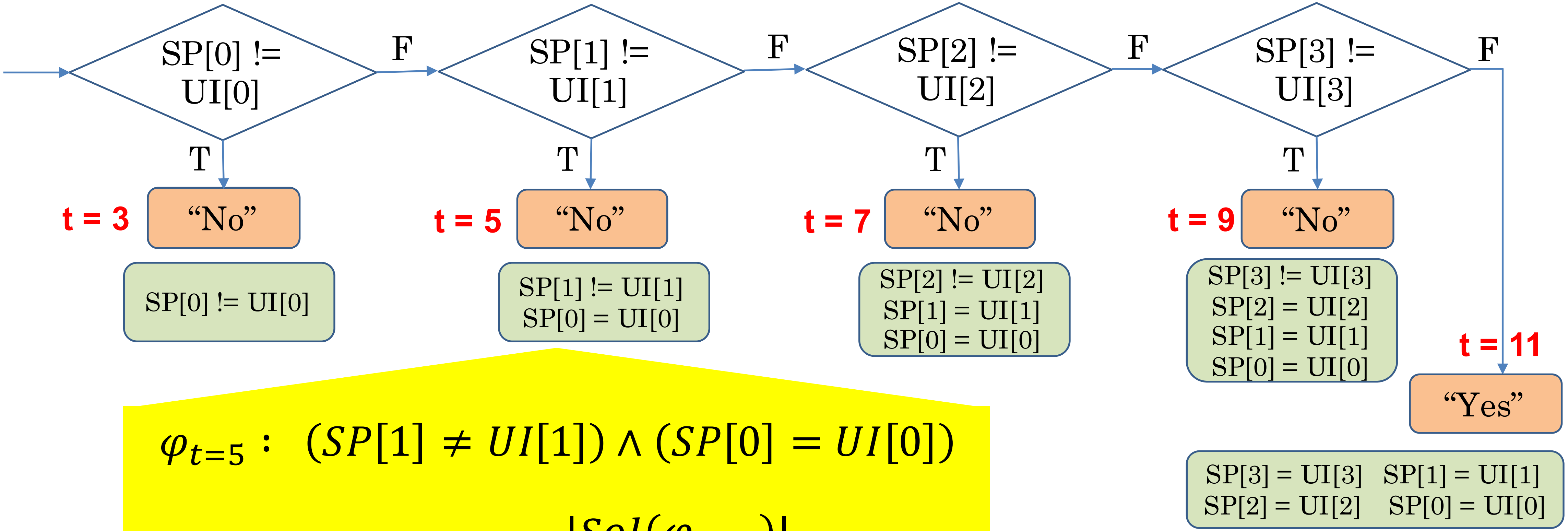
QIF: Partitioning Space of Secret Password

- Observable time effectively partitions values of SP [Bultan2016]




```
PC1 (char[] SP, char[] UI)  {
 for (int i=0; i<SP.length(); i++) {
   if(SP[i] != UI[i]) return "No";
 }
 return "Yes";
}
```

QIF: Probabilities of Observed Times

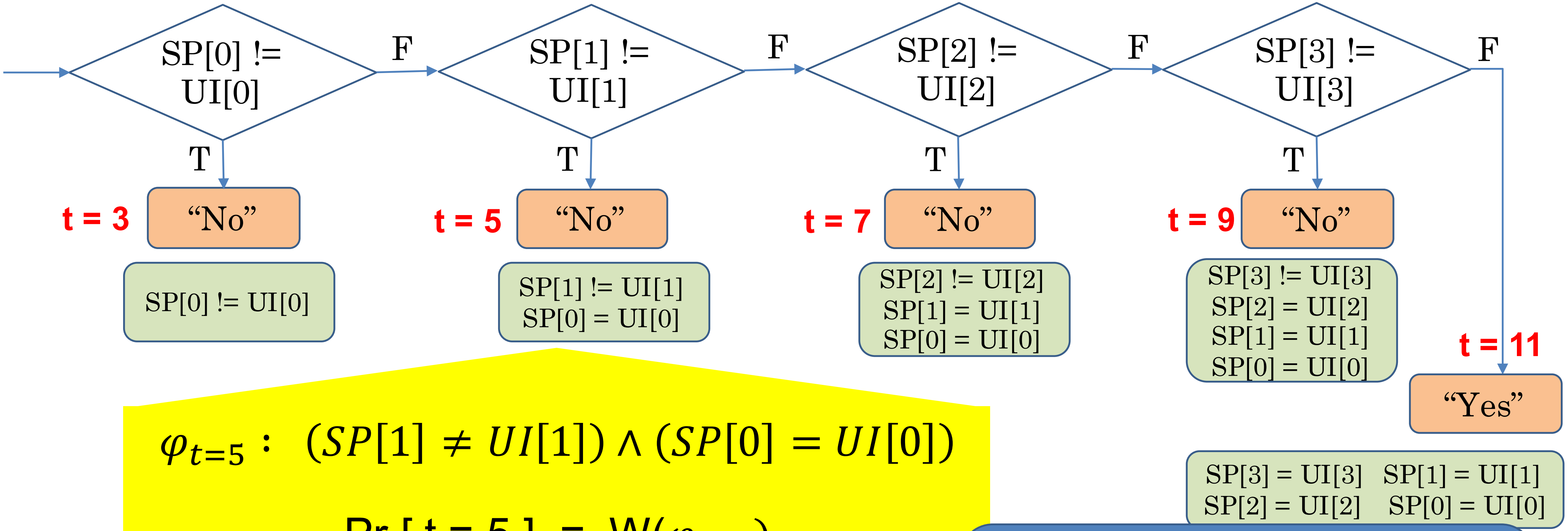


$$\varphi_{t=5} : \quad (SP[1] \neq UI[1]) \wedge (SP[0] = UI[0])$$

$$\Pr[\, t = 5 \,] = \frac{|Sol(\varphi_{t=5})|}{10^4}$$

Model Counting if UI uniformly chosen

QIF: Probabilities of Observed Times



$$\varphi_{t=5} : \ (SP[1] \neq UI[1]) \wedge (SP[0] = UI[0])$$

$$\Pr[\, t = 5 \,] = W(\varphi_{t=5})$$

Discrete Integration if UI chosen according to weight function

QIF: Quantifying Leakage via Integration

- Exp information leakage =
  Shannon entropy of obs times = $\sum_{k \in \{3,5,7,9,11\}} \Pr[t = k] \,.\, \log 1/\Pr[t = k]$

- Information leakage in password checker example
  PC1: 0.52 (more "leaky")
  PC2: 0.0014 (less "leaky")

Discrete integration crucial in obtaining Pr[t = k]

Unweighted Counting Suffices in Principle

Probabilistic Inference

Roth 1996
SBK 2005

Network Reliability

KML 1989, Karger 2000
DMPV 2017

Quantified Information Flow

BKR 2009, NMcCS 2009
KMM 2013, BAPPB 2016

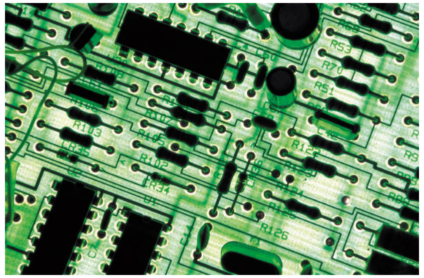
Weighted Model Counting

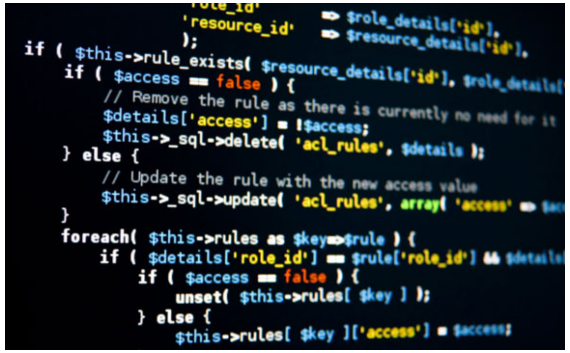
Weighted Model Counting → Unweighted Model Counting

IJCAI 2015

Reduction polynomial in #bits representing weights

Application 4: Constr Random Verification





Functional Verification

- Formal verification
  - Challenges: formal requirements, scalability
  - ~10-15% of verification effort
- Dynamic verification: ***dominant approach***

CRV: Dynamic Verification

- Design is simulated with test vectors
- Test vectors represent different verification scenarios
- Results from simulation compared to intended results

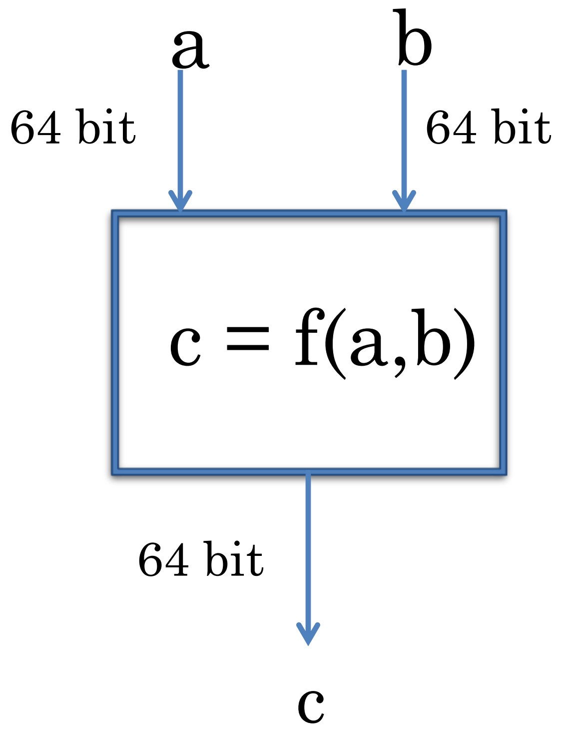
- How do we generate test vectors?

Challenge: Exceedingly large test input space!

Can't try all input combinations

2^{128} combinations for a 64-bit binary operator!!!

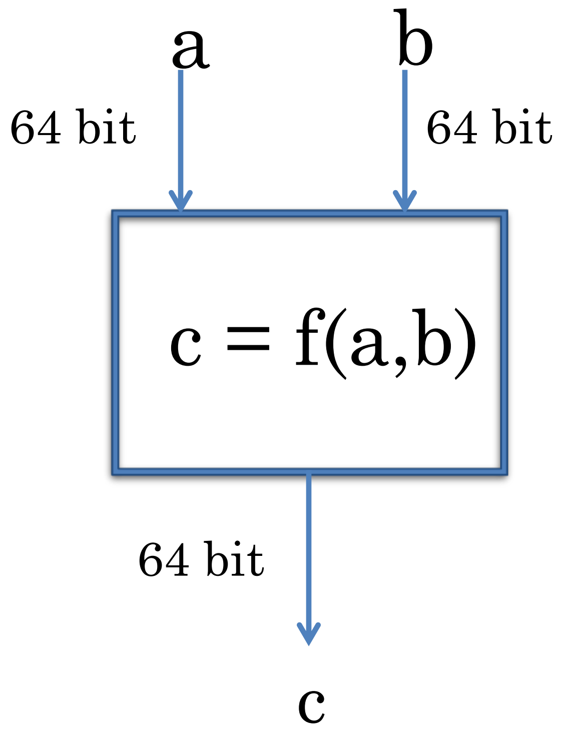
CRV: Sources of Constraints



- Designers:
  1. $a +_{64} 11 *_{32} b = 12$
  2. $a <_{64} (b >> 4)$
- Past Experience:
  1. $40 <_{64} 34 + a <_{64} 5050$
  2. $120 <_{64} b <_{64} 230$
- Users:
  1. $232 *_{32} a + b != 1100$
  2. $1020 <_{64} (b /_{64} 2) +_{64} a <_{64} 2200$

- Test vectors: solutions of constraints

CRV: Why Existing Solvers Don't Suffice



Constraints

- Designers:
  1. $a +_{64} 11 *_{32} b = 12$
  2. $a <_{64} (b >> 4)$
- Past Experience:
  1. $40 <_{64} 34 + a <_{64} 5050$
  2. $120 <_{64} b <_{64} 230$
- Users:
  1. $232 *_{32} a + b != 1100$
  2. $1020 <_{64} (b /_{64} 2) +_{64} a <_{64} 2200$

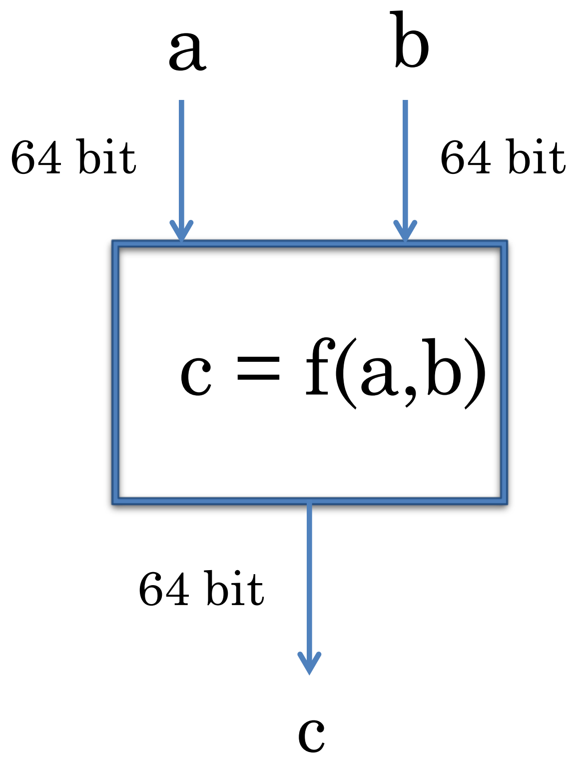
Modern SAT/SMT solvers are complex systems

Efficiency stems from the solver automatically "biasing" search

Fails to give unbiased or user-biased distribution of test vectors

CRV: Need To Go Beyond SAT Solvers

Constrained Random Verification



Set of Constraints

↓

SAT Formula

↓

Sample satisfying assignments uniformly at random

Scalable Uniform Generation of SAT Witnesses

Application 5: Automated Problem Generation

- Large class sizes, MOOC offerings require automated generation of related but randomly different problems
- Discourages plagiarism between students
- Randomness makes it hard for students to guess what the solution would be
- Allows instructors to focus on broad parameters of problems, rather than on individual problem instances
- Enables development of automated intelligent tutoring systems

Auto Prob Gen: Using Problem Templates

- A problem template is a partial specification of a problem
  - "Holes" in the template must be filled with elements from specified sets
  - Constraints on elements chosen to fill various "holes" restricts problem instances so that undesired instances are eliminated
- Example:
  - Non-deterministic finite automata to be generated for complementation

    Holes: States, alphabet size, transitions for (state, letter) pairs, final states, initial states

    Constraints: Alphabet size = 2
    Min/max transitions for a (state, letter) pair = 0/4
    Min/max states = 3/5
    Min/max number of final states = 1/3
    Min/max initial states = 1/2

Auto Prob Gen: An Illustration

- Non-det finite automaton encoded as a formula on following variables

  s_1, s_2, s_3, s_4, s_5 : States
  f_1, f_2, f_3, f_4, f_5: Final states
  n_1, n_2, n_3, n_4, n_5: Initial states
  $s_1a_1s_2, s_1a_2s_2, \ldots$: Transitions

$$\varphi_{\{init\}} = \bigwedge_{\{i\}} (n_i \to s_i) \wedge \left(1 \le \sum_i n_i \le 2\right)$$

$$\varphi_{\{trans\}} = \bigwedge_{\{i\}} (s_i a_j s_k \to s_i \wedge s_k) \wedge \bigwedge_{\{i,j\}} \left(0 \le \sum_k s_i a_j s_k \le 4\right)$$

$$\varphi_{\{stcount\}} = 3 \le \textstyle\sum_i s_i \le 5$$

$$\varphi_{\{finst\}} = \bigwedge_i (f_i \to s_i) \wedge \left(1 \le \sum_i f_i \le 3\right)$$

Every solution of $\varphi_{\{init\}} \wedge \varphi_{\{trans\}} \wedge \varphi_{\{stcount\}} \wedge \varphi_{\{finst\}}$ gives an automaton satisfying specified constraints

Auto Prob Gen: An Illustration

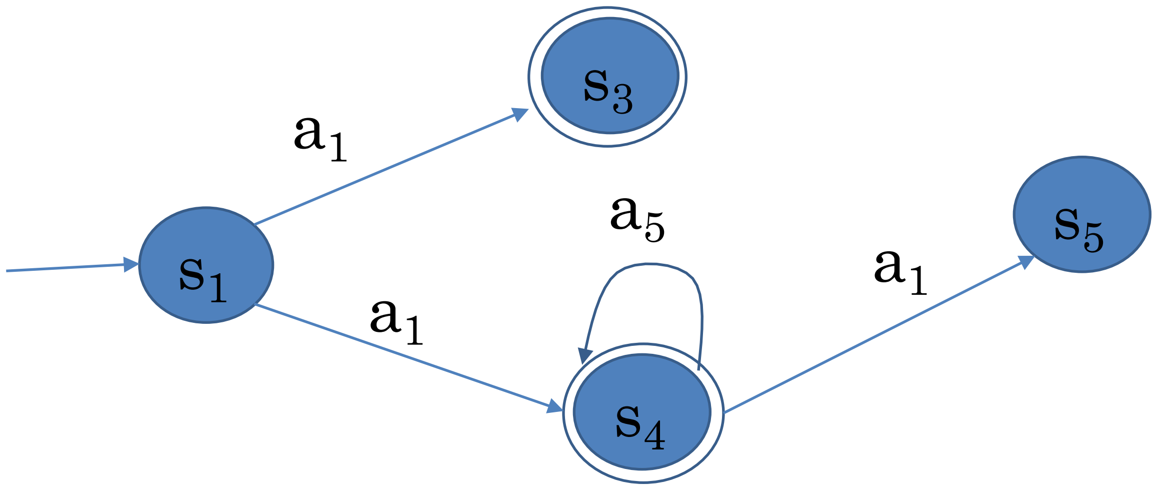
- Non-det finite automaton encoded as a formula on following variables

  $s_1 = 1, s_2 = 0, s_3 = 1, s_4 = 1, s_5 = 1$: States

  $f_1 = 0, f_2 = 0, f_3 = 1, f_4 = 1, f_5 = 0$: Final states

  $n_1 = 1, n_2 = 0, n_3 = 0, n_4 = 0, n_5 = 0$: Initial states

  $s_1a_1s_3 = 1, s_1a_1s_4 = 1, s_4a_2s_4 = 1, s_4a_1s_5 = 1, \ldots$: Transitions



Auto Prob Gen: Discrete Sampling

- Uniform random generation of solutions of constraints gives automata satisfying constraints randomly
- Weighted random generation of solutions gives automata satisfying constraints with different priorities/weightages.

  Examples: Weighing final state variables more gives automata with more final states

  Weighing transitions on letter a_1 more gives automata with more transitions labeled a_1

Discrete Sampling and Integration for the AI Practitioner

Part 2b: Survey of Prior Work

Supratik Chakraborty, IIT Bombay

Kuldeep S. Meel, Rice University

Moshe Y. Vardi, Rice University

How Hard is it to Count/Sample?

- Trivial if we could enumerate R_F: Almost always impractical
- Computational complexity of counting (discrete integration):

Exact unweighted counting: #P-complete [Valiant 1978]

Approximate unweighted counting:

Deterministic: Polynomial time det. Turing Machine with Σ_2^p oracle [Stockmeyer 1983]

$$\frac{|R_F|}{1+\varepsilon} \leq \text{DetEstimate}(\mathrm{F}, \varepsilon) \leq |R_F| \times (1+\varepsilon), \text{ for } \varepsilon > 0$$

Randomized: Poly-time probabilistic Turing Machine with NP oracle

[Stockmeyer 1983; Jerrum,Valiant,Vazirani 1986]

$$\Pr\left[\frac{|R_F|}{1+\varepsilon} \leq \text{RandEstimate}(\mathrm{F}, \varepsilon, \delta) \leq |R_F| \cdot (1+\varepsilon)\right] \geq 1-\delta, \text{ for } \varepsilon > 0,\ 0 < \delta \leq 1$$

Probably Approximately Correct (PAC) algorithm

Weighted versions of counting: Exact: #P-complete [Roth 1996],

Approximate: same class as unweighted version [follows from Roth 1996]

How Hard is it to Count/Sample?

- Computational complexity of sampling:

Uniform sampling: Poly-time prob. Turing Machine with NP oracle [Bellare,Goldreich,Petrank 2000]

$$\Pr[y = \mathrm{UniformGenerator(F)}] = c, \text{ where } \begin{cases} c = 0 \text{ if } y \notin \mathrm{R_F} \\ c > 0 \text{ and indep of } y \text{ if } y \in \mathrm{R_F} \end{cases}$$

Almost uniform sampling: Poly-time prob. Turing Machine with NP oracle [Jerrum,Valiant,Vazirani 1986, also from Bellare,Goldreich,Petrank 2000]

$$\frac{c}{1+\varepsilon} \leq \Pr[y = \mathrm{AUGenerator(F,} \varepsilon)] \leq c \cdot (1+\varepsilon), \text{ where } \begin{cases} c = 0 \text{ if } y \notin \mathrm{R_F} \\ c > 0 \text{ and indep of } y \text{ if } y \in \mathrm{R_F} \end{cases}$$

Pr[Algorithm outputs some y] ≥ ½, if F is satisfiable

Markov Chain Monte Carlo Techniques

- Rich body of theoretical work with applications to sampling and counting [Jerrum,Sinclair 1996]
- Some popular (and intensively studied) algorithms:
  - Metropolis-Hastings [Metropolis et al 1953, Hastings 1970], Simulated Annealing [Kirkpatrick et al 1982]
- High-level idea:
  - Start from a “state” (assignment of variables)
  - Randomly choose next state using “local” biasing functions (depends on target distribution & algorithm parameters)
  - Repeat for an appropriately large number (N) of steps
  - After N steps, samples follow target distribution with high confidence
- Convergence to desired distribution guaranteed only after N (large) steps
- In practice, steps truncated early heuristically

  Nullifies/weakens theoretical guarantees [Kitchen,Keuhlman 2007]

Exact Counters

- DPLL based counters [CDP: Birnbaum,Lozinski 1999]
  - DPLL branching search procedure, with partial truth assignments
  - Once a branch is found satisfiable, if t out of n variables assigned, add 2^{n-t} to model count, backtrack to last decision point, flip decision and continue
  - Requires data structure to check if all clauses are satisfied by partial assignment

    Usually not implemented in modern DPLL SAT solvers
  - Can output a lower bound at any time

Exact Counters

- DPLL + component analysis [RelSat: Bayardo, Pehoushek 2000]
  - Constraint graph G:

    Variables of F are vertices

    An edge connects two vertices if corresponding variables appear in some clause of F
  - Disjoint components of G lazily identified during DPLL search
  - F1, F2, … Fn : subformulas of F corresponding to components

    $|R_F| = |R_{F1}| * |R_{F2}| * |R_{F3}| * \dots$
  - Heuristic optimizations:

    Solve most constrained sub-problems first

    Solving sub-problems in interleaved manner

Exact Counters

- DPLL + Caching [Bacchus et al 2003, Cachet: Sang et al 2004, sharpSAT: Thurley 2006]
  - If same sub-formula revisited multiple times during DPLL search, cache result and re-use it
  - "Signature" of the satisfiable sub-formula/component must be stored
  - Different forms of caching used:
    - Simple sub-formula caching
    - Component caching
    - Linear-space caching
  - Component caching can also be combined with clause learning and other reasoning techniques at each node of DPLL search tree
    - WeightedCachet: DPLL + Caching for weighted assignments

Exact Counters

- Knowledge Compilation based
  - Compile given formula to another form which allows counting models in time polynomial in representation size
  - Reduced Ordered Binary Decision Diagrams (ROBDD) [Bryant 1986]: Construction can blow up exponentially
  - Deterministic Decomposable Negation Normal Form (d-DNNF) [c2d: Darwiche 2004]

    Generalizes ROBDDs; can be significantly more succinct

    Negation normal form with following restrictions:

    Decomposability: All AND operators have arguments with disjoint support

    Determinizability: All OR operators have arguments with disjoint solution sets
  - Sentential Decision Diagrams (SDD) [Darwiche 2011]

Exact Counters: How far do they go?

- Work reasonably well in small-medium sized problems, and in large problem instances with special structure
- Use them whenever possible
  - #P-completeness hits back eventually – scalability suffers!

Bounding Counters

[MBound: Gomes et al 2006; SampleCount: Gomes et al 2007; BPCount: Kroc et al 2008]

- Provide lower and/or upper bounds of model count
- Usually more efficient than exact counters
- No approximation guarantees on bounds
  - Useful only for limited applications

Hashing-based Sampling

- Bellare, Goldreich, Petrank (BGP 2000)
  - Uniform generator for SAT witnesses:
    - Polynomial time randomized algorithm with access to an NP oracle

$$\Pr[y = \mathrm{BGP}(\mathrm{F})] = \begin{cases} 0 \text{ if } y \notin \mathrm{R_F} \\ c\ (>0) \text{ if } y \in \mathrm{R_F}, \text{ where } c \text{ is independent of } y \end{cases}$$

  - Employs n-**universal hash functions**
    - Works well for small values of n
    - For high dimensions (large n), significant computational overheads

Much more on this coming in Part 3

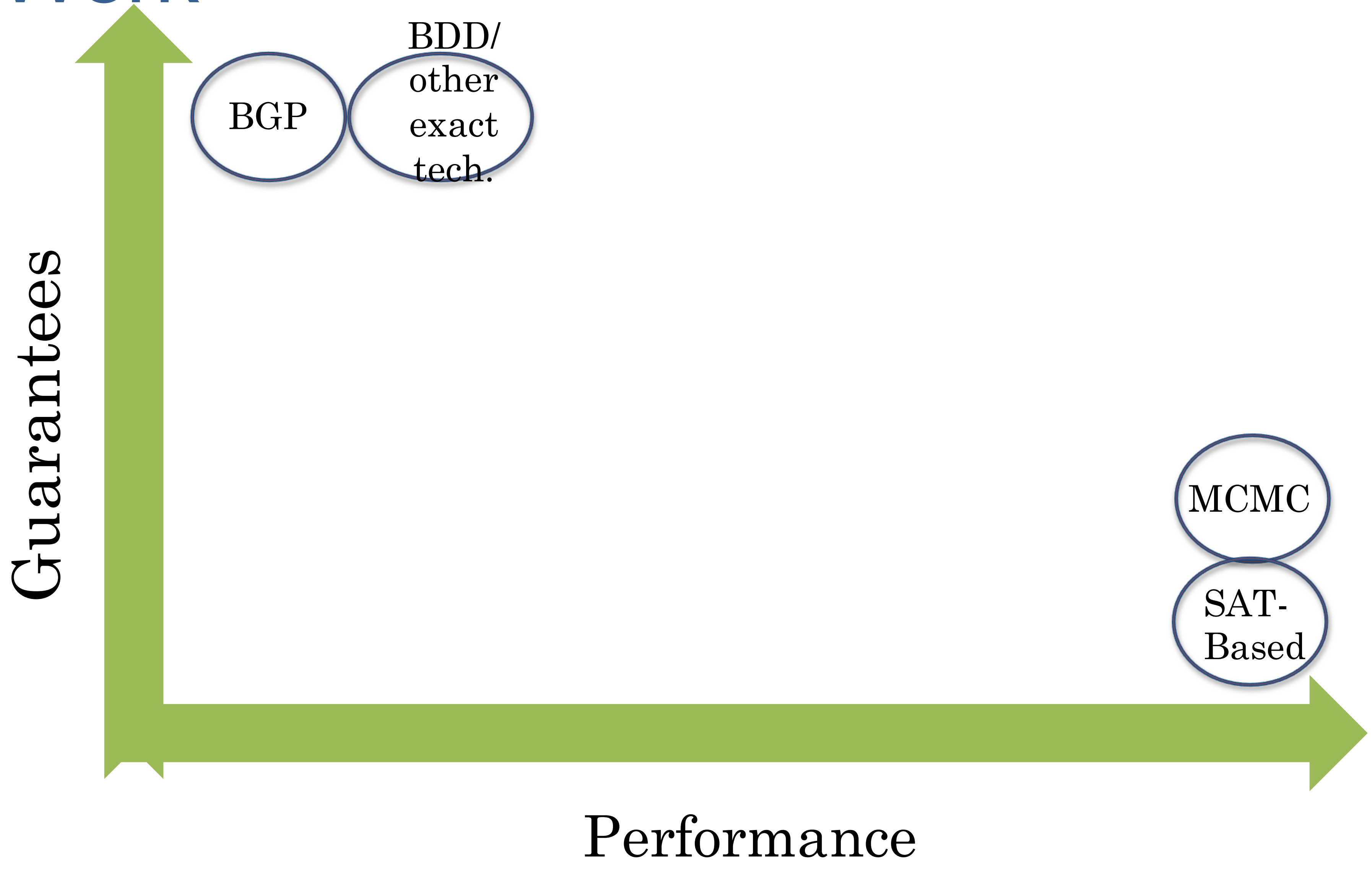
Approximate Integration and Sampling: Close Cousins

- Seminal paper by Jerrum, Valiant, Vazirani 1986

Almost-Uniform Generator $\xleftrightarrow[\text{reduction}]{\text{Polynomial}}$ PAC Counter

- Yet, no practical algorithms that scale to large problem instances were derived from this work
  - No scalable PAC counter or almost-uniform generator existed until a few years back
  - The inter-reductions are practically computation intensive
    - Think of O(n) calls to the counter when n = 100000

Prior Work


BGP
BDD/
other
exact
tech.
MCMC
SAT-
Based
Guarantees
Performance

Techniques using XOR hash functions

- Bounding counters MBound, SampleCount [Gomes et al. 2006, Gomes et al 2007] used random XORs
  - Algorithms geared towards finding bounds without approximation guarantees
  - Power of 2-universal hashing not exploited
- In a series of papers [2013: ICML, UAI, NIPS; 2014: ICML; 2015: ICML, UAI; 2016: AAAI, ICML, AISTATS, …] Ermon et al used XOR hash functions for discrete counting/sampling
  - Random XORs, also XOR constraints with specific structures
  - 2-universality exploited to provide improved guarantees
  - Relaxed constraints (like short XORs) and their effects studied

An Interesting Combination: XOR + MAP Optimization

- WISH: Ermon et al 2013
- Given a weight function $W: \{0,1\}^n \rightarrow \Re^{\geq 0}$
  - Use random XORs to partition solutions into cells
  - After partitioning into 2, 4, 8, 16, … cells
    Use **Max Aposteriori Probability (MAP)** optimizer to find solution with max weight in a cell (say, a_2, a_4, a_8, a_{16}, …)
  - Estimated $W(R_F) = W(a_2)*1 + W(a_4)*2 + W(a_8)*4 + \ldots$
- Constant factor approximation of $W(R_F)$ with high confidence
- MAP oracle needs repeated invokation $O(n.\log_2 n)$
  - MAP is NP-complete
  - Being optimization (not decision) problem), MAP is harder to solve in practice than SAT

XOR-based Counting and Sampling

- Remainder of tutorial
  - Deeper dive into XOR hash-based counting and sampling
  - Discuss theoretical aspects and experimental observations

  - Based on work published in [2013: CP, CAV; 2014: DAC, AAAI; 2015: IJCAI, TACAS; 2016: AAAI, IJCAI, 2017: AAAI]

Discrete Sampling and Integration for the AI Practitioner
Part III: Hashing-based Approach to Sampling and Integration

Supratik Chakraborty, IIT Bombay
Kuldeep S. Meel, Rice University
Moshe Y. Vardi, Rice University

Discrete Integration and Sampling

- Given
  - Variables $X_1, X_2, \cdots X_n$ over finite discrete domains $D_1, D_2, \cdots D_n$
  - Formula φ over $X_1, X_2, \cdots X_n$
  - Weight Function W: $D_1 \times D_2 \cdots \times D_n \mapsto [0,1]$
- $\mathsf{Sol}(\varphi) = \{$solutions of F$\}$
- Discrete Integration: Determine $W(\varphi) = \Sigma_{y \in \mathsf{Sol}(\varphi)} W(y)$
  - If $W(y) = 1$ for all y, then $W(\varphi) = |\mathsf{Sol}(\varphi)|$
- Discrete Sampling: Randomly sample from $\mathsf{Sol}(\varphi)$ such that Pr[y is sampled] $\propto W(y)$
  - If $W(y) = 1$ for all y, then uniformly sample from $\mathsf{Sol}(\varphi)$

Part I

Discrete Integration

From Weighted to Unweighted Integration

Boolean Formula φ and weight function $W : \{0,1\}^n \rightarrow \mathbb{Q}^{\geq 0}$

From Weighted to Unweighted Integration

Boolean Formula φ and weight function $W : \{0,1\}^n \to \mathbb{Q}^{\geq 0}$ ➡ Boolean Formula F'

$$W(\varphi) = c(W) \times |\mathsf{Sol}(F')|$$

From Weighted to Unweighted Integration

Boolean Formula φ and weight function $W : \{0,1\}^n \to \mathbb{Q}^{\geq 0}$ ⇒ Boolean Formula F'

$$W(\varphi) = c(W) \times |\mathsf{Sol}(F')|$$

- Key Idea: Encode weight function as a set of constraints

(CFMV, IJCAI15)

From Weighted to Unweighted Integration

Boolean Formula φ and weight function $W : \{0,1\}^n \to \mathbb{Q}^{\geq 0}$ ⇨ Boolean Formula F'

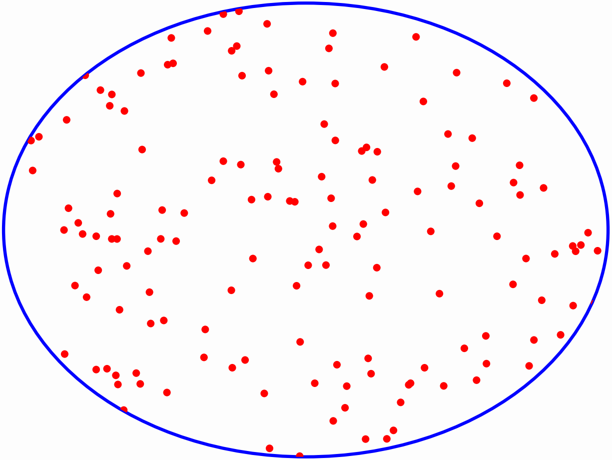
$$W(\varphi) = c(W) \times |\mathsf{Sol}(F')|$$

- Key Idea: Encode weight function as a set of constraints

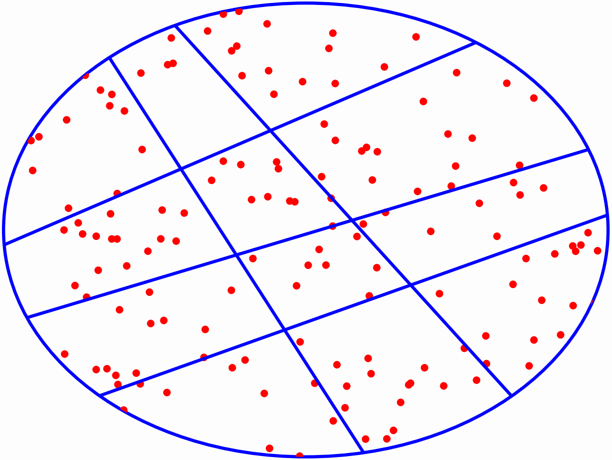
(CFMV, IJCAI15)

How do we estimate $|\mathsf{Sol}(F')|$?

As Simple as Counting Dots

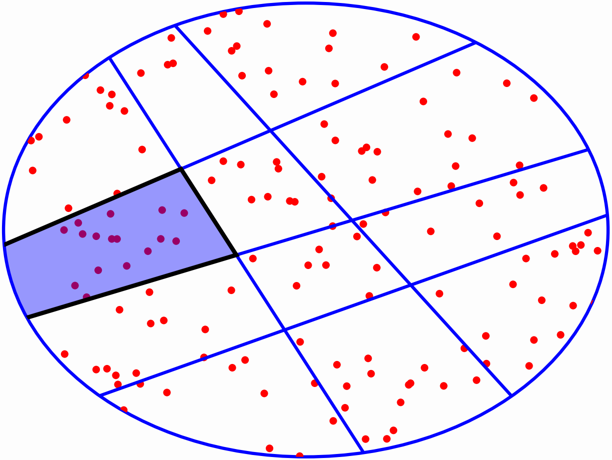


As Simple as Counting Dots



As Simple as Counting Dots

Pick a random cell



Estimate = Number of solutions in a cell × Number of cells

Challenges

Challenge 1 How to partition into roughly equal small cells of solutions without knowing the distribution of solutions?

Challenges

Challenge 1 How to partition into roughly equal small cells of solutions without knowing the distribution of solutions?

Challenge 2 How large is a "small" cell?

Challenges

Challenge 1 How to partition into roughly equal small cells of solutions without knowing the distribution of solutions?

Challenge 2 How large is a "small" cell?

Challenge 3 How many cells?

Challenges

Challenge 1 How to partition into roughly equal small cells of solutions without knowing the distribution of solutions?

- Designing function h : assignments $\rightarrow$ cells (hashing)
- Solutions in a cell α: $\mathsf{Sol}(\varphi) \cap \{y \mid h(y) = \alpha\}$

Challenges

Challenge 1 How to partition into roughly equal small cells of solutions without knowing the distribution of solutions?

- Designing function h : assignments $\rightarrow$ cells (hashing)
- Solutions in a cell α: $\mathsf{Sol}(\varphi) \cap \{y \mid h(y) = \alpha\}$
- Deterministic h unlikely to work

Challenges

Challenge 1 How to partition into roughly equal small cells of solutions without knowing the distribution of solutions?

- Designing function h : assignments $\rightarrow$ cells (hashing)
- Solutions in a cell α: $\mathsf{Sol}(\varphi) \cap \{y \mid h(y) = \alpha\}$
- Deterministic h unlikely to work
- Choose h randomly from a large family H of hash functions

  Universal Hashing (Carter and Wegman 1977)

r-Universal Hashing

- Let H be family of $r-$universal hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$

$$\forall y_1, y_2, \cdots y_r \in \{0,1\}^n, \alpha_1, \alpha_2, \cdots \alpha_r \in \{0,1\}^m, h \xleftarrow{R} H$$

$$\Pr[h(y_1) = \alpha_1] = \cdots \Pr[h(y_r) = \alpha_r] = \left(\frac{1}{2^m}\right)$$

$$\Pr[h(y_1) = \alpha_1 \wedge \cdots \wedge h(y_r) = \alpha_r] = \left(\frac{1}{2^m}\right)^r$$

Desired Properties

- Let h be randomly picked a family of hash function H and Z be the number of solutions in a randomly chosen cell α
  - What is $\mathrm{E}[Z]$ and how much does Z deviate from $\mathrm{E}[Z]$?
- For every $y \in \mathrm{Sol}(\varphi)$, we define $I_y = \begin{cases} 1 & h(y) = \alpha (\text{y is in cell}) \\ 0 & \text{otherwise} \end{cases}$
- $Z = \sum_{y \in \mathrm{Sol}(\varphi)} I_y$
  - Desired: $\mathrm{E}[Z] = \frac{|\mathrm{Sol}(\varphi)|}{2^m}$ and $\sigma^2[Z] \leq \mathrm{E}[Z]$

Desired Properties

- Let h be randomly picked a family of hash function H and Z be the number of solutions in a randomly chosen cell α
  - What is $\mathrm{E}[Z]$ and how much does Z deviate from $\mathrm{E}[Z]$?
- For every $y \in \mathrm{Sol}(\varphi)$, we define $I_y = \begin{cases} 1 & h(y) = \alpha (\text{y is in cell}) \\ 0 & \text{otherwise} \end{cases}$
- $Z = \sum_{y \in \mathrm{Sol}(\varphi)} I_y$
  - Desired: $\mathrm{E}[Z] = \frac{|\mathrm{Sol}(\varphi)|}{2^m}$ and $\sigma^2[Z] \le \mathrm{E}[Z]$
  - It suffices to have H to be 2-universal

Desired Properties

- Let h be randomly picked a family of hash function H and Z be the number of solutions in a randomly chosen cell α
  - What is $\mathsf{E}[Z]$ and how much does Z deviate from $\mathsf{E}[Z]$?
- For every $y \in \mathsf{Sol}(\varphi)$, we define $I_y = \begin{cases} 1 & h(y) = \alpha (\text{y is in cell}) \\ 0 & \text{otherwise} \end{cases}$
- $Z = \sum_{y \in \mathsf{Sol}(\varphi)} I_y$
  - Desired: $\mathsf{E}[Z] = \frac{|\mathsf{Sol}(\varphi)|}{2^m}$ and $\sigma^2[Z] \leq \mathsf{E}[Z]$
  - It suffices to have H to be 2-universal
  - $\Pr\left[\frac{\mathsf{E}[Z]}{1+\varepsilon} \leq Z \leq \mathsf{E}[Z](1+\varepsilon)\right] \geq 1 - \frac{\sigma^2[Z]}{(\frac{\varepsilon}{1+\varepsilon})^2(\mathsf{E}[Z])^2}$

Desired Properties

- Let h be randomly picked a family of hash function H and Z be the number of solutions in a randomly chosen cell α
  - What is $\mathsf{E}[Z]$ and how much does Z deviate from $\mathsf{E}[Z]$?
- For every $y \in \mathsf{Sol}(\varphi)$, we define $I_y = \begin{cases} 1 & h(y) = \alpha (\text{y is in cell}) \\ 0 & \text{otherwise} \end{cases}$
- $Z = \sum_{y \in \mathsf{Sol}(\varphi)} I_y$
  - Desired: $\mathsf{E}[Z] = \frac{|\mathsf{Sol}(\varphi)|}{2^m}$ and $\sigma^2[Z] \leq \mathsf{E}[Z]$
  - It suffices to have H to be 2-universal
  - $\Pr\left[\frac{\mathsf{E}[Z]}{1+\varepsilon} \leq Z \leq \mathsf{E}[Z](1+\varepsilon)\right] \geq 1 - \frac{\sigma^2[Z]}{(\frac{\varepsilon}{1+\varepsilon})^2(\mathsf{E}[Z])^2} \geq 1 - \frac{1}{(\frac{\varepsilon}{1+\varepsilon})^2(\mathsf{E}[Z])}$

Desired Properties

- Let h be randomly picked a family of hash function H and Z be the number of solutions in a randomly chosen cell α
  - What is $\mathsf{E}[Z]$ and how much does Z deviate from $\mathsf{E}[Z]$?
- For every $y \in \mathsf{Sol}(\varphi)$, we define $I_y = \begin{cases} 1 & h(y) = \alpha (\text{y is in cell}) \\ 0 & \text{otherwise} \end{cases}$
- $Z = \sum_{y \in \mathsf{Sol}(\varphi)} I_y$
  - Desired: $\mathsf{E}[Z] = \frac{|\mathsf{Sol}(\varphi)|}{2^m}$ and $\sigma^2[Z] \leq \mathsf{E}[Z]$
  - It suffices to have H to be 2-universal
  - $\Pr\left[\frac{\mathsf{E}[Z]}{1+\varepsilon} \leq Z \leq \mathsf{E}[Z](1+\varepsilon)\right] \geq 1 - \frac{\sigma^2[Z]}{(\frac{\varepsilon}{1+\varepsilon})^2(\mathsf{E}[Z])^2} \geq 1 - \frac{1}{(\frac{\varepsilon}{1+\varepsilon})^2(\mathsf{E}[Z])}$

2-Universal Hash Functions

- Variables: $X_1, X_2, \cdots X_n$
- To construct $h : \{0,1\}^n \rightarrow \{0,1\}^m$, choose m random XORs
- Pick every X_i with prob. $\frac{1}{2}$ and XOR them
  - $X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2} \oplus 1$
  - Expected size of each XOR: $\frac{n}{2}$

2-Universal Hash Functions

- Variables: $X_1, X_2, \cdots X_n$
- To construct $h : \{0,1\}^n \rightarrow \{0,1\}^m$, choose m random XORs
- Pick every X_i with prob. $\frac{1}{2}$ and XOR them
  - $X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2} \oplus 1$
  - Expected size of each XOR: $\frac{n}{2}$
- To choose $\alpha \in \{0,1\}^m$, set every XOR equation to 0 or 1 randomly

$$X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2} \oplus 1 = 0 \qquad (Q_1)$$
$$X_2 \oplus X_5 \oplus X_6 \cdots \oplus X_{n-1} \oplus 1 = 1 \qquad (Q_2)$$
$$\cdots \qquad (\cdots)$$
$$X_1 \oplus X_2 \oplus X_5 \cdots \oplus X_{n-2} = 1 \qquad (Q_m)$$

- Solutions in a cell: $F \wedge Q_1 \cdots \wedge Q_m$

2-Universal Hash Functions

- Variables: $X_1, X_2, \cdots X_n$
- To construct $h : \{0,1\}^n \rightarrow \{0,1\}^m$, choose m random XORs
- Pick every X_i with prob. $\frac{1}{2}$ and XOR them
  - $X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2} \oplus 1$
  - Expected size of each XOR: $\frac{n}{2}$
- To choose $\alpha \in \{0,1\}^m$, set every XOR equation to 0 or 1 randomly

$$X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2} \oplus 1 = 0 \qquad (Q_1)$$
$$X_2 \oplus X_5 \oplus X_6 \cdots \oplus X_{n-1} \oplus 1 = 1 \qquad (Q_2)$$
$$\cdots \qquad (\cdots)$$
$$X_1 \oplus X_2 \oplus X_5 \cdots \oplus X_{n-2} = 1 \qquad (Q_m)$$

- Solutions in a cell: $F \wedge Q_1 \cdots \wedge Q_m$
- Finding a solution is NP-complete

2-Universal Hash Functions

- Variables: $X_1, X_2, \cdots X_n$
- To construct $h : \{0,1\}^n \rightarrow \{0,1\}^m$, choose m random XORs
- Pick every X_i with prob. $\frac{1}{2}$ and XOR them
  - $X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2} \oplus 1$
  - Expected size of each XOR: $\frac{n}{2}$
- To choose $\alpha \in \{0,1\}^m$, set every XOR equation to 0 or 1 randomly

$$X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2} \oplus 1 = 0 \quad (Q_1)$$
$$X_2 \oplus X_5 \oplus X_6 \cdots \oplus X_{n-1} \oplus 1 = 1 \quad (Q_2)$$
$$\cdots \quad (\cdots)$$
$$X_1 \oplus X_2 \oplus X_5 \cdots \oplus X_{n-2} = 1 \quad (Q_m)$$

- Solutions in a cell: $F \wedge Q_1 \cdots \wedge Q_m$
- Finding a solution is NP-complete
  Modern SAT solvers are able to deal routinely with practical problems that involve many thousands of variables, although such problems were regarded as hopeless just a few years ago.
  (Knuth, 2016)

2-Universal Hash Functions

- Variables: $X_1, X_2, \cdots X_n$
- To construct $h : \{0,1\}^n \rightarrow \{0,1\}^m$, choose m random XORs
- Pick every X_i with prob. $\frac{1}{2}$ and XOR them
  - $X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2} \oplus 1$
  - Expected size of each XOR: $\frac{n}{2}$
- To choose $\alpha \in \{0,1\}^m$, set every XOR equation to 0 or 1 randomly

$$X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2} \oplus 1 = 0 \tag{Q_1}$$
$$X_2 \oplus X_5 \oplus X_6 \cdots \oplus X_{n-1} \oplus 1 = 1 \tag{Q_2}$$
$$\cdots \tag{$\cdots$}$$
$$X_1 \oplus X_2 \oplus X_5 \cdots \oplus X_{n-2} = 1 \tag{Q_m}$$

- Solutions in a cell: $F \wedge Q_1 \cdots \wedge Q_m$
- Finding a solution is NP-complete
- Performance of state of the art SAT solvers degrade with increase in the size of XORs (SAT Solvers != SAT oracles)

Improved Universal Hash Functions

- Not all variables are required to specify solution space of φ
  - $F := X_3 \iff (X_1 \vee X_2)$
  - X_1 and X_2 uniquely determines rest of the variables (i.e., X_3)
- Formally: if I is independent support, then $\forall \sigma_1, \sigma_2 \in \mathsf{Sol}(\varphi)$, if σ_1 and σ_2 agree on I then $\sigma_1 = \sigma_2$
  - $\{X_1, X_2\}$ is independent support but $\{X_1, X_3\}$ is not

Improved Universal Hash Functions

- Not all variables are required to specify solution space of φ
  - $F := X_3 \iff (X_1 \vee X_2)$
  - X_1 and X_2 uniquely determines rest of the variables (i.e., X_3)
- Formally: if I is independent support, then $\forall \sigma_1, \sigma_2 \in \mathsf{Sol}(\varphi)$, if σ_1 and σ_2 agree on I then $\sigma_1 = \sigma_2$
  - $\{X_1, X_2\}$ is independent support but $\{X_1, X_3\}$ is not
- Random XORs need to be constructed only over I (CMV DAC14)

Improved Universal Hash Functions

- Not all variables are required to specify solution space of φ
  - $F := X_3 \iff (X_1 \vee X_2)$
  - X_1 and X_2 uniquely determines rest of the variables (i.e., X_3)
- Formally: if I is independent support, then $\forall \sigma_1, \sigma_2 \in \mathsf{Sol}(\varphi)$, if σ_1 and σ_2 agree on I then $\sigma_1 = \sigma_2$
  - $\{X_1, X_2\}$ is independent support but $\{X_1, X_3\}$ is not
- Random XORs need to be constructed only over I (CMV DAC14)
- Typically I is 1-2 orders of magnitude smaller than X
- Auxiliary variables introduced during encoding phase are *dependent* (Tseitin 1968)

Improved Universal Hash Functions

- Not all variables are required to specify solution space of φ
  - $F := X_3 \iff (X_1 \vee X_2)$
  - X_1 and X_2 uniquely determines rest of the variables (i.e., X_3)
- Formally: if I is independent support, then $\forall \sigma_1, \sigma_2 \in \mathsf{Sol}(\varphi)$, if σ_1 and σ_2 agree on I then $\sigma_1 = \sigma_2$
  - $\{X_1, X_2\}$ is independent support but $\{X_1, X_3\}$ is not
- Random XORs need to be constructed only over I (CMV DAC14)
- Typically I is 1-2 orders of magnitude smaller than X
- Auxiliary variables introduced during encoding phase are *dependent* (Tseitin 1968)

Algorithmic procedure to determine I?

Independent Support

- $I \subseteq X$ is an independent support:
  $\forall \sigma_1, \sigma_2 \in \mathsf{Sol}(\varphi)$, σ_1 and σ_2 agree on I then $\sigma_1 = \sigma_2$

Independent Support

- $I \subseteq X$ is an independent support:
  $\forall \sigma_1, \sigma_2 \in \mathsf{Sol}(\varphi)$, σ_1 and σ_2 agree on I then $\sigma_1 = \sigma_2$
- $F(x_1, \cdots x_n) \wedge F(y_1, \cdots y_n) \wedge \bigwedge_{i | x_i \in I} (x_i = y_i) \implies \bigwedge_i (x_i = y_i)$
  where $F(y_1, \cdots y_n) := F(x_1 \rightarrowtail y_1, \cdots x_n \rightarrowtail y_n)$

Independent Support

- $I \subseteq X$ is an independent support:
  $\forall \sigma_1, \sigma_2 \in \mathrm{Sol}(\varphi)$, σ_1 and σ_2 agree on I then $\sigma_1 = \sigma_2$
- $F(x_1, \cdots x_n) \wedge F(y_1, \cdots y_n) \wedge \bigwedge_{i | x_i \in I}(x_i = y_i) \implies \bigwedge_i (x_i = y_i)$
  where $F(y_1, \cdots y_n) := F(x_1 \rightarrowtail y_1, \cdots x_n \rightarrowtail y_n)$
- $Q_{F,I} := F(x_1, \cdots x_n) \wedge F(y_1, \cdots y_n) \wedge \bigwedge_{i | x_i \in I}(x_i = y_i) \wedge \neg(\bigwedge_i (x_i = y_i))$

Independent Support

- $I \subseteq X$ is an independent support:
  $\forall \sigma_1, \sigma_2 \in \mathsf{Sol}(\varphi)$, σ_1 and σ_2 agree on I then $\sigma_1 = \sigma_2$
- $F(x_1, \cdots x_n) \wedge F(y_1, \cdots y_n) \wedge \bigwedge_{i | x_i \in I} (x_i = y_i) \implies \bigwedge_i (x_i = y_i)$
  where $F(y_1, \cdots y_n) := F(x_1 \rightarrowtail y_1, \cdots x_n \rightarrowtail y_n)$
- $Q_{F,I} := F(x_1, \cdots x_n) \wedge F(y_1, \cdots y_n) \wedge \bigwedge_{i | x_i \in I} (x_i = y_i) \wedge \neg (\bigwedge_i (x_i = y_i))$
- Lemma: $Q_{F,I}$ is UNSAT if and only if I is independent support

Independent Support

$$H_1 := \{x_1 = y_1\}, H_2 := \{x_2 = y_2\}, \cdots H_n := \{x_n = y_n\}$$

$$\Omega = F(x_1, \cdots x_n) \wedge F(y_1, \cdots y_n) \wedge \neg(\bigwedge_i (x_i = y_i))$$

Lemma

$I = \{x_i\}$ is independent support iif $H^I \wedge \Omega$ is UNSAT where $H^I = \{H_i | x_i \in I\}$

Minimal Unsatisfiable Subset

Given $\Psi = H_1 \wedge H_2 \cdots \wedge H_m \wedge \Omega$

Unsatisfiable Subset Find subset $\{H_{i1}, H_{i2}, \cdots H_{ik}\}$ of $\{H_1, H_2, \cdots H_m\}$ such that $H_{i1} \wedge H_{i2} \wedge H_{ik} \wedge \Omega$ is UNSAT

Minimal Unsatisfiable Subset

Given $\Psi = H_1 \wedge H_2 \cdots \wedge H_m \wedge \Omega$

Unsatisfiable Subset Find subset $\{H_{i1}, H_{i2}, \cdots H_{ik}\}$ of $\{H_1, H_2, \cdots H_m\}$ such that $H_{i1} \wedge H_{i2} \wedge H_{ik} \wedge \Omega$ is UNSAT

Minimal Unsatisfiable Subset Find **minimal** subset $\{H_{i1}, H_{i2}, \cdots H_{ik}\}$ of $\{H_1, H_2, \cdots H_m\}$ such that $H_{i1} \wedge H_{i2} \wedge H_{ik} \wedge \Omega$ is UNSAT

Minimal Unsatisfiable Subset

Given $\Psi = H_1 \wedge H_2 \cdots \wedge H_m \wedge \Omega$

Unsatisfiable Subset Find subset $\{H_{i1}, H_{i2}, \cdots H_{ik}\}$ of $\{H_1, H_2, \cdots H_m\}$ such that $H_{i1} \wedge H_{i2} \wedge H_{ik} \wedge \Omega$ is UNSAT

Minimal Unsatisfiable Subset Find **minimal** subset $\{H_{i1}, H_{i2}, \cdots H_{ik}\}$ of $\{H_1, H_2, \cdots H_m\}$ such that $H_{i1} \wedge H_{i2} \wedge H_{ik} \wedge \Omega$ is UNSAT

Minimal Independent Support

$$H_1 := \{x_1 = y_1\}, H_2 := \{x_2 = y_2\}, \cdots H_n := \{x_n = y_n\}$$

$$\Omega = F(x_1, \cdots x_n) \wedge F(y_1, \cdots y_n) \wedge \neg(\bigwedge_i (x_i = y_i))$$

Lemma

$I = \{x_i\}$ is Minimal Independent Support iif H^I is Minimal Unsatisfiable Subset where $H^I = \{H_i | x_i \in I\}$

MIS ⇨ MUS

Minimal Independent Support

$$H_1 := \{x_1 = y_1\}, H_2 := \{x_2 = y_2\}, \cdots H_n := \{x_n = y_n\}$$

$$\Omega = F(x_1, \cdots x_n) \wedge F(y_1, \cdots y_n) \wedge \neg(\bigwedge_i (x_i = y_i))$$

Lemma

$I = \{x_i\}$ is Minimal Independent Support iif H^I is Minimal Unsatisfiable Subset where $H^I = \{H_i | x_i \in I\}$

MIS ⇨ MUS

Two orders of magnitude improvement in runtime

Challenges

Challenge 1 How to partition into roughly equal small cells of solutions without knowing the distribution of solutions?

- Independent Support-based 2-Universal Hash Functions

Challenge 2 **How large is a "small" cell?**

Challenge 3 How many cells?

Challenge 2: How large is a “small” cell

- Too large $\rightarrowtail$ Hard to enumerate

Challenge 2: How large is a “small” cell

- Too large $\rightarrowtail$ Hard to enumerate
- Too small $\rightarrowtail$ Weaker probabilistic guarantees

Challenge 2: How large is a "small" cell

- Too large $\rightarrowtail$ Hard to enumerate
- Too small $\rightarrowtail$ Weaker probabilistic guarantees
  - $\Pr\left[\frac{\mathsf{E}[Z]}{1+\varepsilon} \leq Z \leq \mathsf{E}[Z](1+\varepsilon)\right] \geq 1 - \frac{1}{(\frac{\varepsilon}{1+\varepsilon})^2(\mathsf{E}[Z])}$

Challenge 2: How large is a "small" cell

- Too large $\rightarrowtail$ Hard to enumerate
- Too small $\rightarrowtail$ Weaker probabilistic guarantees
  - $\Pr\left[\frac{\mathsf{E}[Z]}{1+\varepsilon} \leq Z \leq \mathsf{E}[Z](1+\varepsilon)\right] \geq 1 - \frac{1}{(\frac{\varepsilon}{1+\varepsilon})^2(\mathsf{E}[Z])}$

We want a "small" cell to have roughly thresh solutions, where

$$\mathrm{thresh} = 5\left(1 + \frac{1}{\varepsilon^2}\right)$$

Challenges

Challenge 1 How to partition into roughly equal small cells of solutions without knowing the distribution of solutions?

- Independent Support-based 2-Universal Hash Functions

Challenge 2 How large is a "small" cell?

- Independent Support-based 2-Universal Hash Functions

Challenge 3 **How many cells?**

Challenge 3: How many cells?

- A cell is small if it has less than $\mathrm{thresh} = 5(1 + \frac{1}{\varepsilon})^2$ solutions
- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(\varphi)|}{\mathrm{thresh}}$

- A cell is small if it has less than $\mathrm{thresh} = 5(1 + \frac{1}{\varepsilon})^2$ solutions
- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(\varphi)|}{\mathrm{thresh}}$
  - Check for every $m = 0, 1, \cdots n$ if the number of solutions $\leq \mathrm{thresh}$

Challenge 3: How many cells?

- A cell is small if it has less than $\mathrm{thresh} = 5(1 + \frac{1}{\varepsilon})^2$ solutions
- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(\varphi)|}{\mathrm{thresh}}$
  - Check for every $m = 0, 1, \cdots n$ if the number of solutions $\leq \mathrm{thresh}$
  - XORs for each m must be independently chosen

Challenge 3: How many cells?

- A cell is small if it has less than $\mathrm{thresh} = 5(1 + \frac{1}{\varepsilon})^2$ solutions
- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(\varphi)|}{\mathrm{thresh}}$
  - Check for every $m = 0, 1, \cdots n$ if the number of solutions $\leq \mathrm{thresh}$
  - XORs for each m must be independently chosen
    - Query 1: Is $\#(F \wedge Q_1^1) \leq \mathrm{thresh}$
    - Query 2: Is $\#(F \wedge Q_1^2 \wedge Q_2^2) \leq \mathrm{thresh}$
    - $\cdots$
    - Query n: Is $\#(F \wedge Q_1^n \cdots \wedge Q_n^n) \leq \mathrm{thresh}$

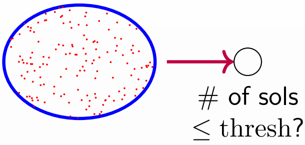
Challenge 3: How many cells?

- A cell is small if it has less than $\mathrm{thresh} = 5(1 + \frac{1}{\varepsilon})^2$ solutions
- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(\varphi)|}{\mathrm{thresh}}$
  - Check for every $m = 0, 1, \cdots n$ if the number of solutions $\leq \mathrm{thresh}$
  - XORs for each m must be independently chosen
    - Query 1: Is $\#(F \wedge Q_1^1) \leq \mathrm{thresh}$
    - Query 2: Is $\#(F \wedge Q_1^2 \wedge Q_2^2) \leq \mathrm{thresh}$
    - $\cdots$
    - Query n: Is $\#(F \wedge Q_1^n \cdots \wedge Q_n^n) \leq \mathrm{thresh}$
  - Stop at the first m where Query m returns YES and return estimate as $\#(F \wedge Q_1^m \cdots \wedge Q_m^m) \times 2^m$

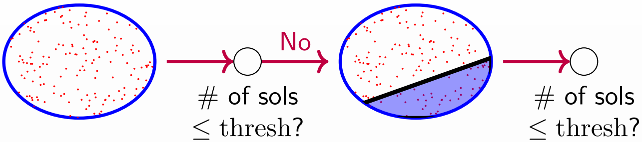
Challenge 3: How many cells?

- A cell is small if it has less than $\mathrm{thresh} = 5(1 + \frac{1}{\varepsilon})^2$ solutions
- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(\varphi)|}{\mathrm{thresh}}$
  - Check for every $m = 0, 1, \cdots n$ if the number of solutions $\leq \mathrm{thresh}$
  - XORs for each m must be independently chosen
    - Query 1: Is $\#(F \wedge Q_1^1) \leq \mathrm{thresh}$
    - Query 2: Is $\#(F \wedge Q_1^2 \wedge Q_2^2) \leq \mathrm{thresh}$
    - $\cdots$
    - Query n: Is $\#(F \wedge Q_1^n \cdots \wedge Q_n^n) \leq \mathrm{thresh}$
  - Stop at the first m where Query m returns YES and return estimate as $\#(F \wedge Q_1^m \cdots \wedge Q_m^m) \times 2^m$
- Number of SAT calls is $\mathcal{O}(n)$ (CMV, CP13) (CFMSV, AAAI14)

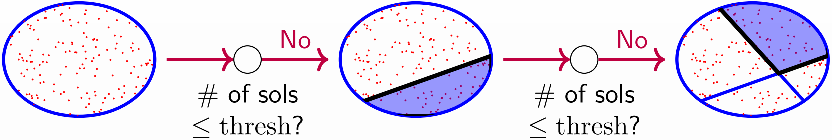
ApproxMC(F, ε, δ)



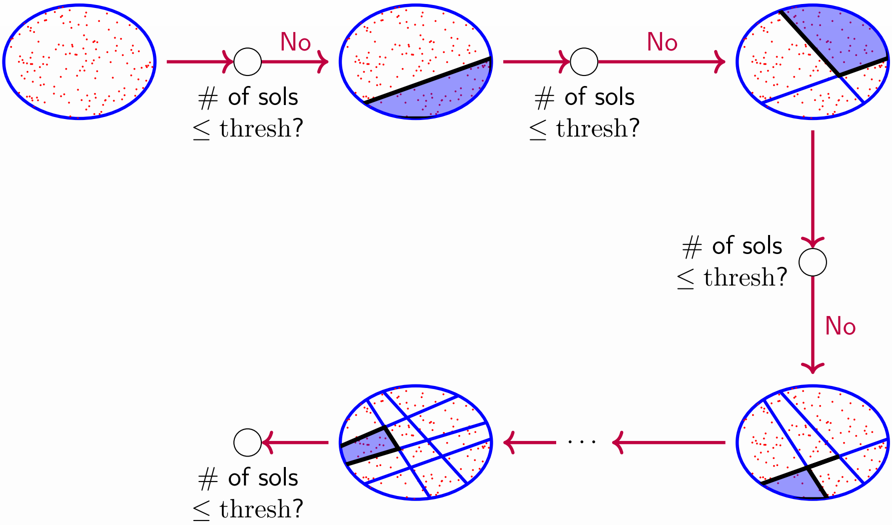
ApproxMC(F, ε, δ)



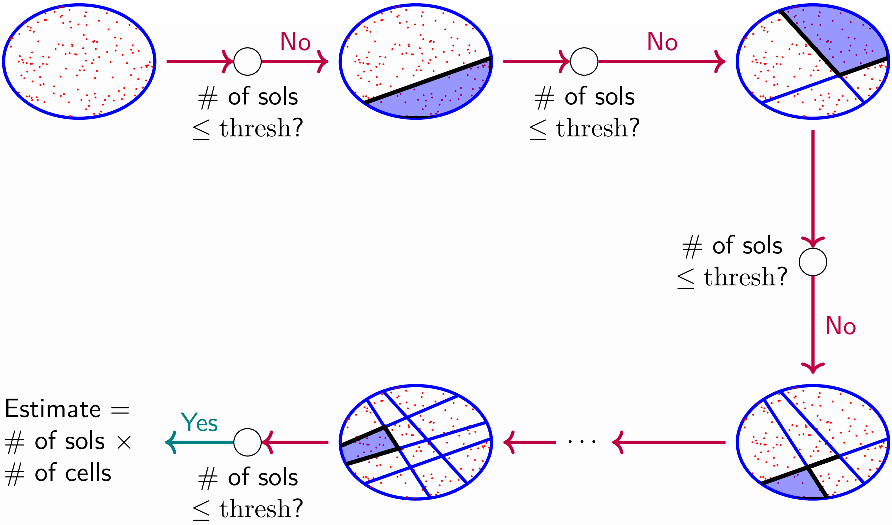
ApproxMC(F, ε, δ)



ApproxMC(F, ε, δ)



ApproxMC(F, ε, δ)



Theoretical Guarantees

Theorem (Correctness)

$$\Pr\left[\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon} \le ApproxMC(F, \varepsilon, \delta) \le |\mathsf{Sol}(\varphi)|(1+\varepsilon)\right] \ge 1 - \delta$$

Theorem (Complexity)

ApproxMC(F, ε, δ) makes $\mathcal{O}(\frac{n \log(\frac{1}{\delta})}{\varepsilon^2})$ calls to SAT oracle.

- *Prior work required $\mathcal{O}(\frac{n \log n \log(\frac{1}{\delta})}{\varepsilon})$ calls to SAT oracle (Stockmeyer 1983)*

Theoretical Guarantees

Theorem (Correctness)

$$\Pr\left[\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon} \leq ApproxMC(F, \varepsilon, \delta) \leq |\mathsf{Sol}(\varphi)|(1+\varepsilon)\right] \geq 1-\delta$$

Theorem (Complexity)

ApproxMC(F, ε, δ) makes $\mathcal{O}(\frac{n \log(\frac{1}{\delta})}{\varepsilon^2})$ calls to SAT oracle.

- *Prior work required $\mathcal{O}(\frac{n \log n \log(\frac{1}{\delta})}{\varepsilon})$ calls to SAT oracle (Stockmeyer 1983)*

Runtime performance

ApproxMC(F, ε, δ)

Theoretical Guarantees

Theorem (Correctness)

$$\Pr\left[\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon} \leq ApproxMC(F, \varepsilon, \delta) \leq |\mathsf{Sol}(\varphi)|(1+\varepsilon)\right] \geq 1-\delta$$

Theorem (Complexity)

$ApproxMC(F, \varepsilon, \delta)$ makes $\mathcal{O}(\frac{n \log(\frac{1}{\delta})}{\varepsilon^2})$ calls to SAT oracle.

- *Prior work required $\mathcal{O}(\frac{n \log n \log(\frac{1}{\delta})}{\varepsilon})$ calls to SAT oracle (Stockmeyer 1983)*

Runtime performance

Handles thousands of variables in few hours but insufficient to solve practical applications

ApproxMC(F, ε, δ)

Theoretical Guarantees

Theorem (Correctness)

$$\Pr\left[\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon} \leq ApproxMC(F, \varepsilon, \delta) \leq |\mathsf{Sol}(\varphi)|(1+\varepsilon)\right] \geq 1-\delta$$

Theorem (Complexity)

ApproxMC(F, ε, δ) makes $\mathcal{O}(\frac{n \log(\frac{1}{\delta})}{\varepsilon^2})$ calls to SAT oracle.

- *Prior work required $\mathcal{O}(\frac{n \log n \log(\frac{1}{\delta})}{\varepsilon})$ calls to SAT oracle (Stockmeyer 1983)*

Runtime performance

Handles thousands of variables in few hours but insufficient to solve practical applications

How to scale to hundreds of thousands of variables and beyond?

Theoretical Guarantees

Theorem (Correctness)

$$\Pr\left[\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon} \leq ApproxMC(F,\varepsilon,\delta) \leq |\mathsf{Sol}(\varphi)|(1+\varepsilon)\right] \geq 1-\delta$$

Theorem (Complexity)

ApproxMC(F, ε, δ) makes $\mathcal{O}(\frac{n\log(\frac{1}{\delta})}{\varepsilon^2})$ calls to SAT oracle.

- *Prior work required $\mathcal{O}(\frac{n\log n\log(\frac{1}{\delta})}{\varepsilon})$ calls to SAT oracle (Stockmeyer 1983)*

Runtime performance

Handles thousands of variables in few hours but insufficient to solve practical applications

How to scale to hundreds of thousands of variables and beyond? Efficient SAT oracle calls?

Beyond ApproxMC

- Query 1: Is $\#(F \wedge Q_1^1) \leq \text{thresh}$
- Query 2: Is $\#(F \wedge Q_1^2 \wedge Q_2^2) \leq \text{thresh}$
- $\cdots$
- Query n: Is $\#(F \wedge Q_1^n \cdots \wedge Q_n^n) \leq \text{thresh}$

Classical View

- Every NP query requires equal amount of time

Beyond ApproxMC

- Query 1: Is $\#(F \wedge Q_1^1) \leq \text{thresh}$
- Query 2: Is $\#(F \wedge Q_1^2 \wedge Q_2^2) \leq \text{thresh}$
- $\cdots$
- Query n: Is $\#(F \wedge Q_1^n \cdots \wedge Q_n^n) \leq \text{thresh}$

Classical View

- Every NP query requires equal amount of time

Practitioner's View

- Solving $(F \wedge Q_1^1)$ followed by $(F \wedge Q_1^2 \wedge Q_2^2)$ requires larger runtime than solving $(F \wedge Q_1^1)$ followed by $(F \wedge Q_1^1 \wedge Q_2^2)$

Beyond ApproxMC

- Query 1: Is $\#(F \wedge Q_1^1) \leq \text{thresh}$
- Query 2: Is $\#(F \wedge Q_1^2 \wedge Q_2^2) \leq \text{thresh}$
- $\cdots$
- Query n: Is $\#(F \wedge Q_1^n \cdots \wedge Q_n^n) \leq \text{thresh}$

Classical View

- Every NP query requires equal amount of time

Practitioner's View

- Solving $(F \wedge Q_1^1)$ followed by $(F \wedge Q_1^2 \wedge Q_2^2)$ requires larger runtime than solving $(F \wedge Q_1^1)$ followed by $(F \wedge Q_1^1 \wedge Q_2^2)$
  - If $(F \wedge Q_1^1) \implies L$ then $(F \wedge Q_1^1 \wedge Q_2^2) \implies L$
  - But, If $(F \wedge Q_1^1) \implies L$ then it is not always the case that $(F \wedge Q_1^2 \wedge Q_2^2) \implies L$

Beyond ApproxMC

- What if we modify our queries to:
  - Query 1: Is $\#(F \wedge Q_1) \leq \text{thresh}$
  - Query 2: Is $\#(F \wedge Q_1 \wedge Q_2) \leq \text{thresh}$
  - $\cdots$
  - Query n: Is $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_n) \leq \text{thresh}$
- Stop at the first m where Query m returns YES and return estimate as $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_m) \times 2^m$
- Observation: $\#(F \wedge Q_1 \cdots \wedge Q_i \wedge Q_{i+1}) \leq \#(F \wedge Q_1 \cdots \wedge Q_i)$
  - If Query i returns YES, then Query $i+1$ must return YES

Beyond ApproxMC

- What if we modify our queries to:
  - Query 1: Is $\#(F \wedge Q_1) \leq \text{thresh}$
  - Query 2: Is $\#(F \wedge Q_1 \wedge Q_2) \leq \text{thresh}$
  - $\cdots$
  - Query n: Is $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_n) \leq \text{thresh}$
- Stop at the first m where Query m returns YES and return estimate as $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_m) \times 2^m$
- Observation: $\#(F \wedge Q_1 \cdots \wedge Q_i \wedge Q_{i+1}) \leq \#(F \wedge Q_1 \cdots \wedge Q_i)$
  - If Query i returns YES, then Query $i+1$ must return YES
  - Galloping search ($\#$ of SAT calls: $\mathcal{O}(\log n)$)
  - Incremental solving

Beyond ApproxMC

- What if we modify our queries to:
  - Query 1: Is $\#(F \wedge Q_1) \leq \text{thresh}$
  - Query 2: Is $\#(F \wedge Q_1 \wedge Q_2) \leq \text{thresh}$
  - $\cdots$
  - Query n: Is $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_n) \leq \text{thresh}$
- Stop at the first m where Query m returns YES and return estimate as $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_m) \times 2^m$
- Observation: $\#(F \wedge Q_1 \cdots \wedge Q_i \wedge Q_{i+1}) \leq \#(F \wedge Q_1 \cdots \wedge Q_i)$
  - If Query i returns YES, then Query $i+1$ must return YES
  - Galloping search (# of SAT calls: $\mathcal{O}(\log n)$)
  - Incremental solving
- But Query i and Query j are no longer independent

Beyond ApproxMC

- What if we modify our queries to:
  - Query 1: Is $\#(F \wedge Q_1) \leq \text{thresh}$
  - Query 2: Is $\#(F \wedge Q_1 \wedge Q_2) \leq \text{thresh}$
  - $\cdots$
  - Query n: Is $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_n) \leq \text{thresh}$
- Stop at the first m where Query m returns YES and return estimate as $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_m) \times 2^m$
- Observation: $\#(F \wedge Q_1 \cdots \wedge Q_i \wedge Q_{i+1}) \leq \#(F \wedge Q_1 \cdots \wedge Q_i)$
  - If Query i returns YES, then Query $i+1$ must return YES
  - Galloping search (# of SAT calls: $\mathcal{O}(\log n)$)
  - Incremental solving
- But Query i and Query j are no longer independent
  - Independence crucial to analysis (Stockmeyer 1983, $\cdots$)

Beyond ApproxMC

- What if we modify our queries to:
  - Query 1: Is $\#(F \wedge Q_1) \leq \text{thresh}$
  - Query 2: Is $\#(F \wedge Q_1 \wedge Q_2) \leq \text{thresh}$
  - $\cdots$
  - Query n: Is $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_n) \leq \text{thresh}$
- Stop at the first m where Query m returns YES and return estimate as $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_m) \times 2^m$
- Observation: $\#(F \wedge Q_1 \cdots \wedge Q_i \wedge Q_{i+1}) \leq \#(F \wedge Q_1 \cdots \wedge Q_i)$
  - If Query i returns YES, then Query $i+1$ must return YES
  - Galloping search (# of SAT calls: $\mathcal{O}(\log n)$)
  - Incremental solving
- But Query i and Query j are no longer independent
  - Independence crucial to analysis (Stockmeyer 1983, $\cdots$)
- Key Insight: The probability of making a bad choice of Q_i is very small for $i \ll m^*$
  - Dependence of Query j upon Query i ($i < j$) does not hurt

(CMV, IJCAI16)

Taming the Curse of Dependence

Let $2^{m^*} = \frac{|\mathsf{Sol}(\varphi)|}{\text{thresh}}$

Lemma (1)

ApproxMC (F, ε, δ) *terminates with* $m \in \{m^* - 1, m^*\}$ *with probability* $\geq$ *0.8*

Lemma (2)

For $m \in \{m^* - 1, m^*\}$*, estimate obtained from a randomly picked cell lies within a tolerance of* ε *of* $|\mathsf{Sol}(\varphi)|$ *with probability* $\geq$ *0.8*

Optimized ApproxMC(F, ε, δ)

Theorem (Correctness)

$$\Pr\left[\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon} \leq \mathit{ApproxMC}(F, \varepsilon, \delta) \leq |\mathsf{Sol}(\varphi)|(1+\varepsilon)\right] \geq 1 - \delta$$

Theorem (Complexity)

ApproxMC(F, ε, δ) *makes* $\mathcal{O}(\frac{\log n \log(\frac{1}{\delta})}{\varepsilon^2})$ *calls to SAT oracle.*

Optimized ApproxMC(F, ε, δ)

Theorem (Correctness)

$$\Pr\left[\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon} \leq ApproxMC(F, \varepsilon, \delta) \leq |\mathsf{Sol}(\varphi)|(1+\varepsilon)\right] \geq 1 - \delta$$

Theorem (Complexity)

ApproxMC(F, ε, δ) makes $\mathcal{O}(\frac{\log n \log(\frac{1}{\delta})}{\varepsilon^2})$ calls to SAT oracle.

Theorem (FPRAS for DNF)

If φ is a DNF formula, then ApproxMC is FPRAS – fundamentally different from the only other known FPRAS for DNF (Karp, Luby 1983)

Beyond Boolean: Handling bit-vectors

- Bit-vector: fixed-width integers
  - Bit-vector constraints can be translated into a Boolean formula
- Significant advancements in bit-vector solving over the past decade
- Challenge: Hash functions for bit vectors
- Lifting hashing from (mod 2) to (mod p) constraints
- p: smallest prime grater than domain of variables

Beyond Boolean: Handling bit-vectors

- Bit-vector: fixed-width integers
  - Bit-vector constraints can be translated into a Boolean formula
- Significant advancements in bit-vector solving over the past decade
- Challenge: Hash functions for bit vectors
- Lifting hashing from (mod 2) to (mod p) constraints
- p: smallest prime grater than domain of variables
- Linear equality (mod p) constraints to hash into cells
- Amenable to Gaussian Elimination

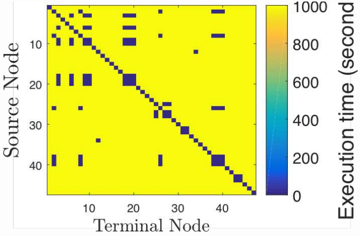
Beyond Boolean: Handling bit-vectors

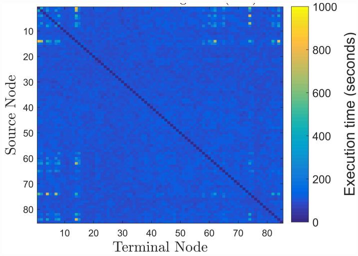
- Bit-vector: fixed-width integers
  - Bit-vector constraints can be translated into a Boolean formula
- Significant advancements in bit-vector solving over the past decade
- Challenge: Hash functions for bit vectors
- Lifting hashing from (mod 2) to (mod p) constraints
- p: smallest prime grater than domain of variables
- Linear equality (mod p) constraints to hash into cells
- Amenable to Gaussian Elimination
- Number of cells: p^m
- Large p does not give finer control on the number of cells
  - Few cells $\rightarrowtail$ too many solutions in a cell
  - Too many cells $\rightarrowtail$ No solutions in most of the cells

H_{SMT}: Efficient word-level Hash Function

- Use different primes to control the number of cells
- Choose appropriate N and express as product of *preferred primes*, i.e., $N = p_1^{c_1} p_2^{c_2} p_3^{c_3} \cdots p_n^{c_n}$
- H_{SMT}:
  - $c_1 \pmod{p}$ constraints
  - $c_2 \pmod{p}$ constraints
  - $\cdots$
- H_{SMT} satisfies guarantees of 2-universality

From Timeouts to under 40 seconds



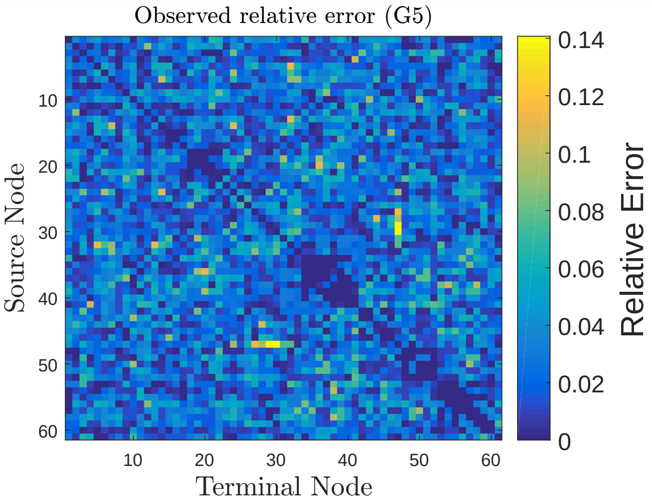


Performance of RDA

Performance of ApproxMC

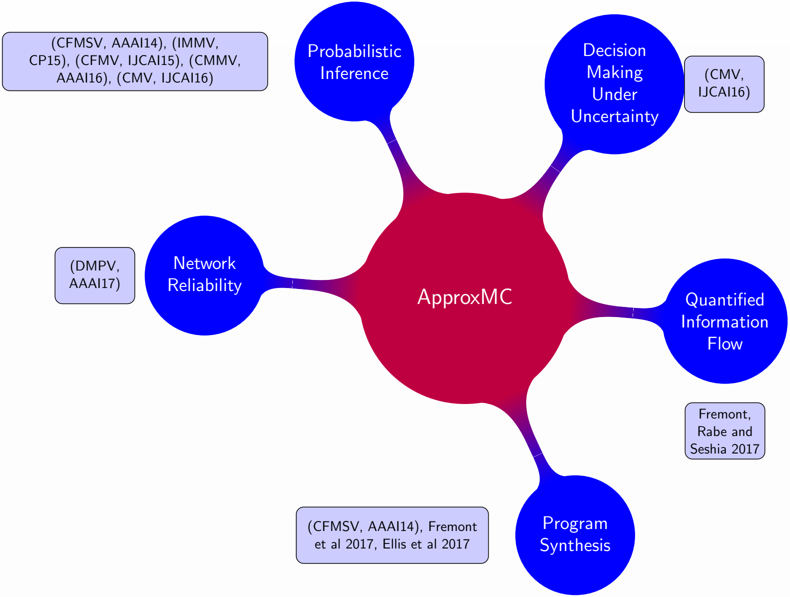
(DMPV, AAAI17)

Highly Accurate Estimates



$(\varepsilon = 0.8, \delta = 0.1)$

Beyond Network Reliability



Part II

Discrete Sampling

Discrete Sampling

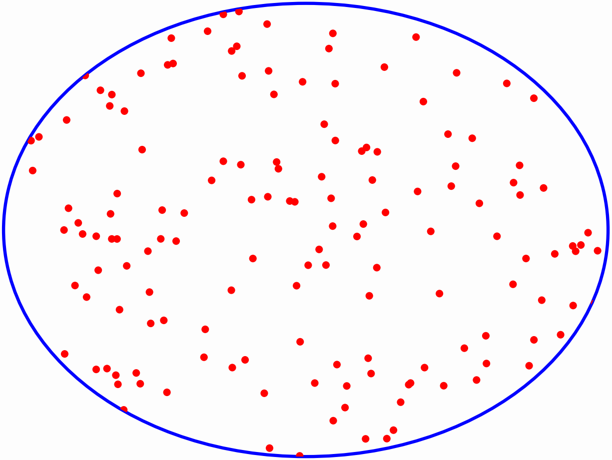
- Given
  - Boolean Variables $X_1, X_2, \cdots X_n$
  - Formula φ over $X_1, X_2, \cdots X_n$
- Uniform Generator

$$\Pr[\text{y is output}] = \frac{1}{|\mathrm{Sol}(\varphi)|}$$

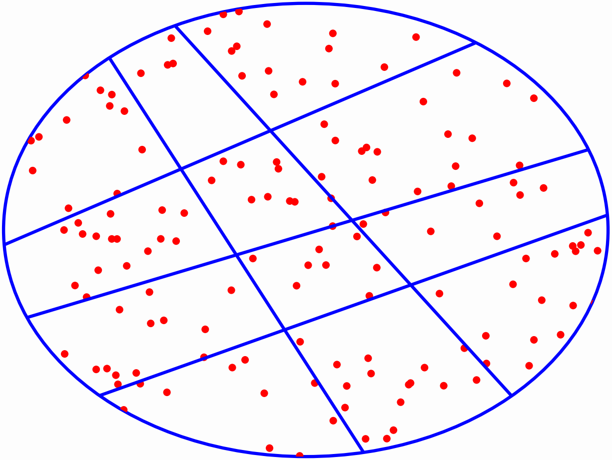
- Almost-Uniform Generator

$$\frac{1}{(1+\varepsilon)|\mathrm{Sol}(\varphi)|} \leq \Pr[\text{y is output}] = \frac{1+\varepsilon}{|\mathrm{Sol}(\varphi)|}$$

As simple as sampling dots

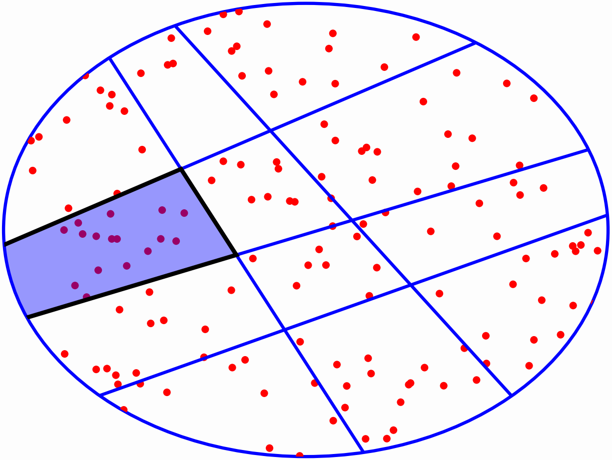


As simple as sampling dots



As simple as sampling dots

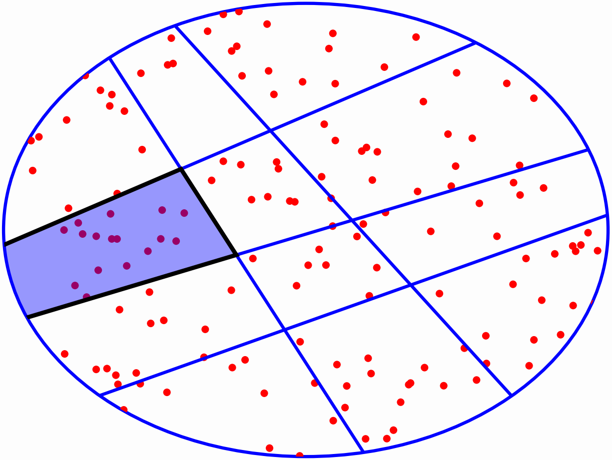
Pick a random cell



Enumerate all the solutions and pick a random solution

As simple as sampling dots

Pick a random cell



Enumerate all the solutions and pick a random solution

Challenge: How many cells?

How many cells?

- Desired Number of cells: $2^{m^*} = \frac{|\mathsf{Sol}(\varphi)|}{\mathrm{thresh}}$
  - But determining $|\mathsf{Sol}(\varphi)|$ is expensive
  - ApproxMC(F, ε, δ) returns C such that
$$\Pr\left[\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon} \leq C \leq |\mathsf{Sol}(\varphi)|(1+\varepsilon)\right] \geq 1-\delta$$
  - $\tilde{m} = \log \frac{C}{\mathrm{thresh}}$ ($m^* = \log \frac{|\mathsf{Sol}(\varphi)|}{\mathrm{thresh}}$)
  - Check for $m = \tilde{m} - 1, \tilde{m}, \tilde{m} + 1$ if a randomly chosen cell is *small*

How many cells?

- Desired Number of cells: $2^{m^*} = \frac{|\mathsf{Sol}(\varphi)|}{\mathrm{thresh}}$
  - But determining $|\mathsf{Sol}(\varphi)|$ is expensive
  - ApproxMC(F, ε, δ) returns C such that
    $$\Pr\left[\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon} \leq C \leq |\mathsf{Sol}(\varphi)|(1+\varepsilon)\right] \geq 1-\delta$$
  - $\tilde{m} = \log \frac{C}{\mathrm{thresh}}$ ($m^* = \log \frac{|\mathsf{Sol}(\varphi)|}{\mathrm{thresh}}$)
  - Check for $m = \tilde{m}-1, \tilde{m}, \tilde{m}+1$ if a randomly chosen cell is *small*
  - Not just a practical hack required non-trivial proof

(CMV, CAV13)
(CMV, DAC14)
(CFMSV, TACAS15)

Theoretical Guarantees

Theorem (Almost-Uniformity)

$$\forall y \in \mathsf{Sol}(\varphi),\ \frac{1}{(1+\varepsilon)|\mathsf{Sol}(\varphi)|} \leq \Pr[y \textit{ is output}] \leq \frac{1+\varepsilon}{|\mathsf{Sol}(\varphi)|}$$

Theoretical Guarantees

Theorem (Almost-Uniformity)

$$\forall y \in \mathsf{Sol}(\varphi),\ \frac{1}{(1+\varepsilon)|\mathsf{Sol}(\varphi)|} \leq \Pr[y \textit{ is output}] \leq \frac{1+\varepsilon}{|\mathsf{Sol}(\varphi)|}$$

Theorem (Query)

For a formula φ over n variables, to generate m samples, UniGen makes **one call** *to approximate counter*

Theoretical Guarantees

Theorem (Almost-Uniformity)

$$\forall y \in \mathsf{Sol}(\varphi),\ \frac{1}{(1+\varepsilon)|\mathsf{Sol}(\varphi)|} \leq \Pr[y \textit{ is output}] \leq \frac{1+\varepsilon}{|\mathsf{Sol}(\varphi)|}$$

Theorem (Query)

For a formula φ over n variables, to generate m samples, UniGen makes **one call** *to approximate counter*

- *JVV (Jerrum, Valiant and Vazirani 1986) makes $n \times m$ calls*

Theoretical Guarantees

Theorem (Almost-Uniformity)

$$\forall y \in \mathsf{Sol}(\varphi),\ \frac{1}{(1+\varepsilon)|\mathsf{Sol}(\varphi)|} \leq \Pr[y \textit{ is output}] \leq \frac{1+\varepsilon}{|\mathsf{Sol}(\varphi)|}$$

Theorem (Query)

For a formula φ over n variables, to generate m samples, UniGen makes **one call** *to approximate counter*

- *JVV (Jerrum, Valiant and Vazirani 1986) makes $n \times m$ calls*

Universality

- JVV employs 2-universal hash functions
- UniGen employs 3-universal hash functions

Theoretical Guarantees

Theorem (Almost-Uniformity)

$$\forall y \in \mathsf{Sol}(\varphi),\ \frac{1}{(1+\varepsilon)|\mathsf{Sol}(\varphi)|} \leq \Pr[y \textit{ is output}] \leq \frac{1+\varepsilon}{|\mathsf{Sol}(\varphi)|}$$

Theorem (Query)

For a formula φ over n variables, to generate m samples, UniGen makes **one call** *to approximate counter*

- *JVV (Jerrum, Valiant and Vazirani 1986) makes $n \times m$ calls*

Universality

- JVV employs 2-universal hash functions
- UniGen employs 3-universal hash functions

Random XORs are 3-universal

Three Orders of Improvement

	Relative Runtime
SAT Solver	1
Desired Uniform Generator	10
UniGen	**20**
XORSample (2012 state of the art)	50000

Experiments over 200+ benchmarks

Three Orders of Improvement

	Relative Runtime
SAT Solver	1
Desired Uniform Generator	10
UniGen	**20**
XORSample (2012 state of the art)	50000

Experiments over 200+ benchmarks

UniGen is highly parallelizable – achieves linear speedup i.e., runtime decreases linearly with number of processors.

Three Orders of Improvement

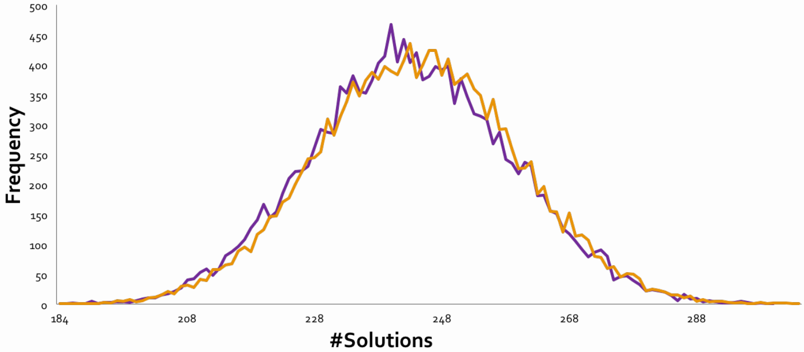
	Relative Runtime
SAT Solver	1
Desired Uniform Generator	10
UniGen (two cores)	**10**
XORSample (2012 state of the art)	50000

Experiments over 200+ benchmarks

UniGen is highly parallelizable – achieves linear speedup i.e., runtime decreases linearly with number of processors.

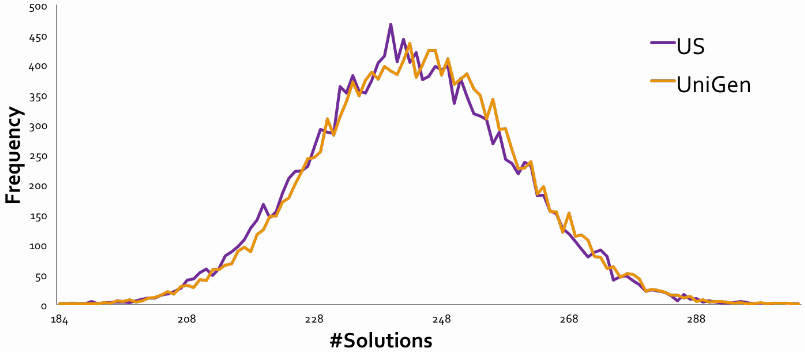
Closer to technical transfer

Uniformity



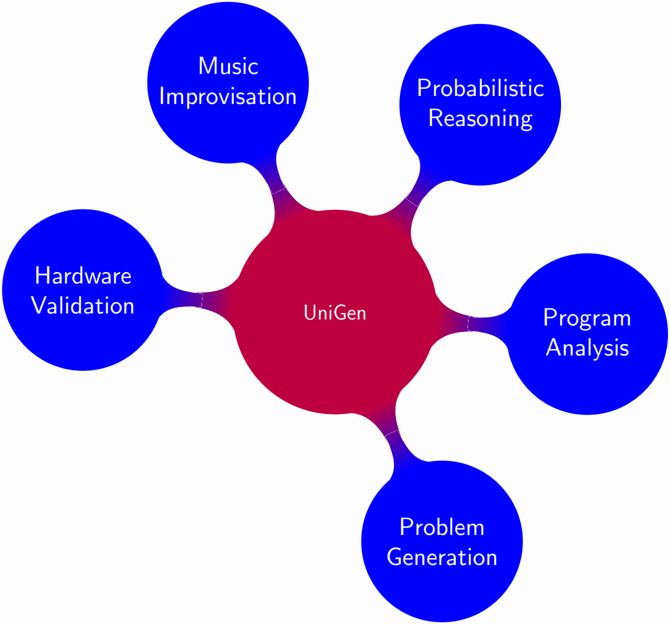
- Benchmark: case110.cnf; #var: 287; #clauses: 1263
- Total Runs: 4×10^6; Total Solutions : 16384

Statistically Indistinguishable



- Benchmark: case110.cnf; #var: 287; #clauses: 1263
- Total Runs: 4×10^6; Total Solutions : 16384

Beyond Verification



Towards Discrete Sampling and Integration Revolution

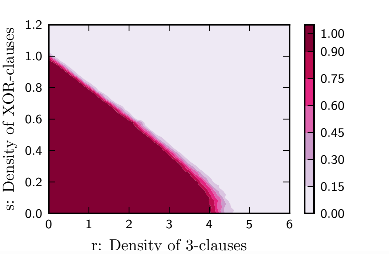
Towards Discrete Sampling and Integration Revolution

- Tighter integration between solvers and algorithms

Towards Discrete Sampling and Integration Revolution

- Tighter integration between solvers and algorithms
- Exploring solution space structure of CNF+XOR formulas

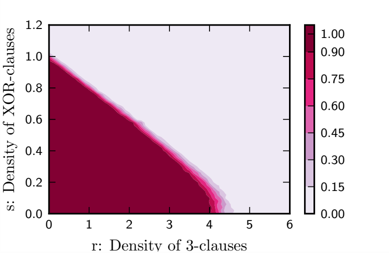
(DMV, IJCAI16)



Towards Discrete Sampling and Integration Revolution

- Tighter integration between solvers and algorithms
- Exploring solution space structure of CNF+XOR formulas

(DMV, IJCAI16)



- Can we handle real variables without discretization?

Summary

- Counting and Sampling are fundamental problems in Computer Science
  - Applications from network reliability, probabilistic inference, side-channel attacks to hardware verification
- Hashing-based approaches provide theoretical guarantees and demonstrate scalability
  - From problems with tens of variables to hundreds of thousands of variables

Generator	Relative Runtime
SAT Solver	1
Desired Uniform Generator	10
UniGen	20
UniGen (two cores)	10
XORSample	50000