Discrete Sampling and Integration in High Dimensional Spaces

Supratik Chakraborty (IIT Bombay)

Kuldeep S. Meel (Rice)

Moshe Y. Vardi (Rice)

Recommended Citation: Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Discrete Sampling and Integration in High Dimensional Spaces, Tutorial at Conference on Uncertainty in Artificial Intelligence, New York, 2016

Problem Definition

- Given
  - X_1 , … X_n : variables with finite discrete domains D_1, … D_n
  - Constraint (logical formula) F over X_1 , … X_n
  - Weight function W: D_1 … D_n $\quad$ 0

Let R_F: set of assignments of X_1 , … X_n that satisfy F

- Determine $W(R_F) = \quad {}_{y \quad R_F} W(y)$
  If W(y) = 1 for all y, then $W(R_F) = | R_F |$

Discrete Integration (Model Counting)

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

Discrete Sampling

Suffices to consider all domains as {0, 1}: assume for this tutorial

Discrete Integration: An Application

- **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]** ?

Discrete Integration: An Application

Probabilistic Inference: Bayes' rule to the rescue

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

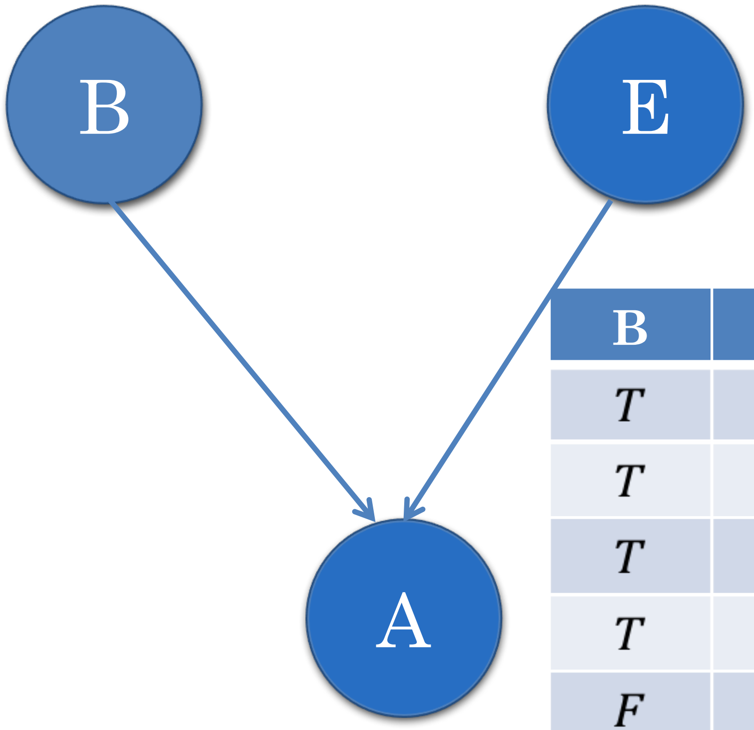
$$\Pr[event_j \cap evidence] = \Pr[evidence \mid event_j] \times \Pr[event_j]$$

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

Discrete Integration: An Application

Probabilistic Graphical Models

B	Pr
T	0.8
F	0.2



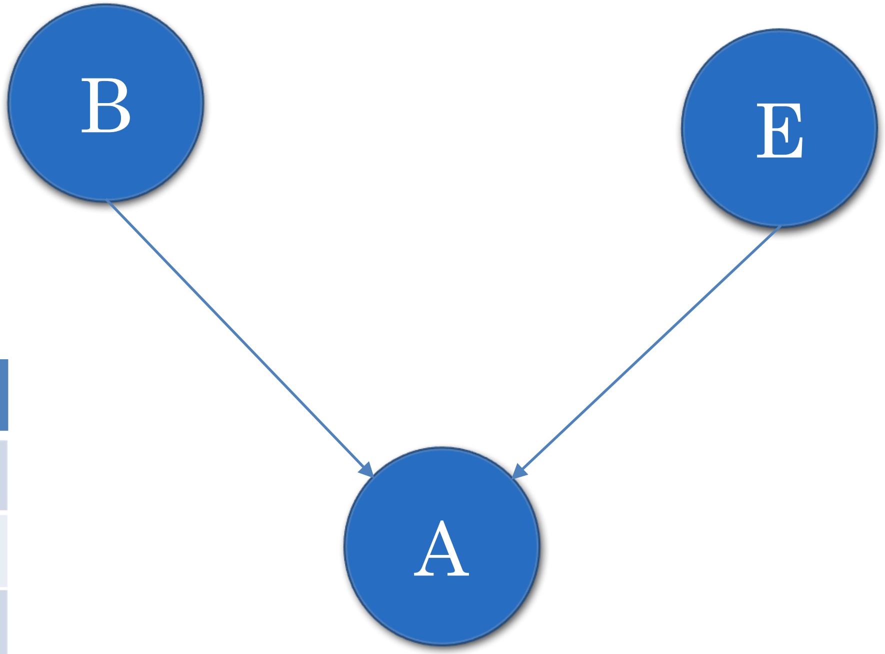
E	Pr
T	0.1
F	0.9

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)

Discrete Integration: An Application

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

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

Discrete Integration: An Application

- **Probabilistic Inference: From probabilities to logic**

$V = \{v_A, v_{\sim A}, v_B, v_{\sim B}, v_E, v_{\sim E}\}$ 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 ($_{PGM}$):

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

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

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

Discrete Integration: An Application

- **Probabilistic Inference: From probabilities to 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

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

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

Product of weights of literals in assignment

Discrete Integration: An Application

- **Probabilistic Inference: From probabilities to logic and weights**

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

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

Formula encoding combination of events in probabilistic model

(Alarm and Earthquake) $F = \ \text{PGM} \quad v_A \quad v_E$

Set of satisfying assignments of F:

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

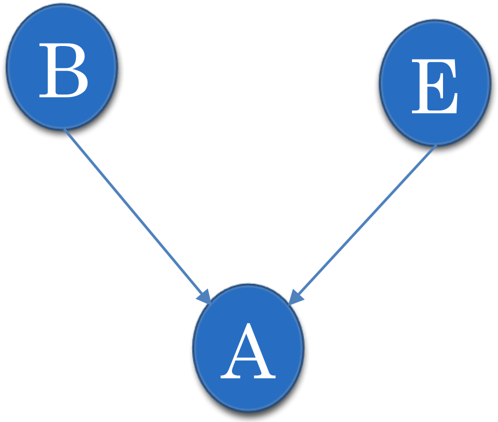
Weight of satisfying assignments of F:

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

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

Discrete Integration: An Application

From probabilistic inference to unweighted model counting



$\Pr[E|A]$

Roth 1996

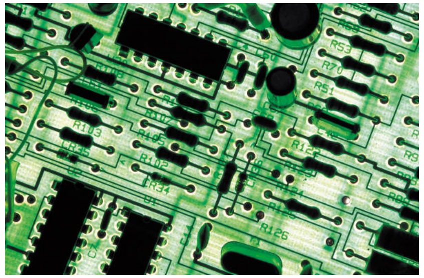
Weighted Model Counting

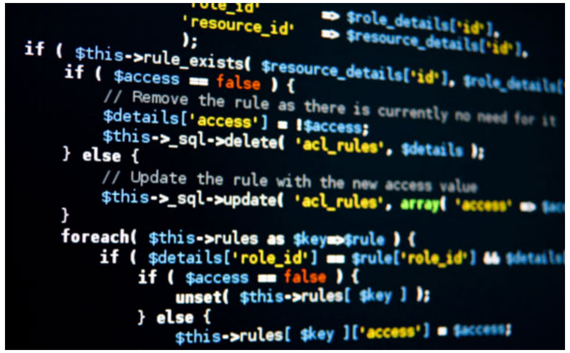
Weighted Model Counting → Unweighted Model Counting

IJCAI 2015

Reduction polynomial in #bits representing CPT entries

Discrete Sampling: An Application





Functional Verification

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

Discrete Sampling: An Application

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

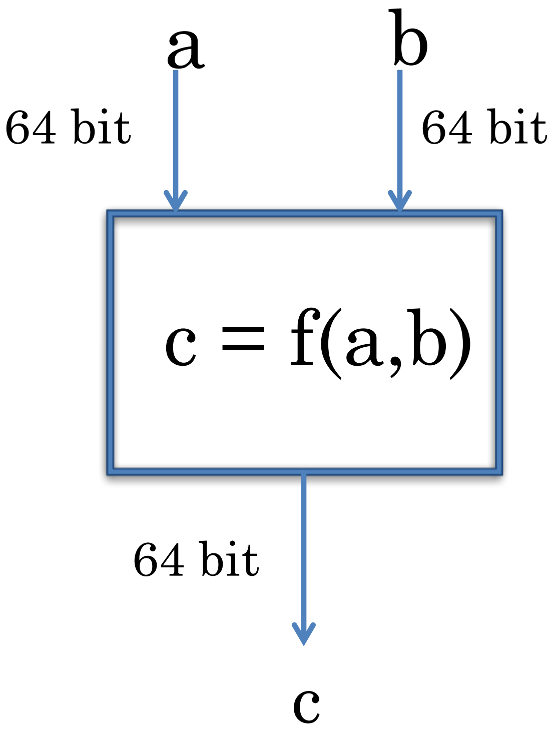
- How do we generate test vectors?

Challenge: Exceedingly large test input space!

Can't try all input combinations

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

Discrete Sampling: An Application

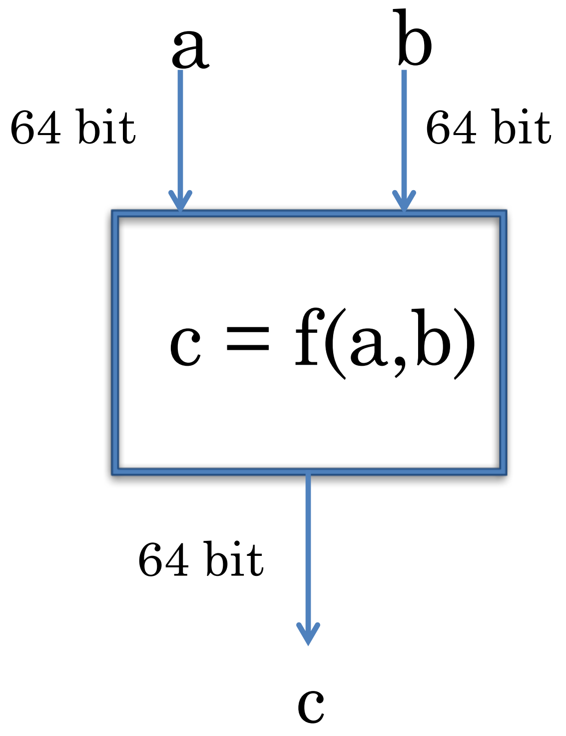


Sources for Constraints

- 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. $232 *_{32} a + b \mathrel{!=} 1100$
  2. $1020 <_{64} (b /_{64} 2) +_{64} a <_{64} 2200$

- Test vectors: solutions of constraints

Discrete Sampling: An Application



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. $1020 <_{64} (b /_{64} 2) +_{64} a <_{64} 2200$

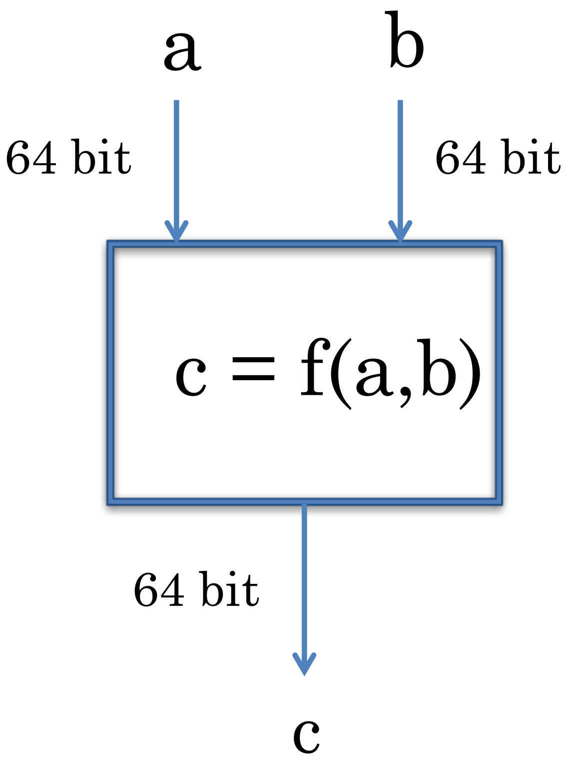
Modern SAT/SMT solvers are complex systems

Efficiency stems from the solver automatically "biasing" search

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

Discrete Sampling: An Application

Constrained Random Verification



Set of Constraints

↓

SAT Formula

↓

Sample satisfying assignments uniformly at random

Scalable Uniform Generation of SAT Witnesses

Discrete Integration and Sampling

- Many, many more applications
  - Physics, economics, network reliability estimation, …

- Discrete integration and discrete sampling are closely related
  - Insights into solving one efficiently and approximately can often be carried over to solving the other
  - More coming in subsequent slides …

Agenda (Part I)

- Hardness of counting/integration and sampling
- Early work on counting and sampling
- Universal hashing
- Universal-hashing based algorithms: an overview

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 ${}_2^{p}$ oracle [Stockmeyer 1983]

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

  Randomized: Polynomial time probabilistic Turing Machine with NP oracle

  [Stockmeyer 1983; Jerrum,Valiant,Vazirani 1986]

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

  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: Polynomial 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: Polynomial time prob. Turing Machine with NP oracle [Jerrum,Valiant,Vazirani 1986, also from Bellare,Goldreich,Petrank 2000]

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

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

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
    $|R_F| = |R_{F1}| * |R_{F2}| * |R_{F3}| * \ldots$
  - 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 easoning 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

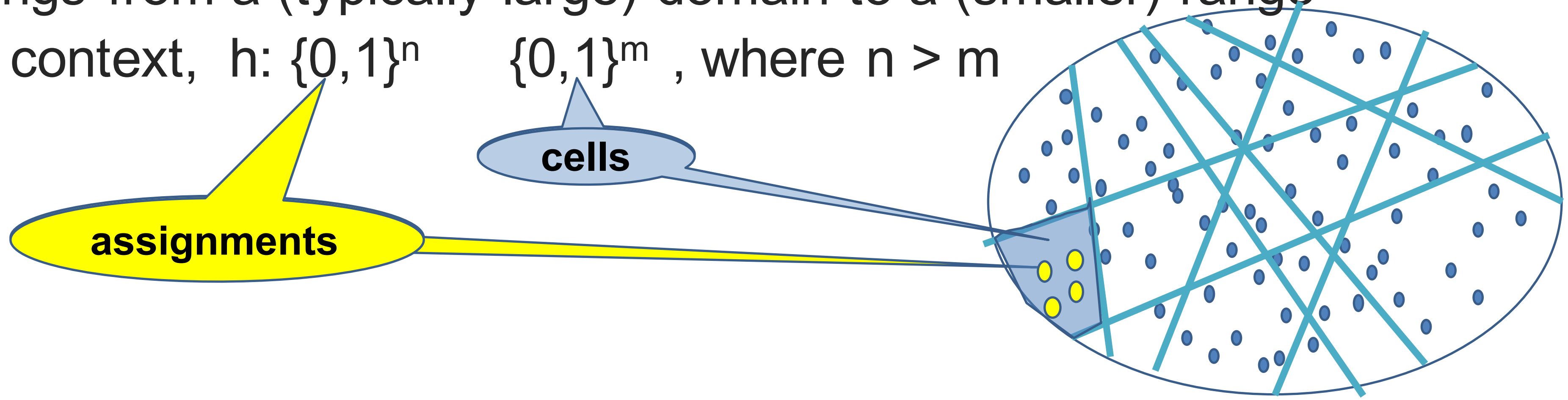
Markov Chain Monte Carlo Techniques

- 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,Kuehlman 2007]

Hashing-based Sampling/Counting

- Extremely successful in recent years [CP2013, CAV2013, NIPS2013, DAC 2014, AAAI 2014, UAI 2014, NIPS 2014, ICML 2014, UAI 2015, ICML 2015, AAAI 2016, ICML 2016, IJCAI 2016, …]
- Focus of remainder of tutorial
- Hash functions:
  - Mappings from a (typically large) domain to a (smaller) range
  - In our context, $h: \{0,1\}^n \quad \{0,1\}^m$, where n > m



More on Hash Functions

- Good deterministic hash function:
  - Inputs distributed uniformly   All cells are small in expectation
  - But solutions of constraints can’t be considered random
- Universal hash functions [Carter,Wegman 1977; Sipser 1983]
  - Define a family of hash functions H having some properties

    Each h   H is a function: $\{0,1\}^n \rightarrow \{0,1\}^m$
  - Choose randomly one hash function h from H
  - For every distribution of inputs, all cells are small and similar in expectation

    Guarantees probabilistic properties of cell sizes even without knowing distribution of inputs
  - Used by Sipser (1983) for combinatorial optimization, by Stockmeyer (1983) for deterministic approximate counting

Universality of Hash Functions and Complexity

- H(n,m,r): Family of r-universal hash functions
  - h : $\{0,1\}^n \rightarrow \{0,1\}^m$
  - For every X $\{0,1\}^n$ and every α $\{0,1\}^m$

    Pr[h(X) = α | h chosen uniformly rand. from H] = **$1/2^m$**

    Uniformity
  - For distinct $X_1, \ldots X_r$ $\{0,1\}^n$ and for every $\alpha_1, \ldots \alpha_r$ $\{0,1\}^m$,

    Pr[$h(X_1) = \alpha_1 \wedge \ldots \wedge h(X_r) = \alpha_r$ | h rand. From H] = **$1/2^{m \cdot r}$**

    Independence-like
- Higher r Stronger guarantees on size of cells

  Lower probability of large variations in cell sizes

- r-wise universality can be implemented using polynomials of degree r-1 in $GF(2^{\max(n,m)})$

  Can be computationally challenging; say n = r = 10000, m < n

- Lower r Lower complexity of reasoning about r-universal hashing

2-Universal Hashing: Simple to Compute

- Variables: $X_1, X_2, X_3, \ldots\ldots, X_n$
- To construct h: $\{0,1\}^n \rightarrow \{0,1\}^m$, choose m random XORs
- Pick every variable with prob. ½ , XOR them and add 1 with prob. ½
- E.g.: $X_1 \oplus X_3 \oplus X_6 \oplus \ldots\ldots \oplus X_{n-1}$
- $\alpha \in \{0,1\}^m \rightarrow$ Set every XOR equation to 0 or 1 randomly
- The cell: F∧XOR (CNF+XOR)

$$\left.\begin{array}{l}
X_1 \oplus X_3 \oplus X_6 \oplus \ldots\ldots X_{n-1} = 0 \\
X_1 \oplus X_2 \oplus X_4 \oplus \ldots\ldots X_{n-1} = 1 \\
X_1 \oplus X_3 \oplus X_5 \oplus \ldots\ldots X_{n-1} = 0 \\
X_2 \oplus X_3 \oplus X_4 \oplus \ldots\ldots X_{n-1} = 0 \\
\ldots\ldots \\
X_1 \oplus X_2 \oplus X_3 \oplus \ldots\ldots X_{n-1} = 0
\end{array}\right\} \text{m XORs}$$

2-Universal Hashing: Yet Powerful

- Let X be the number of solutions of F in an arbitrarily chosen cell
  - What is μ_X, and how much can X deviate from μ_X?
- For every $y \in R_F$, we define $\mathrm{I}_y = \begin{cases} 1, & y \text{ is in cell} \\ 0, & \text{otherwise} \end{cases}$
- X = $\sum_{y \in R_F} I_y$
  - $\mu_X = \frac{|R_F|}{2^m}$......... From random choice of hash function
  - $\sigma_X^2 \leq \mu_X$......... From 2-universality of hash function
- This gives the concentration bound:

$$\Pr\left[\frac{\mu_X}{1+\epsilon} \leq X \leq \mu_X(1+\epsilon)\right] \geq 1 - \frac{\sigma^2}{(\frac{\varepsilon}{1+\epsilon})^2(\mu_X)^2} \geq 1 - \frac{1}{(\frac{\varepsilon}{1+\epsilon})^2 \mu_X}$$

Having μ_X>k(1+$\frac{1}{\epsilon^2}$) gives us $1 - \frac{1}{k}$ lower bound

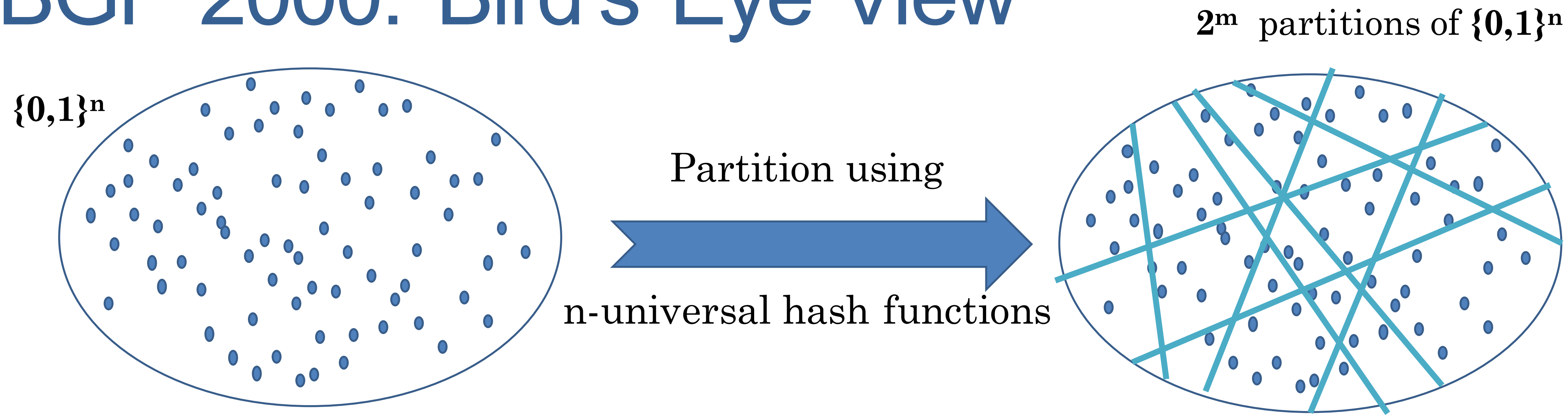
Hashing-based Sampling

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

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

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

BGP 2000: Bird's Eye View



- For right choice of m, all the cells are small (# of solutions $\leq 2n^2$)
- Check if all the cells are small (NP- Query)
- If yes, pick a solution randomly from randomly picked cell

In practice, the query is too long and complex for large n, and can not be handled by modern SAT Solvers!

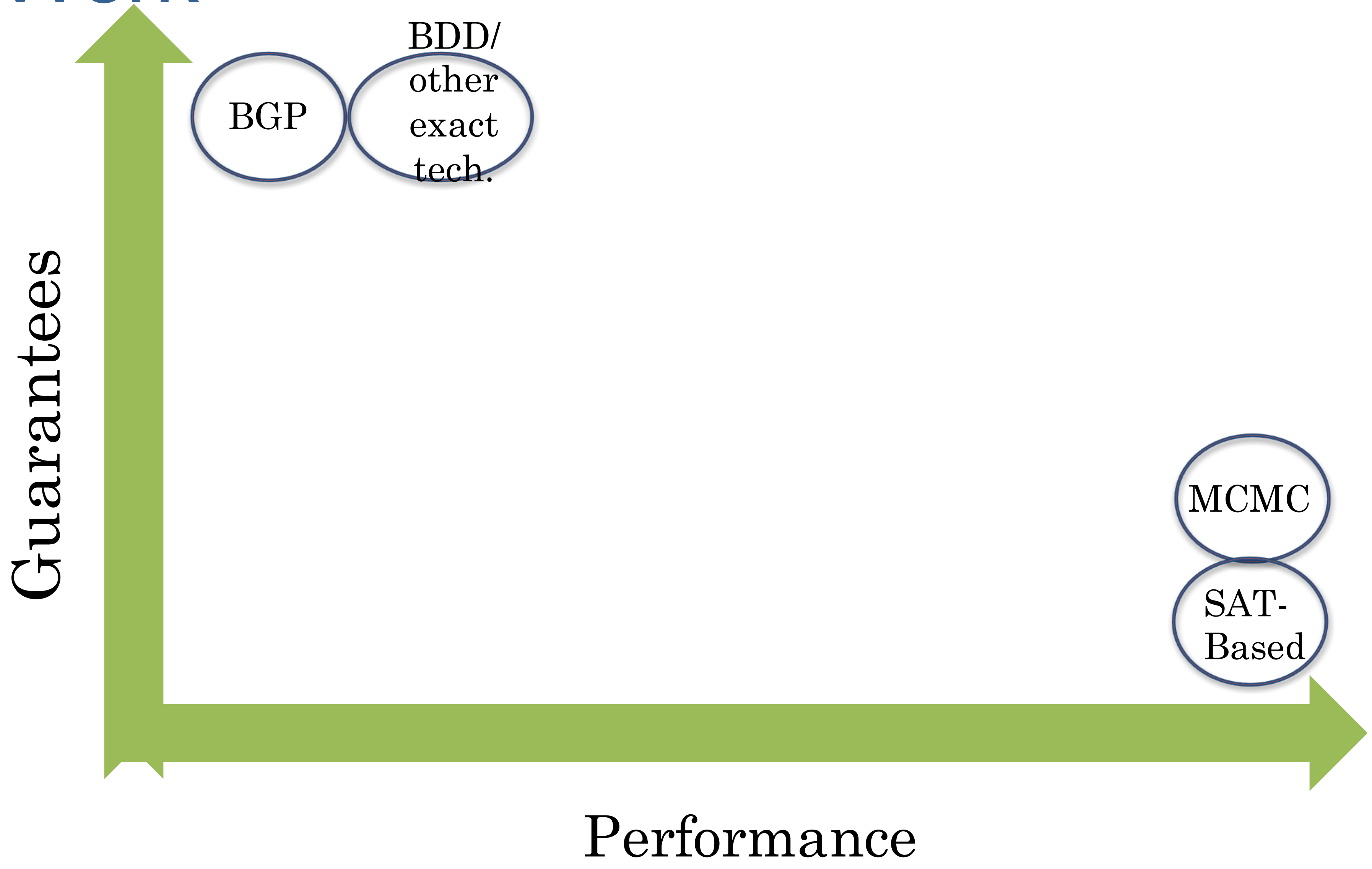
Approximate Integration and Sampling: Close Cousins

- 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: $\{0,1\}^n$ $_0$
  - 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 $W(R_F) = W(a_2)*1 + W(a_4)*2 + W(a_8)*4 + \dots$
- 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 Sampling

- Remainder of tutorial
  - Deeper dive into XOR hash-based counting and sampling
  - Discuss theoretical aspects and experimental observations
  - Leverage power of modern SAT solvers for CNF + XOR clauses (CryptoMiniSAT)

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

  - Tutorial to focus mostly on unweighted case, to elucidate key ideas

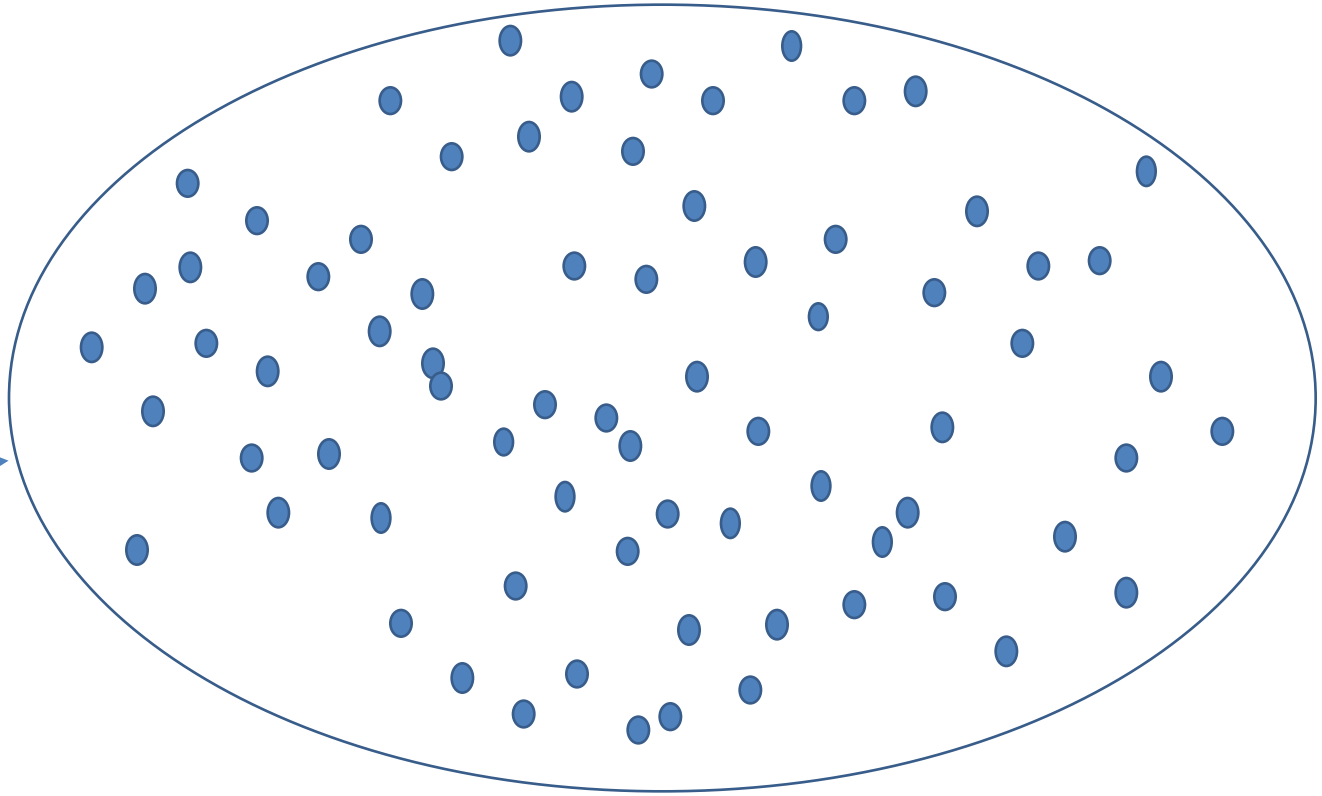
Agenda (Part II)

1. Hashing-based Approaches to Unweighted Model COunting
2. Hashing-based Approaches to Sampling
3. Design of Efficient Hash Functions
4. Summary

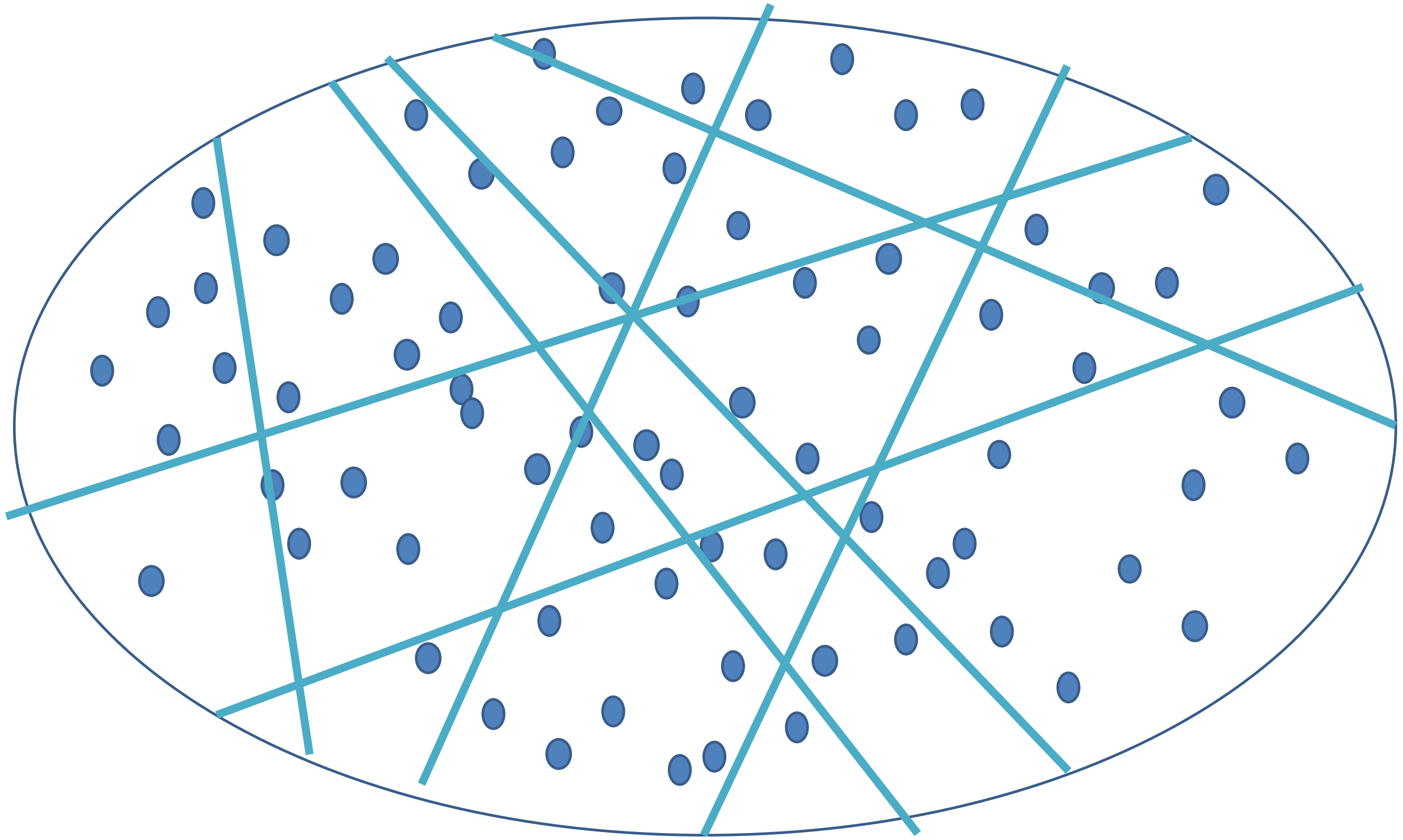
Counting Dots

- Solution to constraints

$\{0,1\}^n$

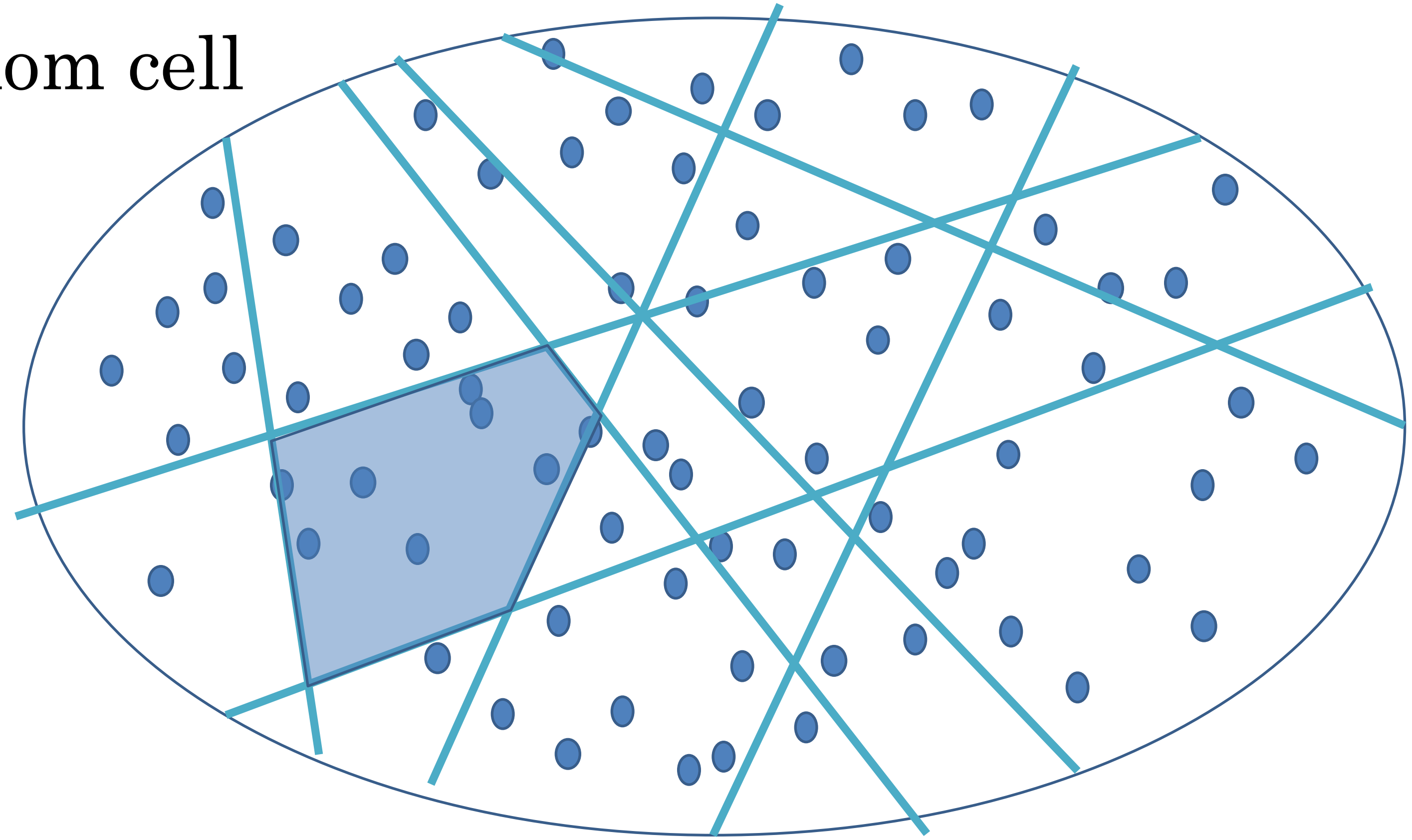


Partitioning into equal “small” cells



Partitioning into equal "small" cells

Pick a random cell



Estimate = # of solutions (dots) in cell * # of cells

How to Partition?

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

2-Universal Hashing [Carter-Wegman 1977]

Partitioning

1. How large is the “small” cell?

2. How do we compute solutions inside a cell?

3. How many cells?

Question 1: Size of cell

- Too large Hard to enumerate
- Too small Ratio of variance to mean is very high

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

Question 2: Solving a cell

- Variables: $X_1, X_2, X_3, \ldots\ldots, X_n$
- To construct h: $\{0,1\}^n \rightarrow \{0,1\}^m$, choose m random XORs
- Pick every variable with prob. ½ , XOR them and add 1 with prob. ½
- E.g.: $X_1 \oplus X_3 \oplus X_6 \oplus \ldots. \oplus X_{n-1}$
- $\alpha \in \{0,1\}^m \rightarrow$ Set every XOR equation to 0 or 1 randomly
- The cell: F $\wedge$ XOR (CNF+XOR)

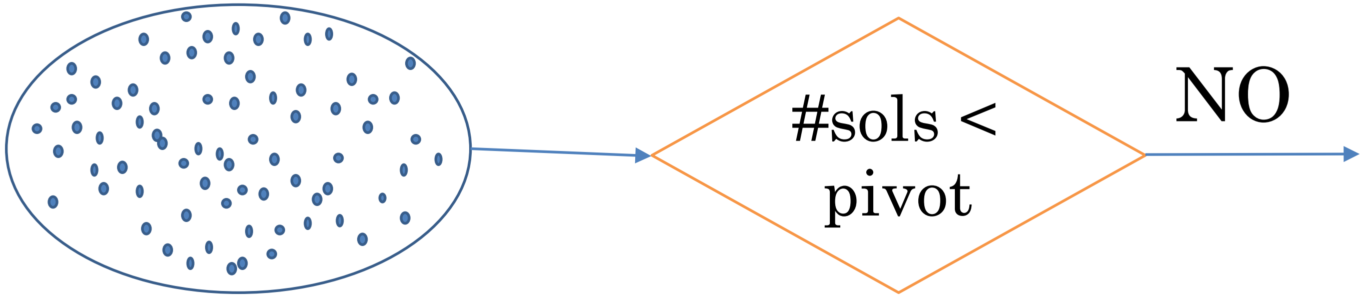
$$\left.\begin{array}{l} X_1 \oplus X_3 \oplus X_6 \oplus \ldots.\ X_{n-1} = 0 \\ X_1 \oplus X_2 \oplus X_4 \oplus \ldots.X_{n-1} = 1 \\ X_1 \oplus X_3 \oplus X_5 \oplus \ldots.\ X_{n-1} = 0 \\ X_2 \oplus X_3 \oplus X_4 \oplus \ldots.\ X_{n-1} = 0 \\ \ldots\ldots \\ X_1 \oplus X_2 \oplus X_3 \oplus \ldots.\ X_{n-1} = 0 \end{array}\right\} \text{m XORs}$$

Question 3: How many cells?

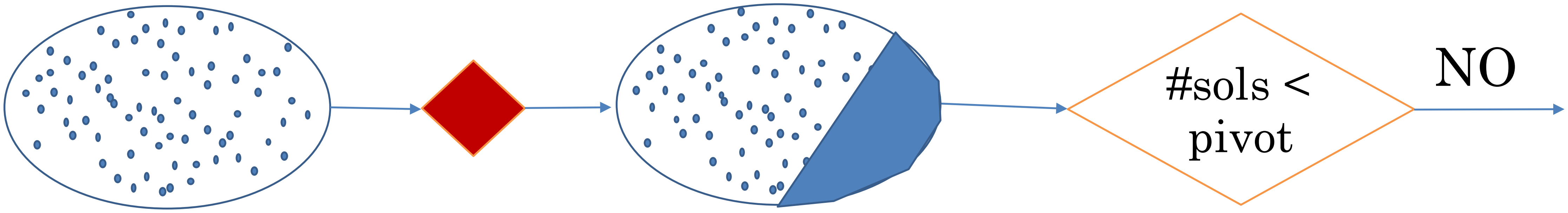
- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|R_F|}{pivot}$
  - Check for every m = 0,1….n if the number of solutions < pivot (function of ε)
  - Stop at the first m where number of solutions < pivot
  - Hash functions must be independent across different checks
- # of SAT calls is O(n)

(CP 2013)

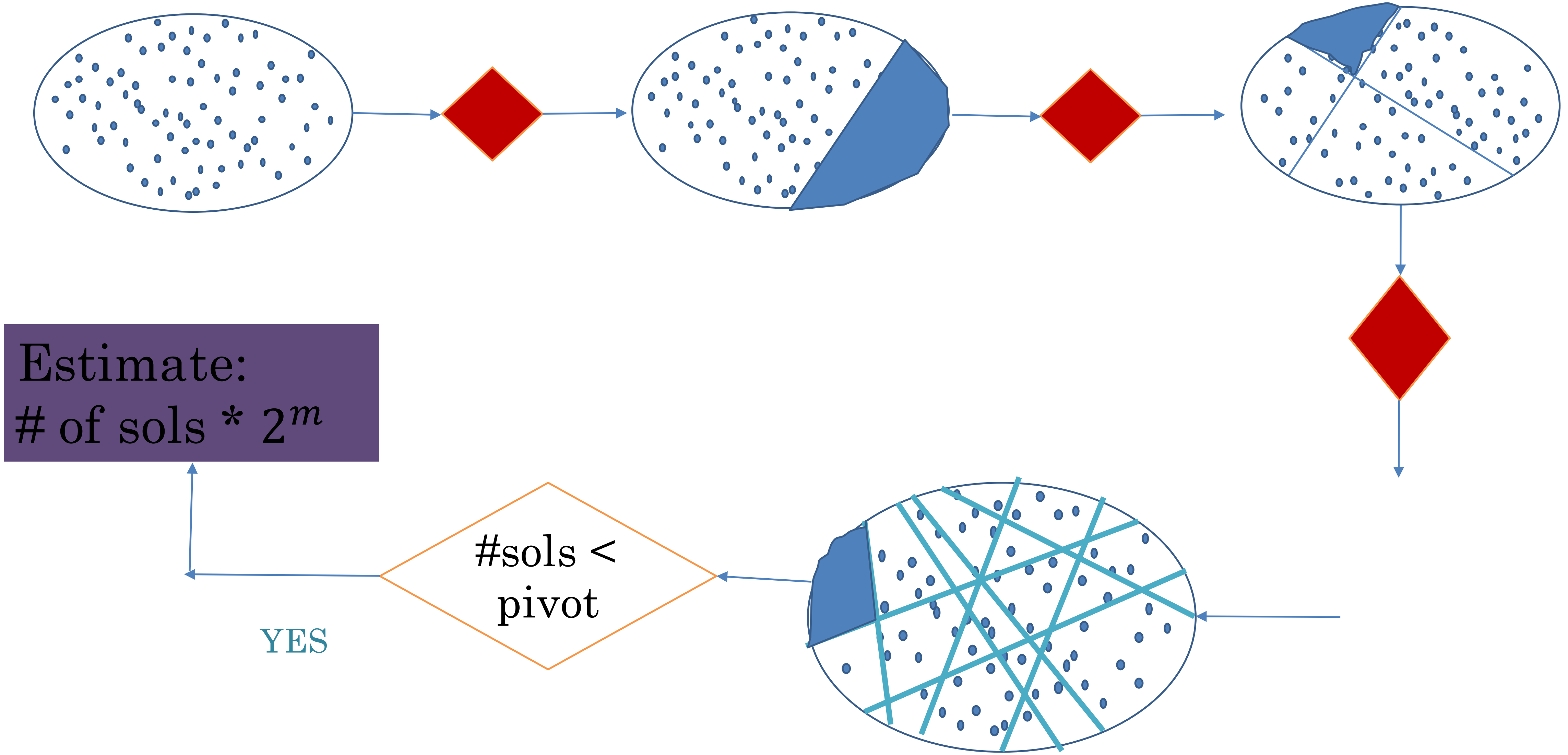
ApproxMC(F,ε, δ)



ApproxMC(F,ε, δ)



ApproxMC(F,ε, δ)



ApproxMC(F,ε, δ)

Key Lemmas

Let $m^* = \log\frac{|R_F|}{pivot}$ (i.e., $2^{m^*} = \frac{|R_F|}{pivot}$)

Lemma 1: The algorithm terminates with $m \in [m^* - 1, m^*]$ with high probability

Lemma 2: The estimate from a randomly picked cell for $m \in [m^* - 1, m^*]$ is correct with high probability

ApproxMC(F,ε, δ)

Theorem 1:

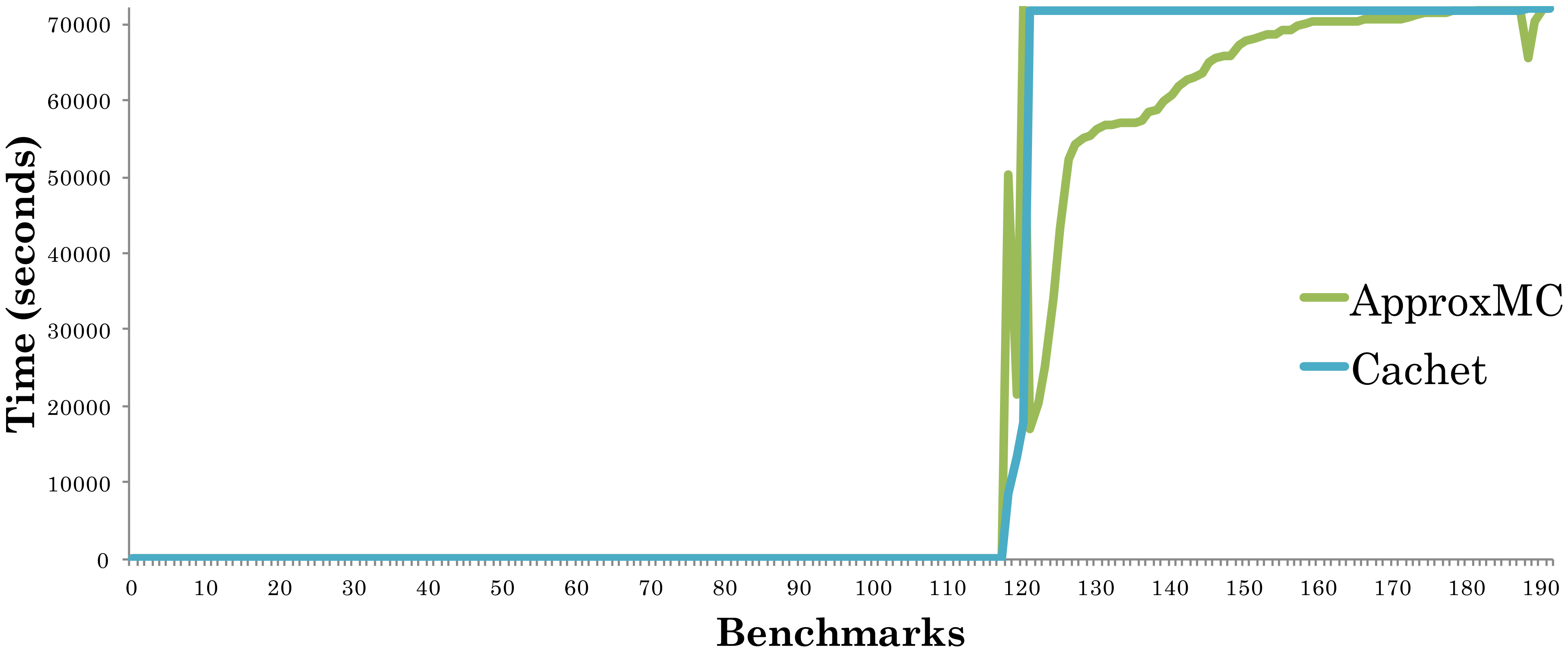
$$\Pr\left[\frac{|R_F|}{(1+\varepsilon)} \leq \text{ApproxMC}(\text{F},\varepsilon,\delta) \leq |R_F|(1+\varepsilon)\right] \geq 1-\delta$$

Theorem 2:

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

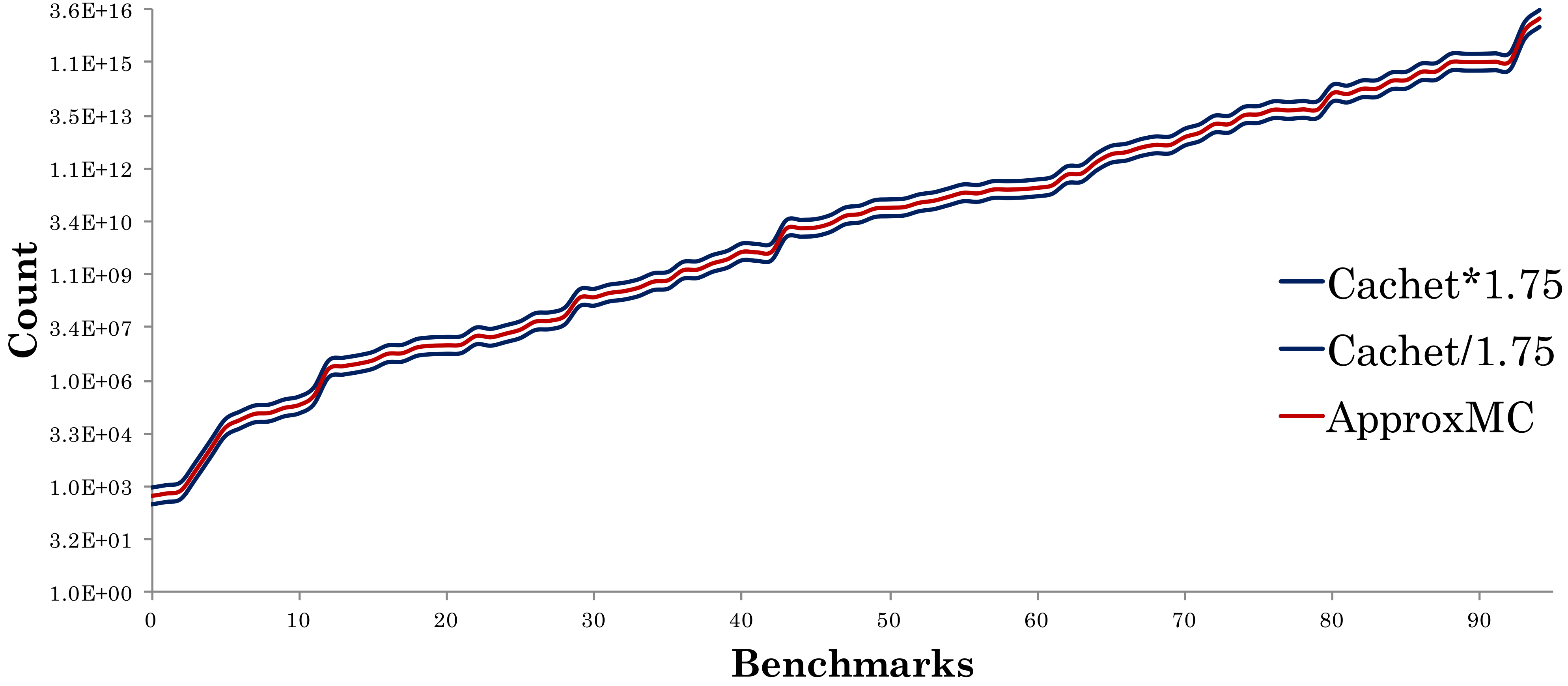
Runtime Performance of ApproxMC

Can Solve a Large Class of Problems



Large class of problems that lie beyond the exact algorithms but can be computed by ApproxMC

Mean Error: Only 4% (allowed: 75%)



Mean error: 4% – much smaller than the theoretical guarantee of 75%

Challenge

- Can we reduce the number of SAT calls from O(n)?

Experimental Observations

- ApproxMC “seems to work” even if we do not have independence across different hash functions
  - Can we really give up independence?

Beyond ApproxMC

- We want to partition into 2^m cells
  - Check for every m = 0,1….n if the number of solutions < pivot
  - Stop at the first m where number of solutions < pivot
  - Hash functions must be independent across different checks (Stockmeyer 1983, Jerrum, Valiant and Vazirani 1986…..)
- **Suppose:** Hash functions *can be dependent* across different checks
- # of solutions is monotonically non-increasing with m
  - Can find the right value of m by search in any order.
  - Binary search

ApproxMC2: From Linear to Logarithmic SAT calls

- The Proof: Hash functions *can be dependent* across different checks
- Key Idea: Probability of making a bad choice early on is very small.
  - Inversely (exponentially!) proportional to distance from m*)

(IJCAI 2016)

ApproxMC2(F,ε, δ)

Theorem 1:

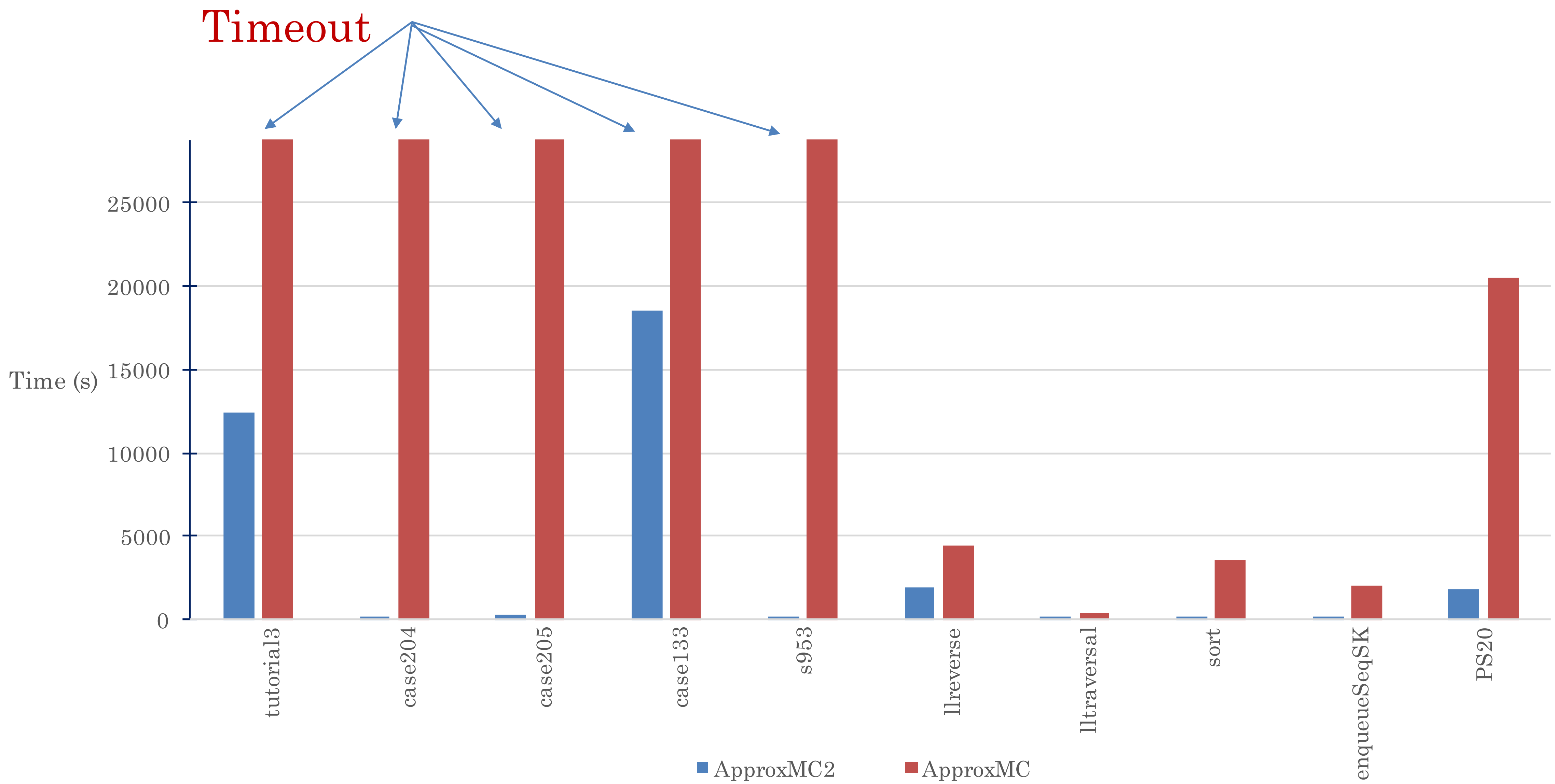
$$\Pr\left[\frac{|R_F|}{(1+\varepsilon)} \leq \text{ApproxMC2}(\text{F},\varepsilon,\delta) \leq |R_F|(1+\varepsilon)\right] \geq 1-\delta$$

Theorem 2:

ApproxMC2(F,ε, δ) makes $\mathrm{O}\left(\frac{(\log n)\log\frac{1}{\delta}}{\varepsilon^2}\right)$ calls to NP oracle

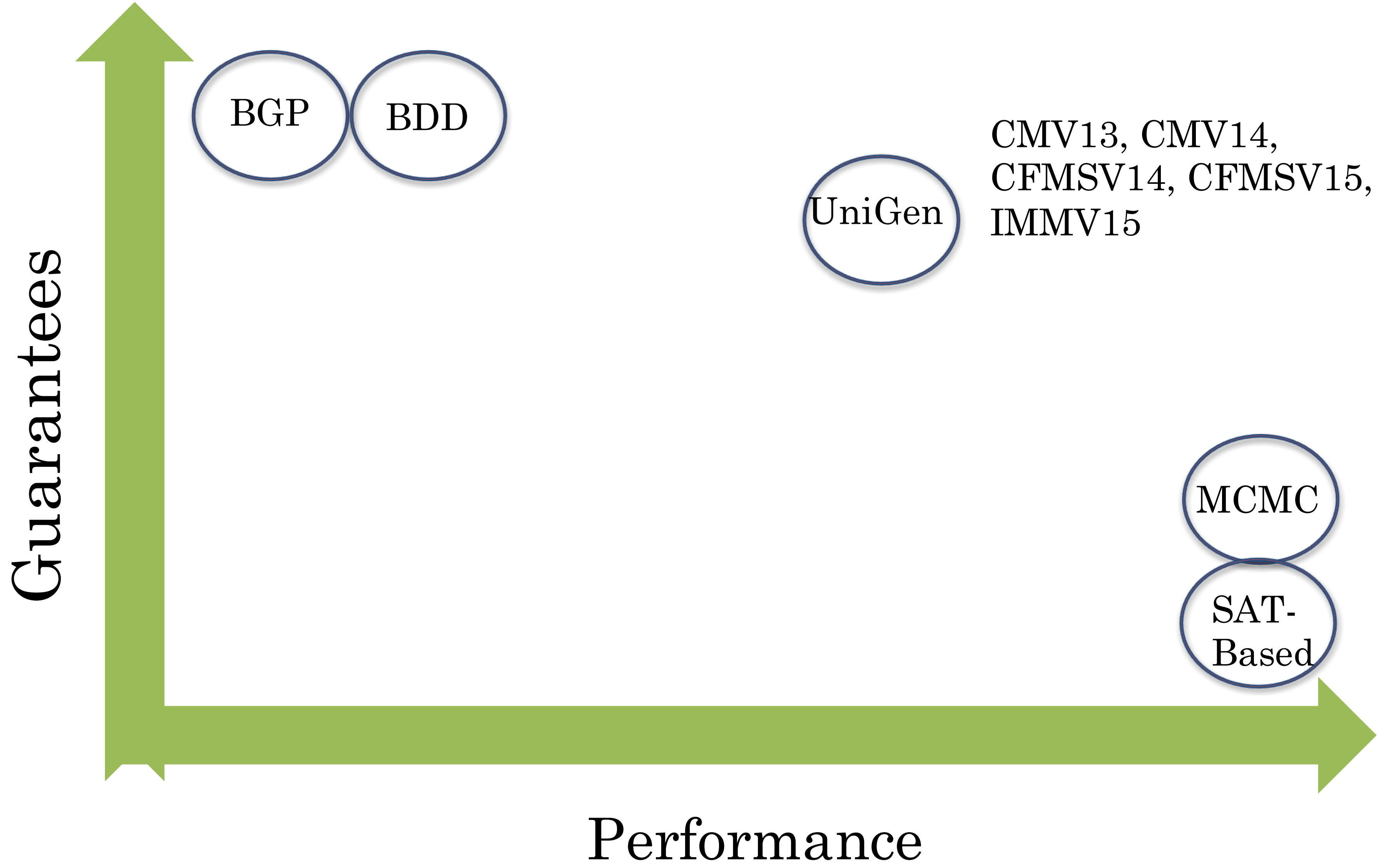
Theorem 1 requires a completely new proof.

Runtime Performance Comparison

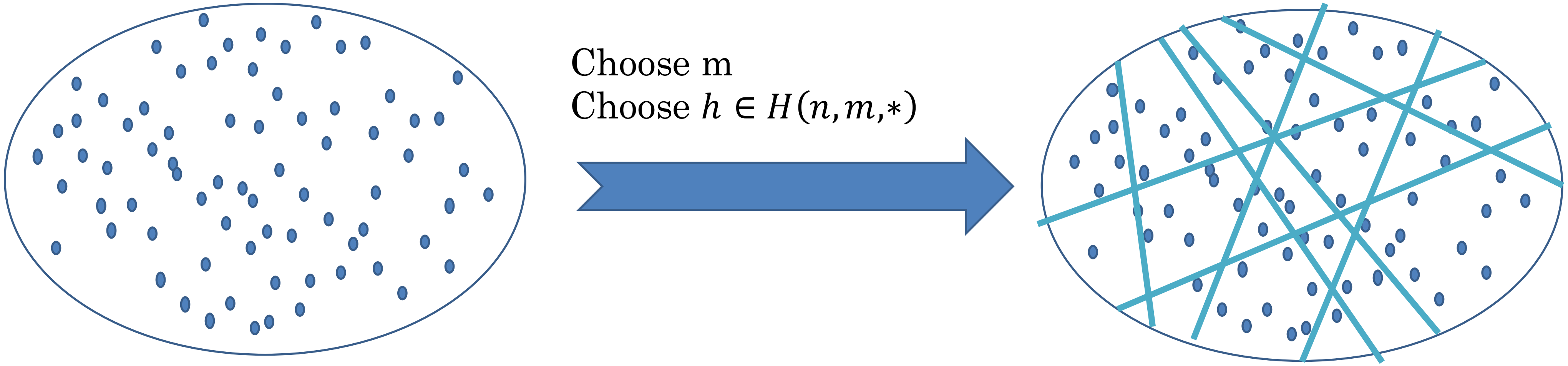


Discrete Uniform Sampling

Hashing-based Approaches



Key Ideas



- For right choice of m, large number of cells are "small"
  - "almost all" the cells are "roughly" equal
- Check if a randomly picked cell is "small"
- If yes, pick a solution randomly from randomly picked cell

Key Challenges

- F: Formula X: Set of variables R_F: Solution space
- $R_{F,h,\alpha}$: Set of solutions for $F \wedge (h(X) = \alpha)$ where
  - $h \in H(n, m,*)$; $\alpha \in \{0,1\}^m$

1. How large is “small” cell ?
2. How much universality do we need?
3. What is the value of m?

Size of cell

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

Independence

Theorem (CMV 14):

3-universal hashing is sufficient to provide almost uniformity.
(3-universality of XOR-based hash functions due to Gomes et al.)

CAV 2013, DAC 2014

How many cells?

- Our desire: $m = \log\frac{|R_F|}{pivot}$ (Number of cells: 2^m)
  - But determining $|R_F|$ is expensive (#P complete)

- How about approximation?
  - $ApproxMC(F, \varepsilon, \delta)$ returns C:

    $$\Pr\left[\frac{|R_F|}{1+\varepsilon} \leq C \leq (1+\varepsilon)|R_F| \right] \geq 1 - \delta$$
  - $q = \log\frac{C}{pivot}$

  - Concentrate on m = q-1, q, q+1

UniGen(F,ε)

One time execution

1. C = ApproxMC(F,ε)
2. Compute pivot
3. $q = \log|C| - \log pivot$
4. for i in {q-1, q, q+1}:
5. Choose h **randomly*** from H(n,i,3)
6. Choose α **randomly*** from $\{0,1\}^m$
7. If $(1 \leq |R_{F,h,\alpha}| \leq pivot)$:
8. Pick $y \in R_{F,h,\alpha}$ randomly

Run for every sample required

Are we back to JVV (Jerrum, Valiant and Vazirani)?

NOT Really

- JVV makes linear (in n) calls to Approximate counter compared to just 1 in UniGen

- # of calls to ApproxMC is only 1 regardless of the number of samples required unlike JVV

Theoretical Guarantees

- Almost-Uniformity

  For every solution $y \in R_F$

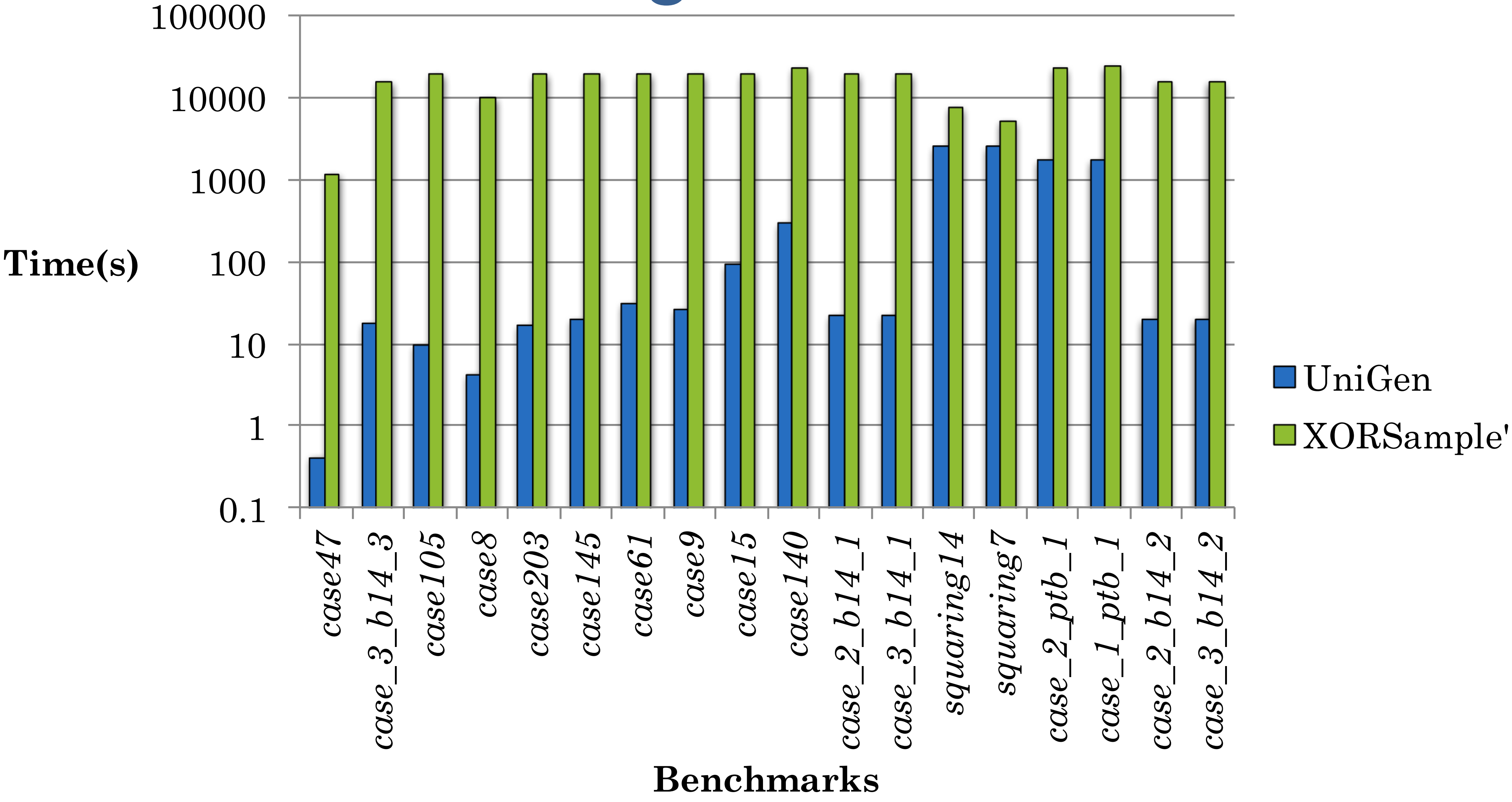
  $$\forall y \in R_F, \qquad \frac{1}{(1+\varepsilon)|R_F|} \leq \Pr[y \text{ is output }] \leq \frac{(1+\varepsilon)}{|R_F|}$$

- UniGen succeeds with probability ≥ 0.52
  - In practice, success probabiliy ≥ 0.99

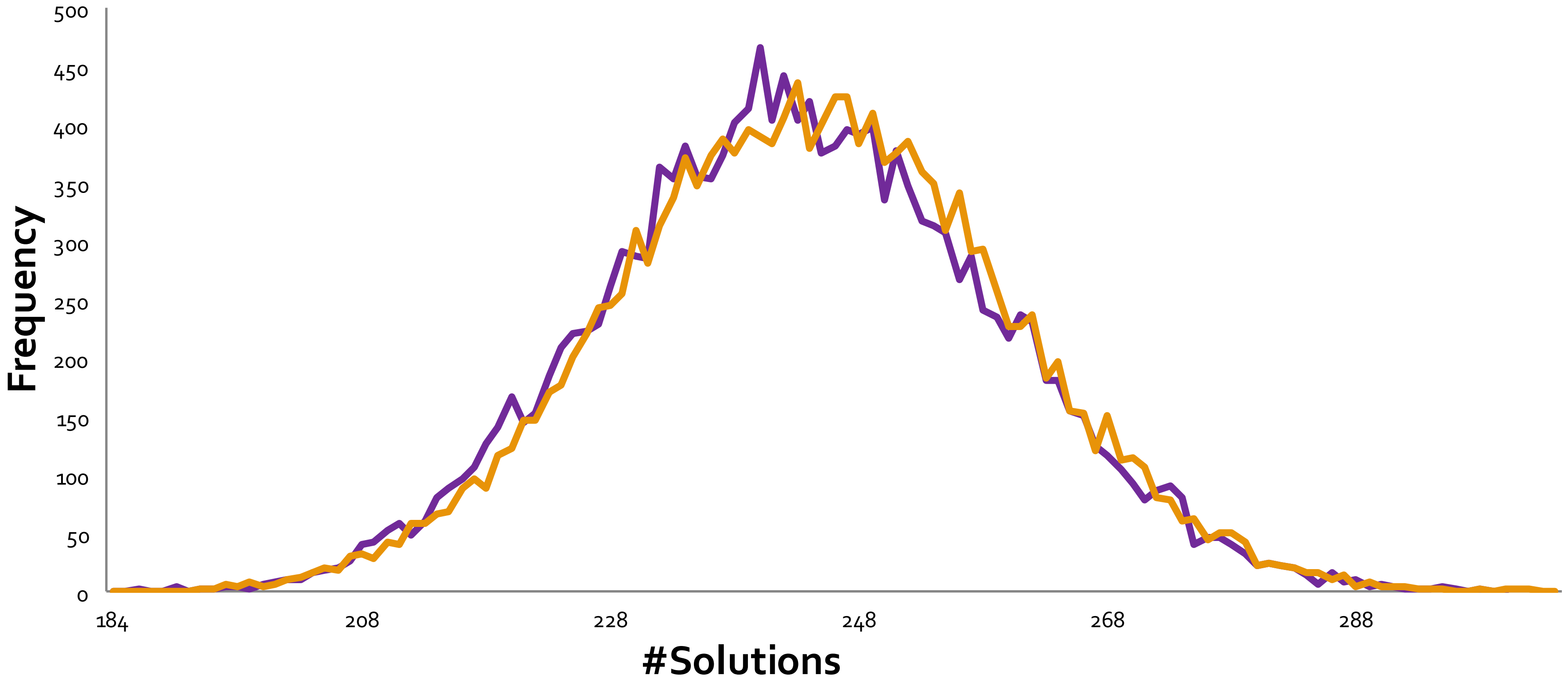
- UniGen makes O($\frac{n}{\varepsilon^2}$) calls to NP oracle (SAT solver)

Runtime Performance of UniGen

1-2 Orders of Magnitude Faster

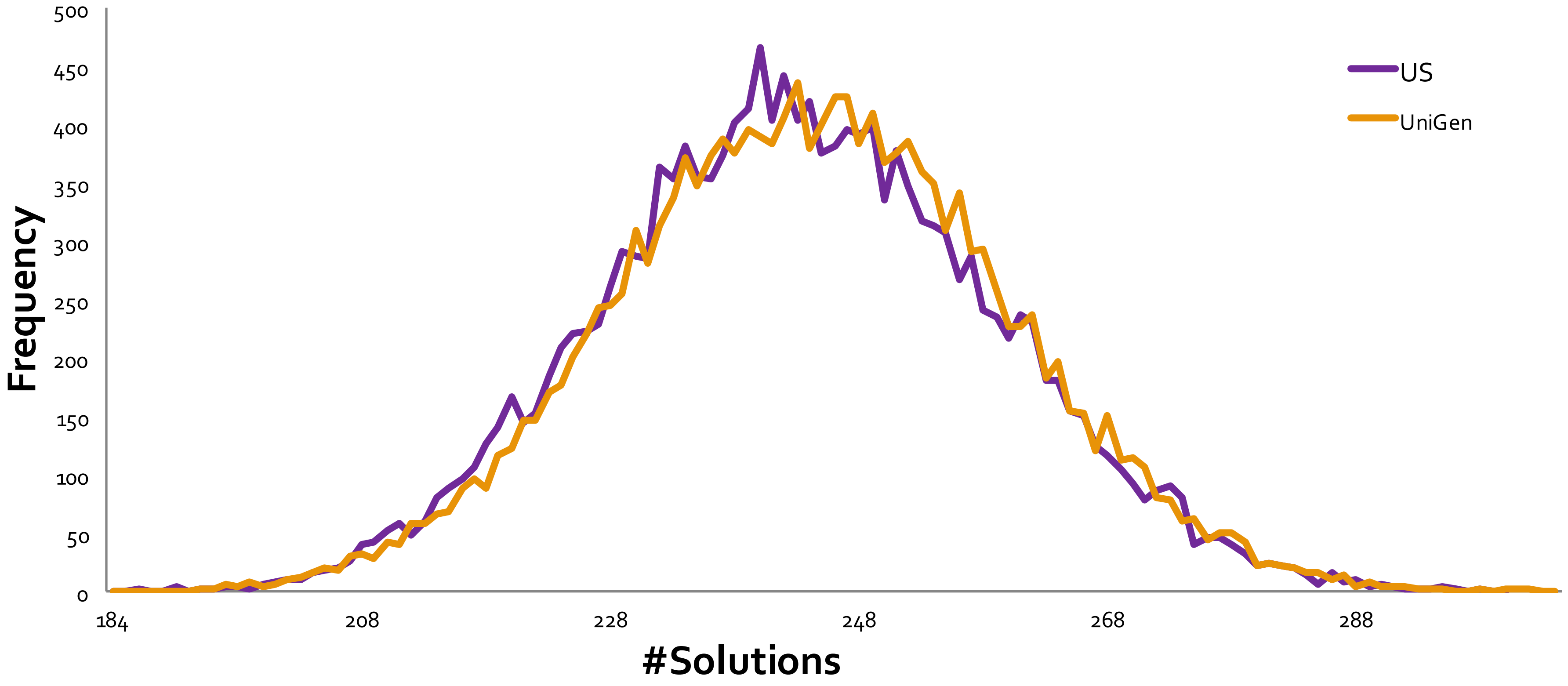


Results: Uniformity



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

Results: Uniformity



- Benchmark: case110.cnf; #var: 287; #clauses: 1263
- Total Runs: $4\text{x}10^6$; Total Solutions : 16384

Contribution of Hashing-based Approaches

- ApproxMC: The first scalable approximate model counter
- UniGen: The first scalable uniform generator
- Outperforms state-of-the-art generators/counters

Towards Efficient Hash Functions

Parity-Based Hashing

- Variables: $X_1, X_2, X_3, \ldots\ldots, X_n$
- To construct h: $\{0,1\}^n \rightarrow \{0,1\}^m$, choose m random XORs
- Pick every variable with prob. ½ , XOR them and add 1 with prob. ½
- E.g.: $X_1 \oplus X_3 \oplus X_6 \oplus \ldots. \oplus X_{n-1}$
- $\alpha \in \{0,1\}^m \rightarrow$ Set every XOR equation to 0 or 1 randomly
- The cell: F ∧ XOR (CNF+XOR)

$$\left.\begin{array}{l}
X_1 \oplus X_3 \oplus X_6 \oplus \ldots.\ X_{n-3} = 0 \\
X_1 \oplus X_2 \oplus X_4 \oplus \ldots.X_{n-1} = 1 \\
X_1 \oplus X_3 \oplus X_5 \oplus \ldots.\ X_{n-2} = 0 \\
X_2 \oplus X_3 \oplus X_4 \oplus \ldots.\ X_{n-1} = 0 \\
\ldots\ldots \\
X_1 \oplus X_2 \oplus X_3 \oplus \ldots.\ X_{n-1} = 0
\end{array}\right\}\ \text{m XORs}$$

Parity-Based Hashing

- Avg Length : n/2
- Smaller parity constraints➔ better performance

How to shorten XOR clauses?

Inspired from Error Correcting Codes

- X = # of solutions in a cell; $\mu_X = \frac{|R_F|}{2^m}$
- 2-universal hashing ensures $\sigma_X^2 \leq \mu_X$
- Key result: Using sparse constraints of size O(log n), we have:

$$\frac{\sigma_X^2}{\mu_X^2} \text{ is monotonically decreasing with X}$$

  - Challenge: Unable to guarantee $\sigma_X^2 \leq \mu_X$; therefore weaker concentration inequalities
- The resulting algorithms require $\theta(n \log n)$ NP calls in comparison to O(log n) calls based on 2-universal hashing algorithms

(Ermon et al 2014, 16; Achlioptas et al. 2015, Asteris et al 2016)

Independent Support

- Set I of variables such that assignments to these uniquely determine assignments to rest of variables (for satisfying assignments)
- If σ_1 and σ_2 agree on I then $\sigma_1 = \sigma_2$
- c ⟷ (a V b) ; Independent Support I: {a, b}
  - {a,c} is NOT an Independent Support
- ***Key Idea***: Hash only on the independent variables
  - Average size of XOR: $\frac{n}{2}$ to $\frac{|I|}{2}$

Formal Definition

Input Formula: F, Solution space: R_F

$$\forall \sigma_1, \sigma_2 \in R_F, \text{ If } \sigma_1 \text{ and } \sigma_2 \text{ agree on I, then } \sigma_1 = \sigma_2$$

$$F(x_1, \ldots, x_n) \wedge F(y_1, \ldots, y_n) \wedge \bigwedge_{i | x_i \in I} (x_i = y_i) \implies \bigwedge_j (x_j = y_j)$$

$$\text{where } F(y_1, \ldots, y_n) = F(x_1 \to y_1, \ldots, x_n \to y_n)$$

Key Idea

$$F(x_1, \ldots, x_n) \wedge F(y_1, \ldots, y_n) \wedge \bigwedge_{i | x_i \in I} (x_i = y_i) \implies \bigwedge_j (x_j = y_j)$$

$$Q_{F,I} = F(x_1, \ldots, x_n) \wedge F(y_1, \ldots, y_n) \wedge \bigwedge_{i | x_i \in I} (x_i = y_i) \wedge \neg \left(\bigwedge_j (x_j = y_j) \right).$$

Theorem: $Q_{F,I}$ is unsatisfiable if and only if I is independent support

Key Idea

$$H_1 = \{x_1 = y_1\},\ \ldots,\ H_n = \{x_n = y_n\}$$

$$\Omega = F(x_1, \ldots, x_n) \wedge F(y_1, \ldots, y_n) \wedge (\neg \bigwedge_j (x_j = y_j))$$

$I = \{x_i\}$ is Independent Support iff $H^I \wedge \Omega$ is unsatisfiable
where $H^I = \{H_i \mid x_i \in I\}$

Minimal Unsatisfiable Subset

- Given $\Psi = H_1 \wedge H_2 \cdots H_m \wedge \Omega$
  - Find subset $\{H_{i1}, H_{i2}, \cdots H_{ik}\}$ of $\{H_1, H_2, \cdots H_m\}$ such that $H_{i1} \wedge H_{i2} \cdots H_{ik} \wedge \Omega$ is UNSAT

    Unsatisfiable subset

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

    Minimal Unsatisfiable subset

Minimal Independent Support

$$H_1 = \{x_1 = y_1\}, \;\ldots,\; H_n = \{x_n = y_n\}$$

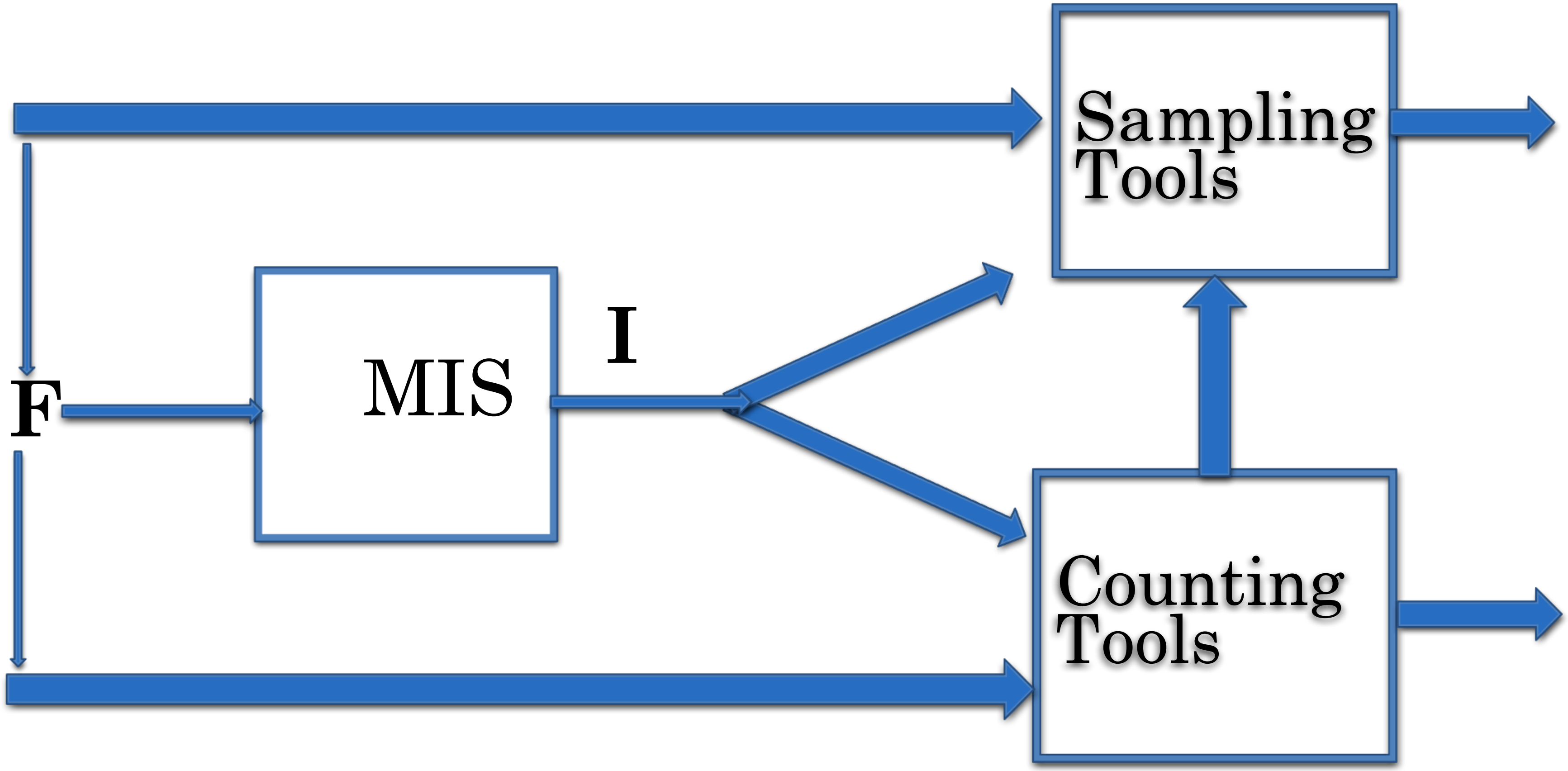
$$\Omega = F(x_1, \ldots, x_n) \wedge F(y_1, \ldots, y_n) \wedge (\neg \bigwedge_j (x_j = y_j))$$

$I = \{x_i\}$ is minimal Independent Support iff H^I is minimal unsatisfiable subset where $H^I = \{H_i \mid x_i \in I\}$

Key Idea

Minimal Independent Support (MIS) → Minimal Unsatisfiable Subset (MUS)

Impact on Sampling and Counting Techniques

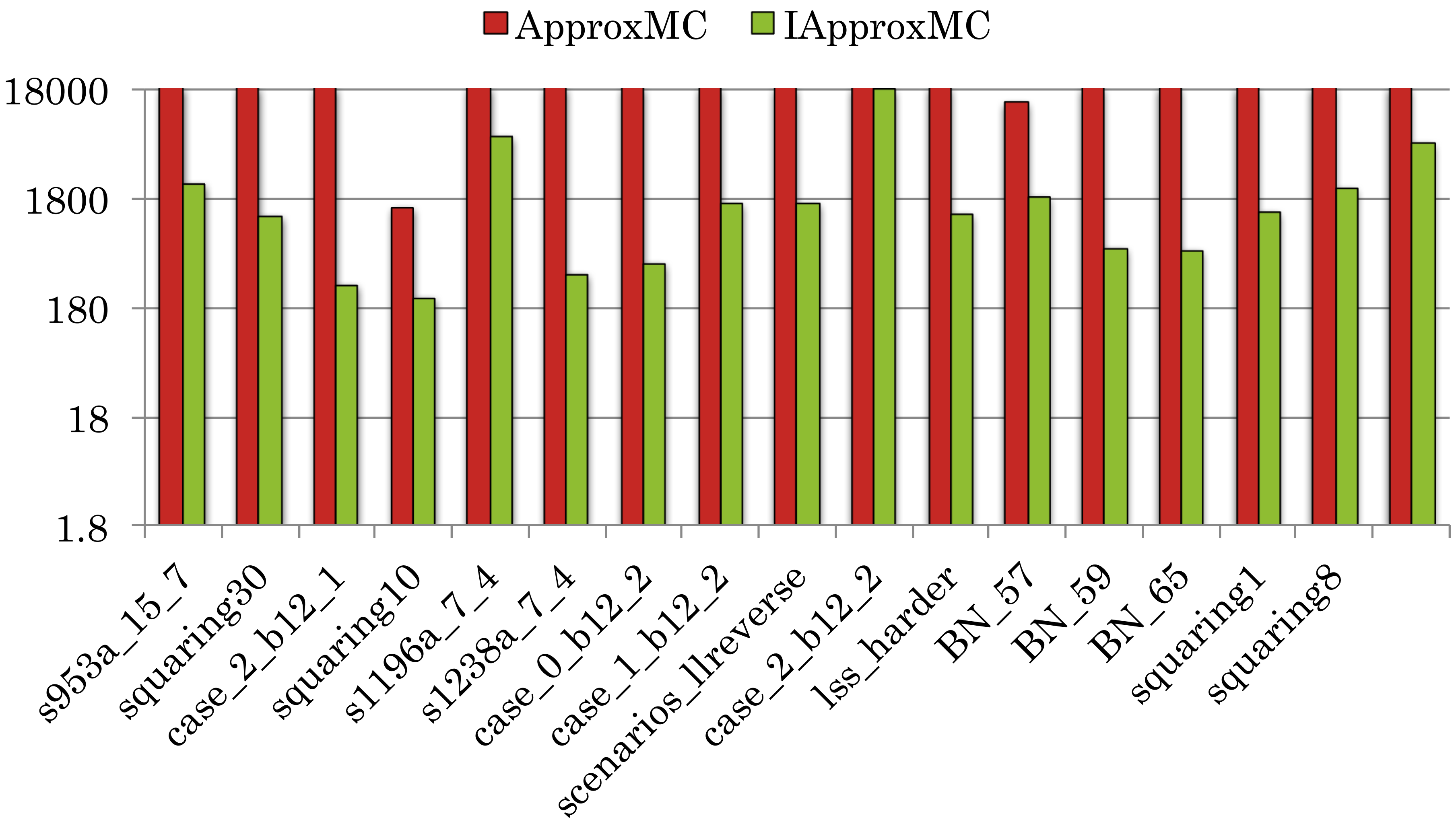


What about complexity

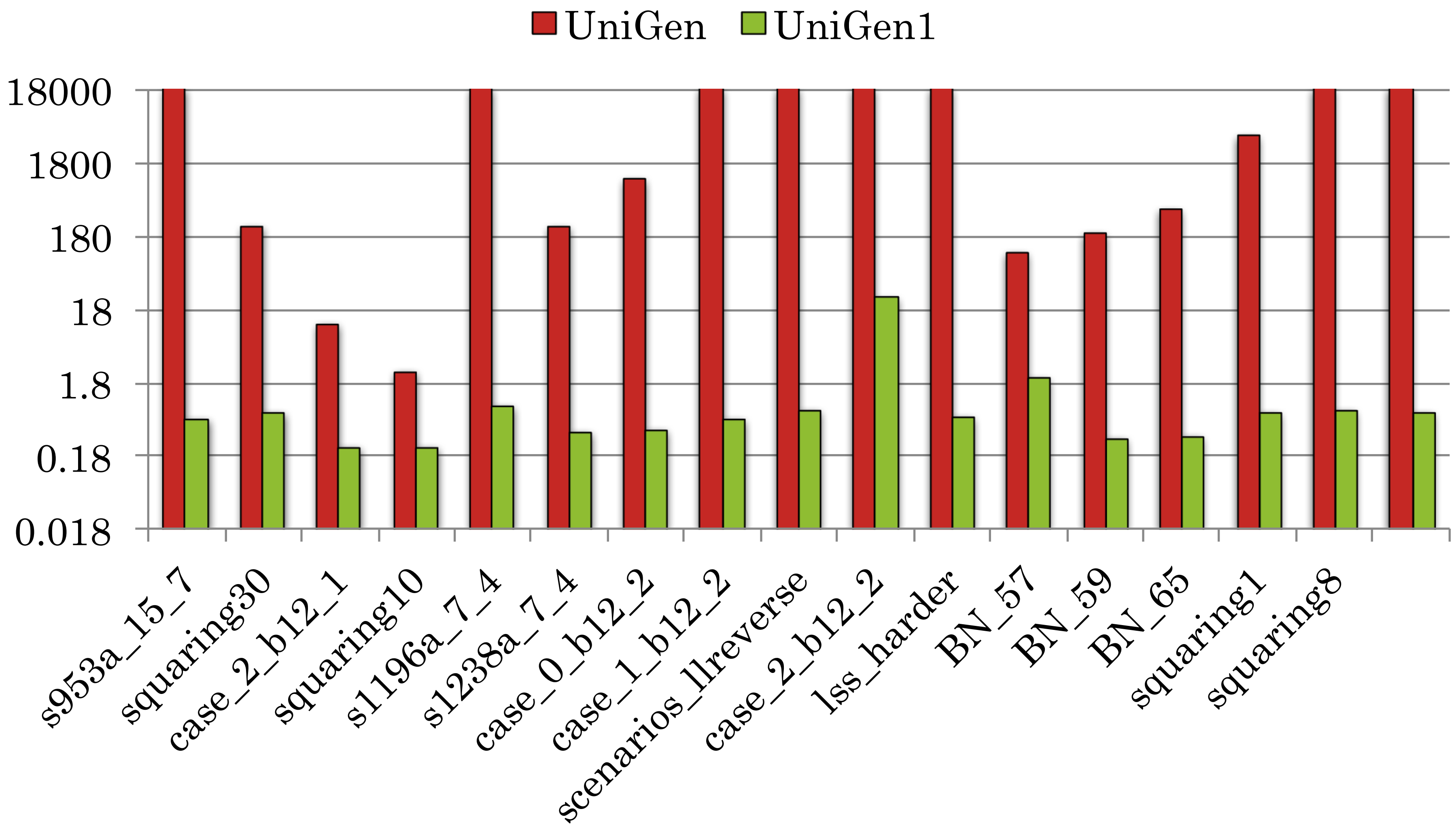
- Computation of MUS: FP^{NP}
- Why solve a FP^{NP} for almost-uniform generation/approximate counter (PTIME PTM with NP Oracle)

Settling the debate through practice!

Performance Impact on Integration



Performance Impact on Uniform Sampling



Future Directions

Extension to More Expressive domains

- Efficient hashing schemes
  - Extending bit-wise XOR to richer constraint domains provides guarantees but fails to harness progress in solving engines for richer domains

- Solvers to handle F + Hash efficiently
  - CryptoMiniSAT has fueled progress for SAT domain
  - Similar solvers for other domains?
- Initial forays with bit-vector constraints and Boolector [AAAI 2016]
  - Uses new linear modular hash function that generalizes XOR-based hash functions
  - Significant speedups compared to bit-blasted versions

Summary

- Sampling and Integration are fundamental problems in Artificial Intelligence.
  - Applications from probabilistic inference, automatic problem generation to system verification.

- Drawback of related approaches: theoretical guarantees or scalability (Choose one)

- Hashing-based approaches promise theoretical guarantees and scalability

Take Away: Hashing-based Approaches

- Theoretical
  - Discrete Integration
    - Reduction of NP calls from O(n log n) to O(log n)
    - Efficient hash functions based on Independent support
  - Sampling
    - Reduction of Approximate Counting calls from O(n) to O(1)
    - Usage of 2-universal hash functions
- Practical
  - From problems with tens of variables (before 2013) to hundreds of thousands of variables

Acknowledgements

Daniel Fremont
(UCB)

Dror Fried
(Rice)

Alexander Ivrii
(IBM)

Sharad Malik
(Princeton)

Rakesh Mistry
(IITB)

Sanjit Seshia
(UCB)

Mate Soos
(CMS)

Thanks!

Questions?

Software and papers are available at http://tinyurl.com/uai16tutorial