Scaling Discrete Integration and Sampling: Foundations and Challenges

Supratik Chakraborty, *IIT Bombay*

Kuldeep S. Meel, National University of Singapore

Outline

- Part 1: Applications
- Part 2: Prior Work
- Part 3: Overview of SAT Solving
- Part 4: Hashing-based Approach for Uniform Distribution
- Part 5: Beyond Propositional
- Part 6: Challenges

Logical breakpoint in Part 4 for coffee break
Slides will be available at https://tinyurl.com/ijcai18tutorial

Notation

- Given
  - $X_1, \dots X_n$: variables with 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 $W(\varphi) = \sum_{y \in \mathrm{Sol}(\varphi)} W(y)$
    If $W(y) = 1$ for all y, then $W(\varphi) = |\mathrm{Sol}(\varphi)|$

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 $\{0, 1\}^n$, R, Z – Bit-vectors, reals, integers

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

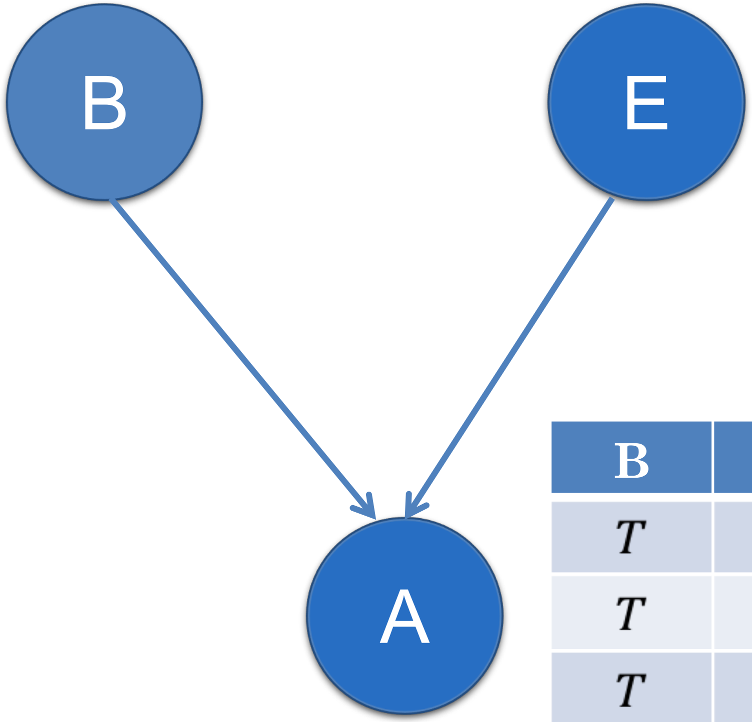
$$\Pr[event_i \mid evidence] = \frac{\Pr[event_i \cap evidence]}{\Pr[evidence]} = \frac{\Pr[event_i \cap evidence]}{\sum_j \Pr[event_j \cap evidence]}$$

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

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

Probablistic Inference: Graphical Models

B	Pr
T	0.8
F	0.2



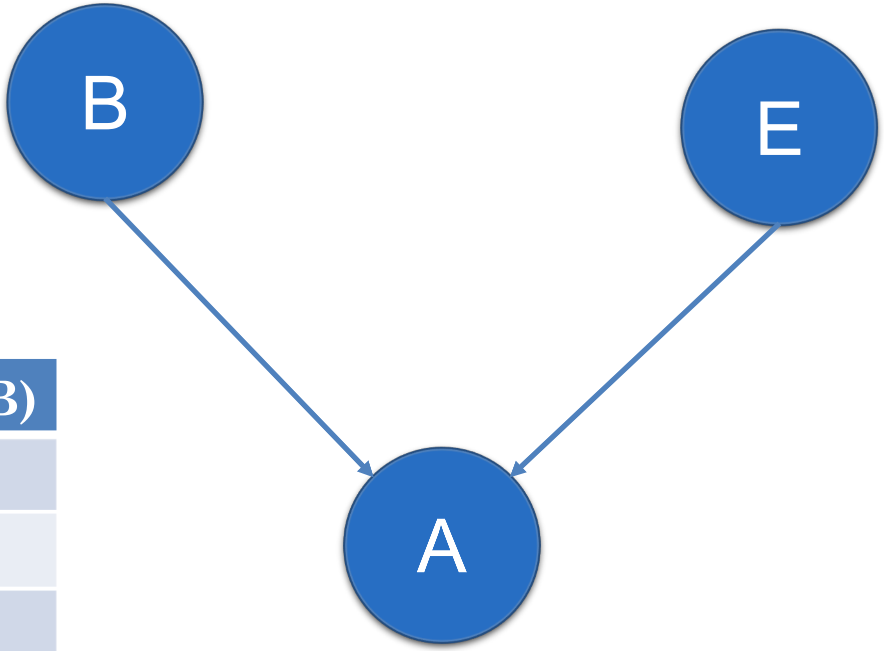
E	Pr
T	0.1
F	0.9

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

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

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:

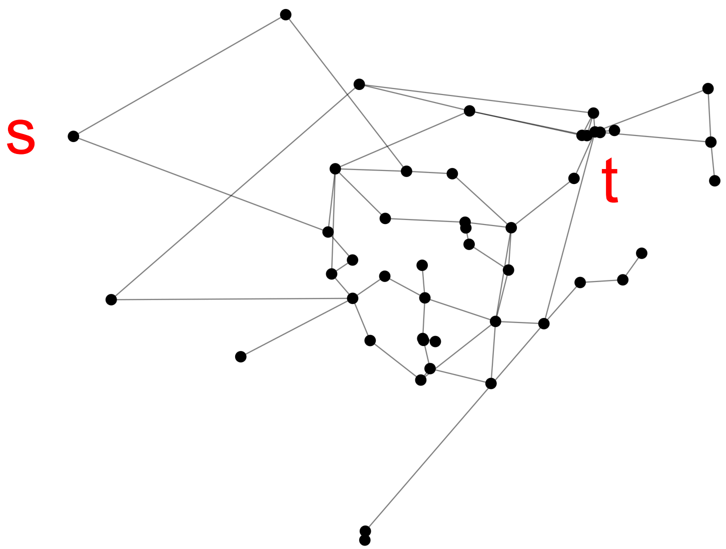
$R_F = \{ (v_A = 1, v_E = 1, v_B = 1, t_{A|B,E} = 1, \text{all else } 0), (v_A = 1, v_E = 1, v_{\sim B} = 1, t_{A|\sim B,E} = 1, \text{all else } 0) \}$

Weight of satisfying assignments of F:

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

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

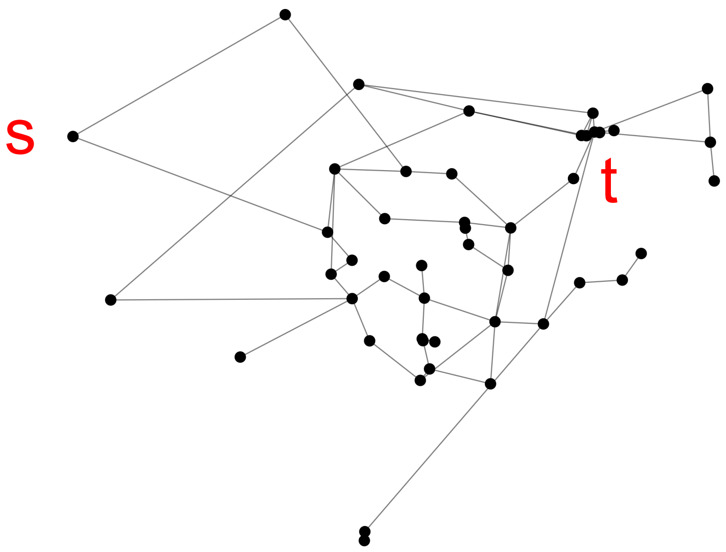
Application 2: Network Reliability



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) ∈ [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

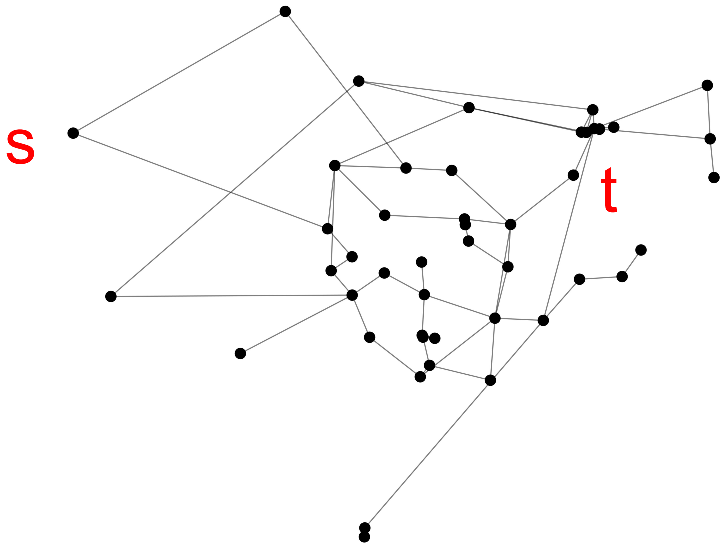
$$\Pr[\pi] = \prod_{e:\,\pi(e)=0} g(e) \times \prod_{e:\,\pi(e)=1} (1 - g(e))$$

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
- $\varphi_{s,t}$ $(p_{v1}, \ldots p_{vn}, q_{e1}, \ldots q_{em})$: 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

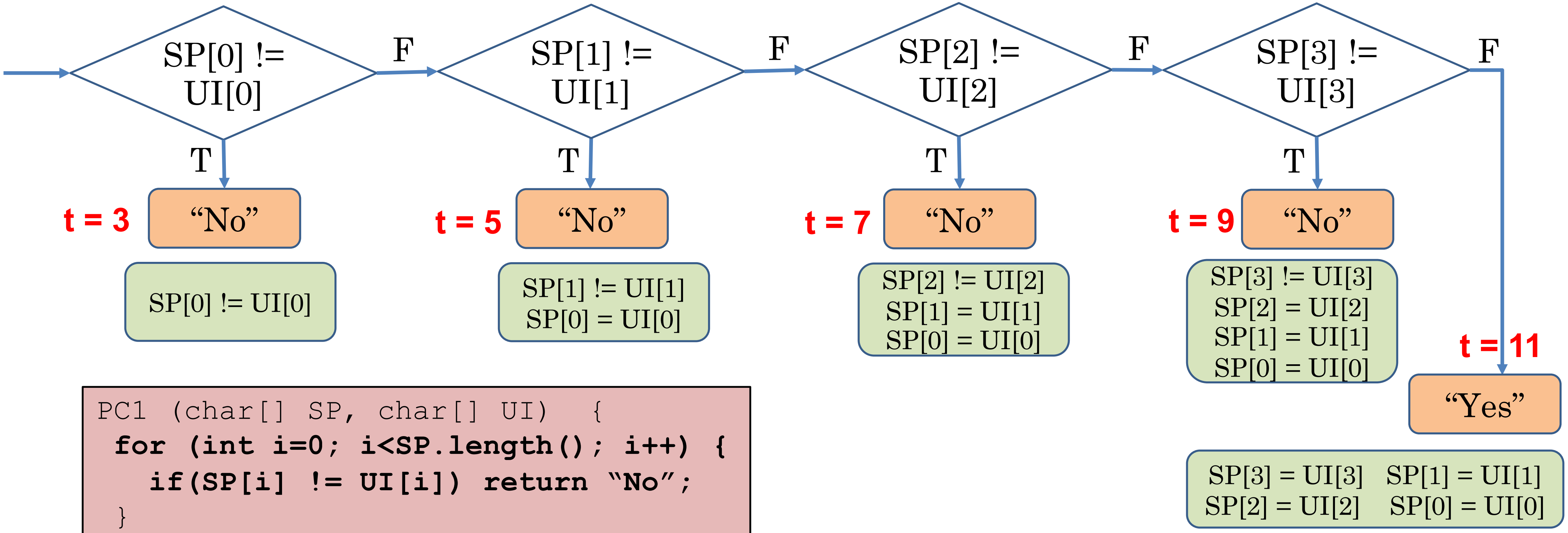
  PC1 executes for loop 4 times

```
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]



SP[0] != UI[0]

F

SP[1] != UI[1]

F

SP[2] != UI[2]

F

SP[3] != UI[3]

F

T

t = 3 "No"

SP[0] != UI[0]

T

t = 5 "No"

SP[1] != UI[1]
SP[0] = UI[0]

T

t = 7 "No"

SP[2] != UI[2]
SP[1] = UI[1]
SP[0] = UI[0]

T

t = 9 "No"

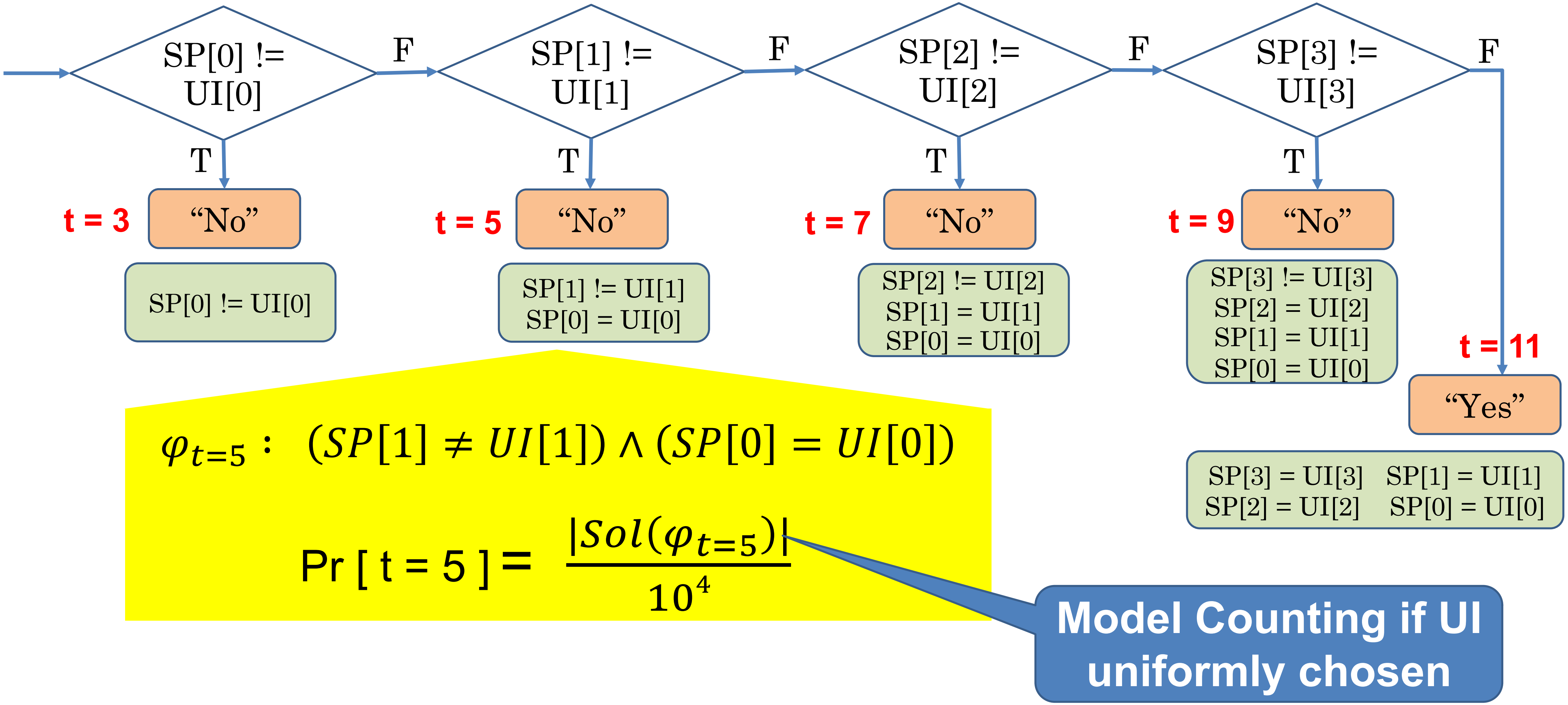
SP[3] != UI[3]
SP[2] = UI[2]
SP[1] = UI[1]
SP[0] = UI[0]

t = 11 "Yes"

SP[3] = UI[3] SP[1] = UI[1]
SP[2] = UI[2] SP[0] = UI[0]

```
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



SP[0] != UI[0]

F

SP[1] != UI[1]

F

SP[2] != UI[2]

F

SP[3] != UI[3]

F

T

t = 3 "No"

SP[0] != UI[0]

T

t = 5 "No"

SP[1] != UI[1]
SP[0] = UI[0]

T

t = 7 "No"

SP[2] != UI[2]
SP[1] = UI[1]
SP[0] = UI[0]

T

t = 9 "No"

SP[3] != UI[3]
SP[2] = UI[2]
SP[1] = UI[1]
SP[0] = UI[0]

t = 11 "Yes"

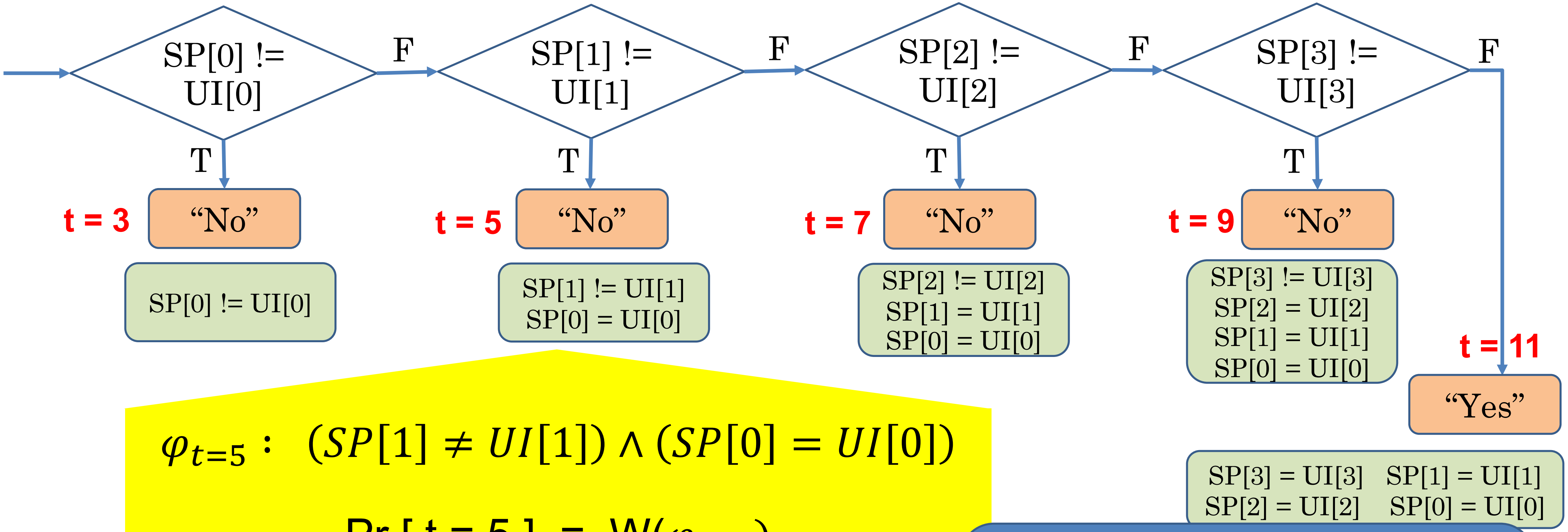
SP[3] = UI[3] SP[1] = UI[1]
SP[2] = UI[2] SP[0] = UI[0]

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

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

Model Counting if UI uniformly chosen

QIF: Probabilities of Observed Times



SP[0] != UI[0]

F

SP[1] != UI[1]

F

SP[2] != UI[2]

F

SP[3] != UI[3]

F

T

t = 3

"No"

SP[0] != UI[0]

T

t = 5

"No"

SP[1] != UI[1]
SP[0] = UI[0]

T

t = 7

"No"

SP[2] != UI[2]
SP[1] = UI[1]
SP[0] = UI[0]

T

t = 9

"No"

SP[3] != UI[3]
SP[2] = UI[2]
SP[1] = UI[1]
SP[0] = UI[0]

t = 11

"Yes"

SP[3] = UI[3]   SP[1] = UI[1]
SP[2] = UI[2]   SP[0] = UI[0]

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

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

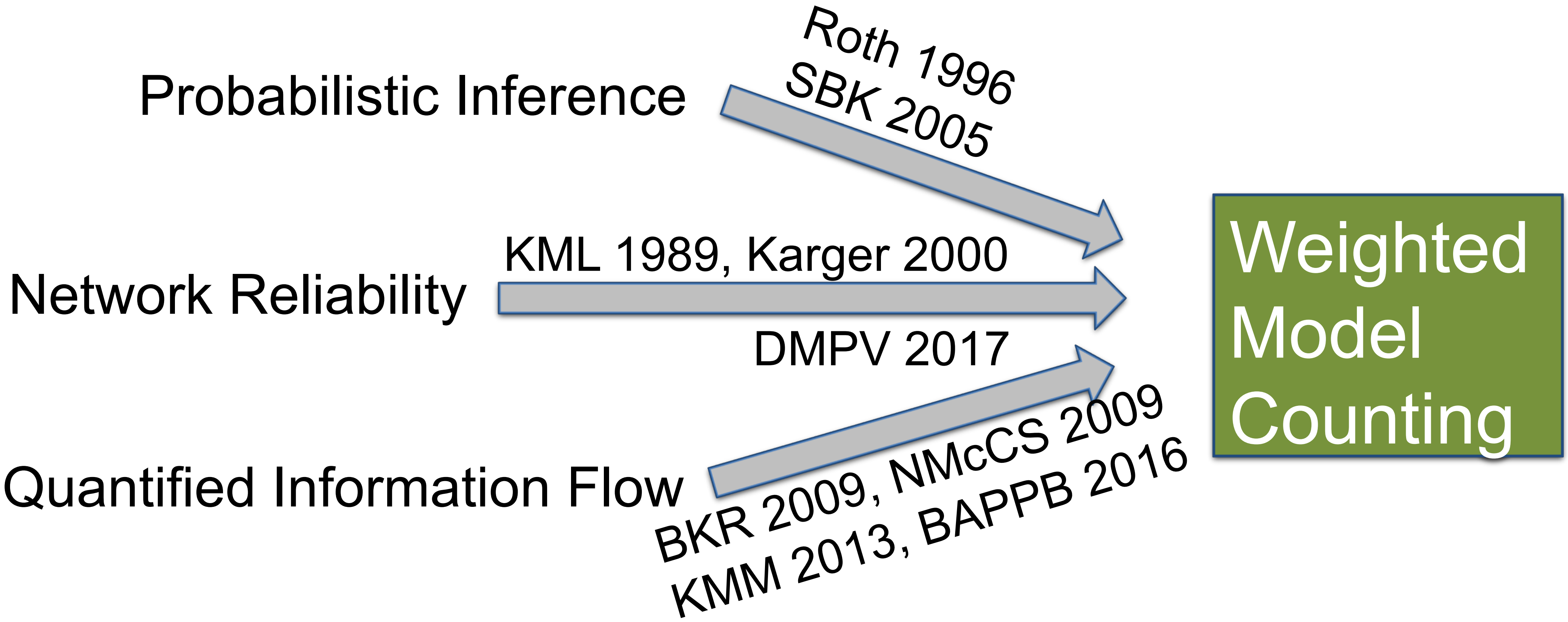
Discrete Integration if UI chosen according to weight function

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

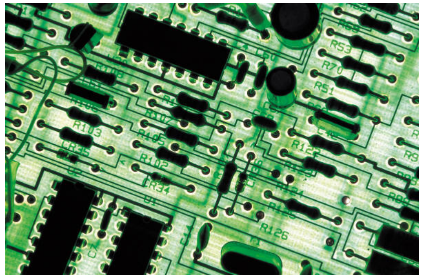
Weighted Model Counting

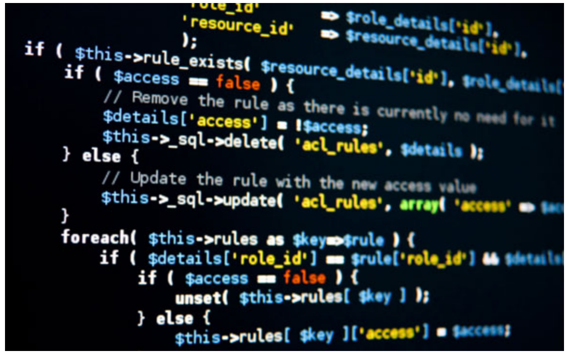
Weighted Model Counting → Unweighted Model Counting

IJCAI 2015

Reduction polynomial in #bits representing weights

Application 4: Constr Random Verification





```
                  'resource_id'    => $role_details['id'],
                  );                => $resource_details['id'],
if ( $this->rule_exists( $resource_details['id'], $role_details[
    if ( $access == false ) {
        // Remove the rule as there is currently no need for it
        $details['access'] = !$access;
        $this->_sql->delete( 'acl_rules', $details );
    } else {
        // Update the rule with the new access value
        $this->_sql->update( 'acl_rules', array( 'access' => $ac
    }
    foreach( $this->rules as $key=>$rule ) {
        if ( $details['role_id'] == $rule['role_id'] && $details
            if ( $access == false ) {
                unset( $this->rules[ $key ] );
            } else {
                $this->rules[ $key ]['access'] = $access;
```

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

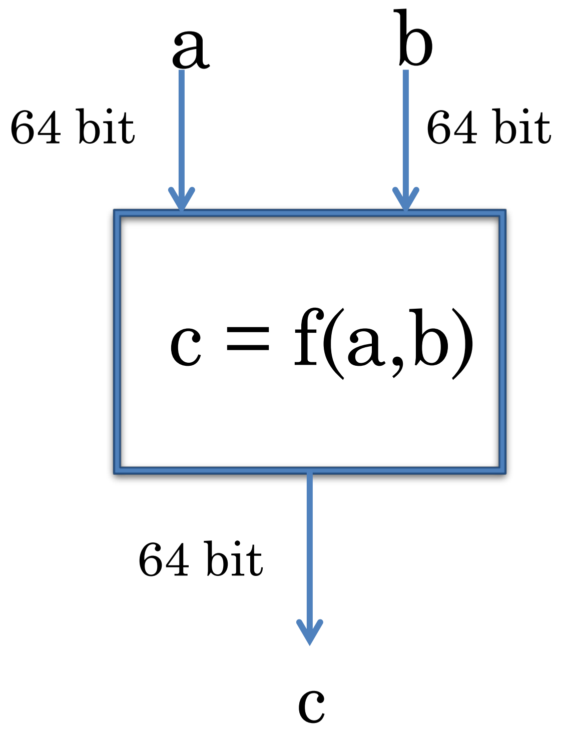
- How do we generate test vectors?

Challenge: Exceedingly large test input space!

Can't try all input combinations

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

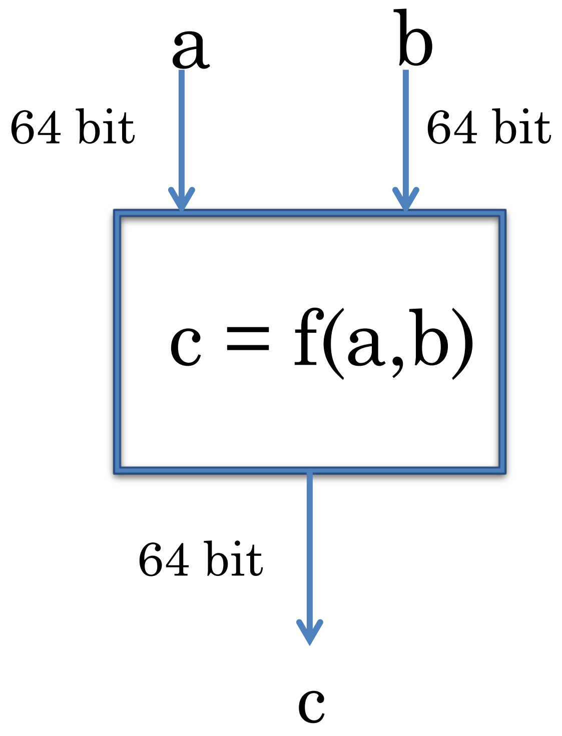
CRV: Sources of Constraints



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

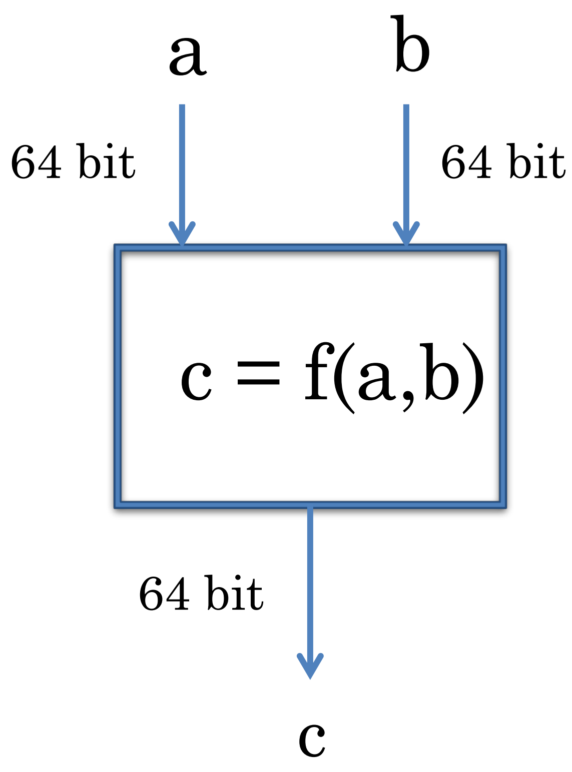
Modern SAT/SMT solvers are complex systems

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

Outline

- Part 1: Applications
- Part 2: Prior Work
- Part 3: Overview of SAT Solving
- Part 4: Hashing-based Approach for Uniform Distribution
- Part 5: Beyond Propositional
- Part 6: Challenges

Logical breakpoint in Part 4 for coffee break
Slides will be available at https://tinyurl.com/ijcai18tutorial

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]

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

Randomized: Poly-time probabilistic Turing Machine with NP oracle

[Stockmeyer 1983; Jerrum,Valiant,Vazirani 1986]

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

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]

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

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

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

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

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

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

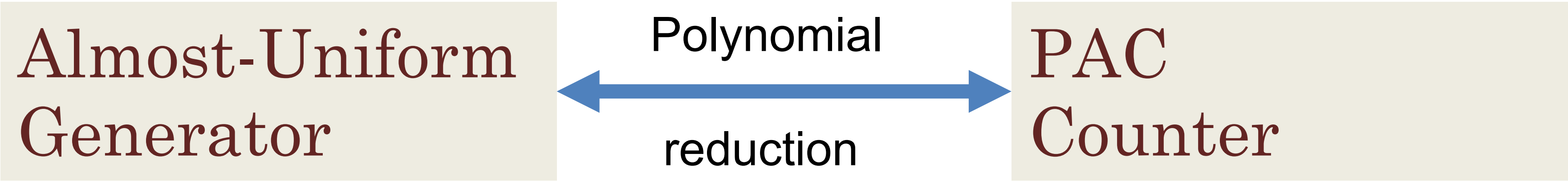
$$\Pr[y = \mathrm{BGP(F)}] = \begin{cases} 0 \text{ if } y \notin \mathrm{R_F} \\ c\ (>0) \text{ if } y \in \mathrm{R_F}, \text{ where } c \text{ is independent of } y \end{cases}$$

  - 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

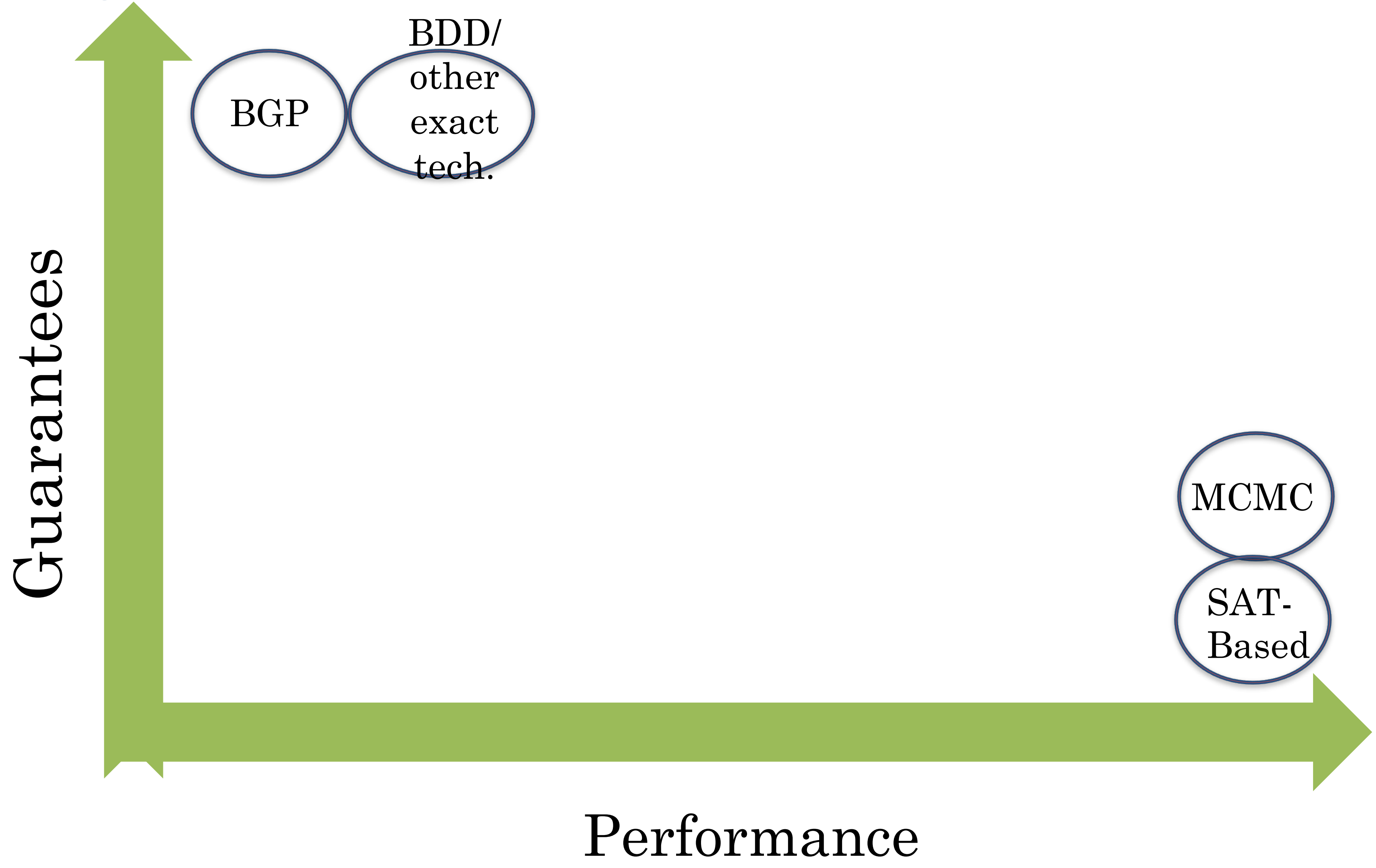
Approximate Integration and Sampling: Close Cousins

- Seminal paper by Jerrum, Valiant, Vazirani 1986



- 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

Outline

Part 1: Applications
Part 2: Prior Work
Part 3: Overview of SAT Solving
Part 4: Hashing-based Approach for Uniform Distribution
Part 5: Beyond Propositional
Part 6: Challenges

Part III

Overview of SAT Solving

A Tale of Constraints

Boolean Satisfiability (SAT): Given a Boolean expression, using “and” ($\wedge$), “or” ($\vee$), and “not” ($\neg$) is there a solution, i.e., an assignment of 0’s and 1’s to the variables that makes the expression equal 1?

Example: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_3)$

$x_1 = 1, x_2 = 1, x_3 = 1$

A Tale of Constraints

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

Example: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_3)$

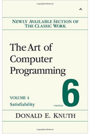
$x_1 = 1, x_2 = 1, x_3 = 1$

Ernst Schroder, 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: SAT is NP-complete

The Tale of Triumph of SAT Solvers

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. (Donald Knuth, 2016)



NEWLY AVAILABLE SECTION OF
THE CLASSIC WORK

The Art of
Computer
Programming

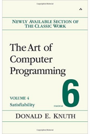
VOLUME 4
Satisfiability

6

DONALD E. KNUTH

The Tale of Triumph of SAT Solvers

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. (Donald Knuth, 2016)



Industrial usage of SAT Solvers: hardware verification, planning, Genome Rearrangement, Telecom Feature Subscription, Resource Constrained Scheduling, Noise Analysis, Games, $\cdots$

Resolution

- Resolution rule: [DP60,R65]

$$\frac{(\alpha \vee x) \qquad\qquad (\beta \vee \bar{x})}{(\alpha \vee \beta)}$$

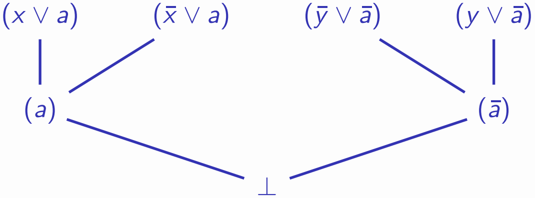
  - Complete proof system for propositional logic

Resolution

- Resolution rule: [DP60,R65]

$$\frac{(\alpha \vee x) \qquad\qquad (\beta \vee \bar{x})}{(\alpha \vee \beta)}$$

  - Complete proof system for propositional logic



$(x \vee a)$ $(\bar{x} \vee a)$ $(\bar{y} \vee \bar{a})$ $(y \vee \bar{a})$

(a) $(\bar{a})$

$\perp$

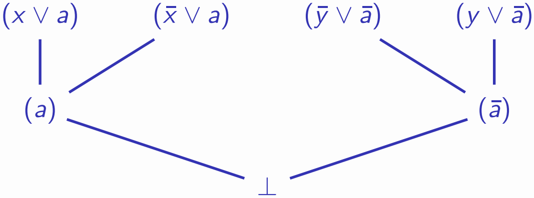
  - Extensively used with (CDCL) SAT solvers

Resolution

- **Resolution rule**: [DP60,R65]

$$\frac{(\alpha \lor x) \qquad\qquad (\beta \lor \bar{x})}{(\alpha \lor \beta)}$$

  - Complete proof system for propositional logic



  - Extensively used with (CDCL) SAT solvers

- **Self-subsuming** resolution (with $\alpha' \subseteq \alpha$): [E.g. SP04,EB05]

$$\frac{(\alpha \lor x) \qquad\qquad (\alpha' \lor \bar{x})}{(\alpha)}$$

  - (α) subsumes $(\alpha \lor x)$

Unit propagation

$$\begin{aligned}\mathcal{F} \quad = \quad & (r) \land (\bar{r} \lor s) \land \\ & (\bar{w} \lor a) \land (\bar{x} \lor \bar{a} \lor b) \land \\ & (\bar{y} \lor \bar{z} \lor c) \land (\bar{b} \lor \bar{c} \lor d)\end{aligned}$$

Unit propagation

$$\begin{aligned}\mathcal{F} \quad = \quad & (r) \wedge (\bar{r} \vee s) \wedge \\ & (\bar{w} \vee a) \wedge (\bar{x} \vee \bar{a} \vee b) \wedge \\ & (\bar{y} \vee \bar{z} \vee c) \wedge (\bar{b} \vee \bar{c} \vee d)\end{aligned}$$

- What can we deduce?

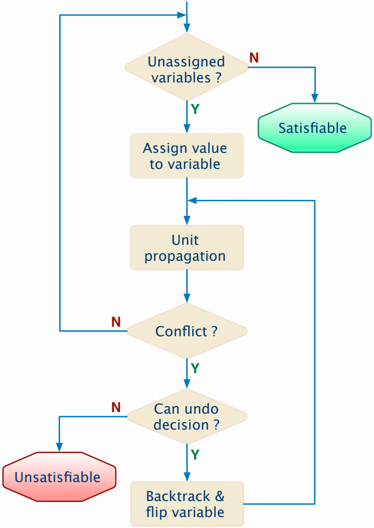
Unit propagation

$$\begin{aligned}\mathcal{F} &= (r) \wedge (\bar{r} \vee s) \wedge \\ &\quad (\bar{w} \vee a) \wedge (\bar{x} \vee \bar{a} \vee b) \wedge \\ &\quad (\bar{y} \vee \bar{z} \vee c) \wedge (\bar{b} \vee \bar{c} \vee d)\end{aligned}$$

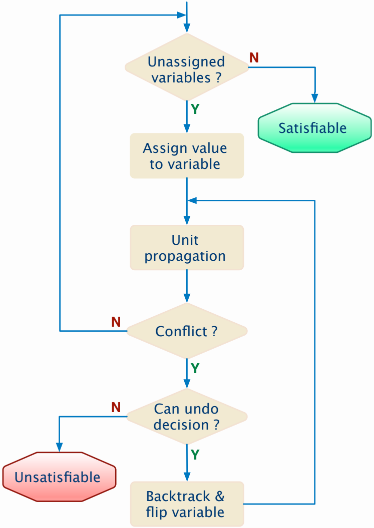
- What can we deduce?
- $s = 1$

The DPLL algorithm

[DL60,DLL62]



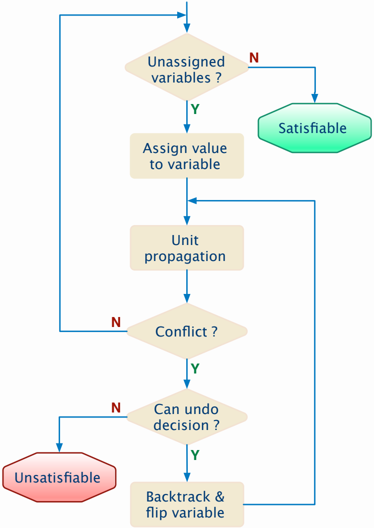
The DPLL algorithm



[DL60,DLL62]

$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

The DPLL algorithm


Unassigned variables ?

N

Satisfiable

Y

Assign value to variable

Unit propagation

Conflict ?

N

Y

Can undo decision ?

N

Unsatisfiable

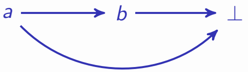
Y

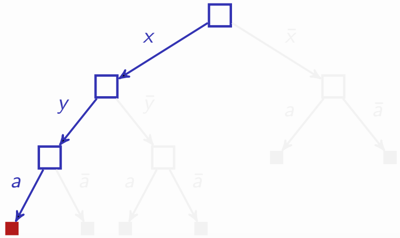
Backtrack & flip variable

[DL60,DLL62]

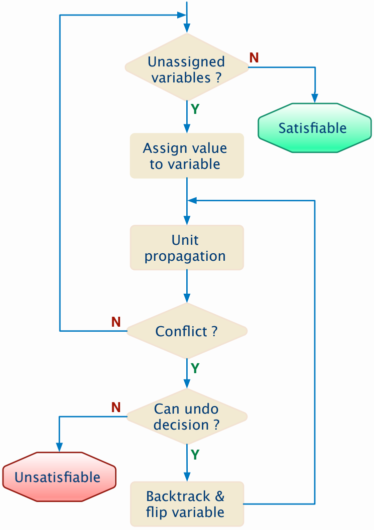
$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

Level	Dec.	Unit Prop.
0	$\emptyset$	
1	x	
2	y	
3	a	b $\bot$





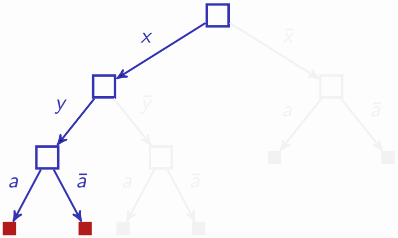
The DPLL algorithm



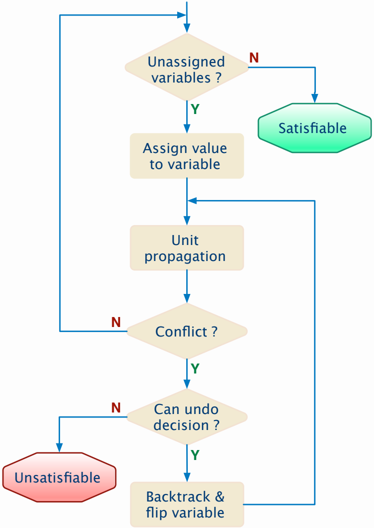
[DL60,DLL62]

$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

Level	Dec.	Unit Prop.
0	$\emptyset$	
1	x	
2	y	
3	$\bar{a}$	$\longrightarrow \bar{b} \longrightarrow \perp$



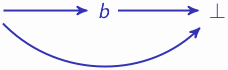
The DPLL algorithm

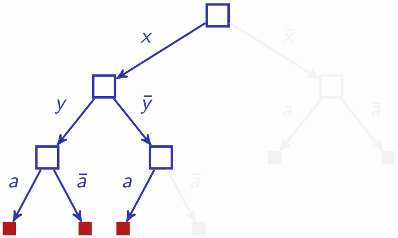


[DL60,DLL62]

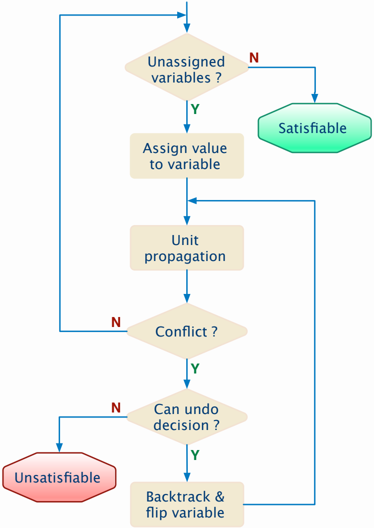
$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

Level	Dec.	Unit Prop.
0	$\emptyset$	
1	x	
2	$\bar{y}$	
3	a	b $\perp$





The DPLL algorithm



Unassigned variables ?

N

Satisfiable

Y

Assign value to variable

Unit propagation

Conflict ?

N

Y

Can undo decision ?

N

Unsatisfiable

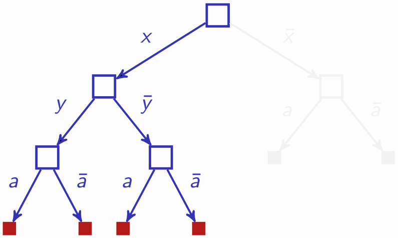
Y

Backtrack & flip variable

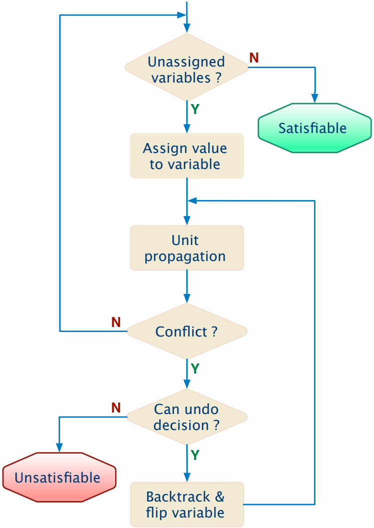
[DL60,DLL62]

$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

Level	Dec.	Unit Prop.
0	$\emptyset$	
1	x	
2	$\bar{y}$	
3	$\bar{a}$	$\rightarrow \bar{b} \rightarrow \perp$



The DPLL algorithm



Unassigned variables ?

N

Satisfiable

Y

Assign value to variable

Unit propagation

Conflict ?

N

Y

Can undo decision ?

N

Unsatisfiable

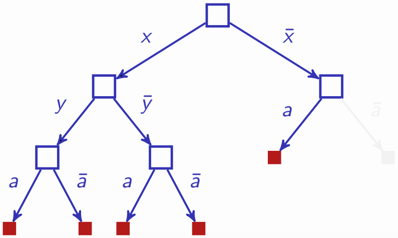
Y

Backtrack & flip variable

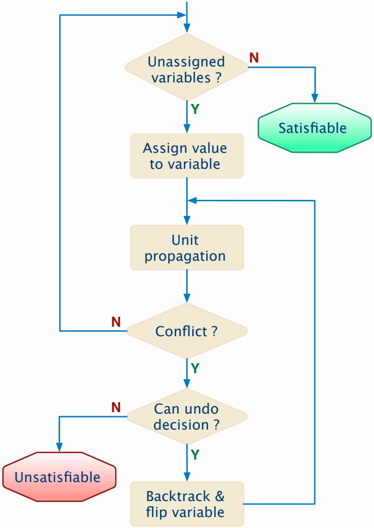
[DL60,DLL62]

$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

Level	Dec.	Unit Prop.
0	$\emptyset$	
1	$\bar{x}$	$\longrightarrow y$
2	a	$\longrightarrow b \longrightarrow \perp$



The DPLL algorithm



Unassigned variables ?

N

Satisfiable

Y

Assign value to variable

Unit propagation

N

Conflict ?

Y

N

Can undo decision ?

Y

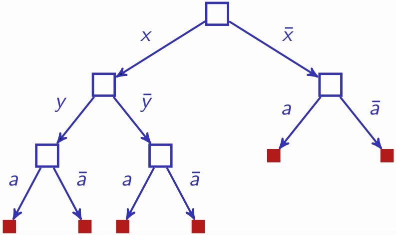
Unsatisfiable

Backtrack & flip variable

[DL60,DLL62]

$$\mathcal{F} = (x \vee y) \wedge (a \vee b) \wedge (\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee \bar{b})$$

$$\begin{tikzcd} \text{Level} & \text{Dec.} & \text{Unit Prop.} \\ 0 & \emptyset \\ 1 & \bar{x} \arrow[r] & y \\ 2 & \bar{a} \arrow[r] \arrow[rr, bend right] & \bar{b} \arrow[r] & \bot \end{tikzcd}$$



What is a CDCL SAT solver?

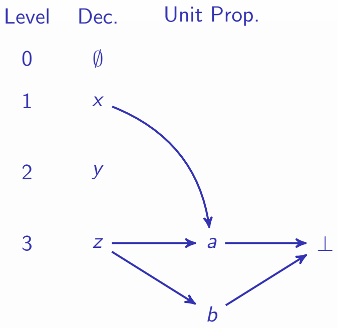
- Extend DPLL SAT solver with: [DP60,DLL62]
  - Clause learning & non-chronological backtracking [MSS96a,MSS99,BS97,Z97]
  - Search restarts [GSK98,BMS00,H07,B08]
  - Lazy data structures
  - Conflict-guided branching
  - ...

What is a CDCL SAT solver?

- Extend DPLL SAT solver with: [DP60,DLL62]
    - Clause learning & non-chronological backtracking [MSS96a,MSS99,BS97,Z97]
        - Exploit UIPs [MSS96a,SSS12]
        - Minimize learned clauses [SB09,VG09]
        - Opportunistically delete clauses [MSS96a,MSS99,GN02]
    - Search restarts [GSK98,BMS00,H07,B08]
    - Lazy data structures
        - Watched literals [MMZZM01]
    - Conflict-guided branching
        - Lightweight branching heuristics [MMZZM01]
        - Phase saving [S00,PD07]
    - ...

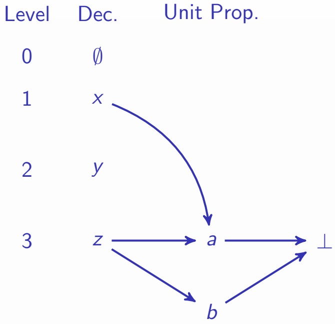
Clause learning

$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$



Clause learning

$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$

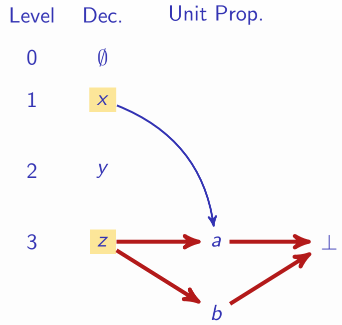


- Analyze conflict

[MSS96a,MSS96b,MSS96c,MSS96d,MSS99]

Clause learning

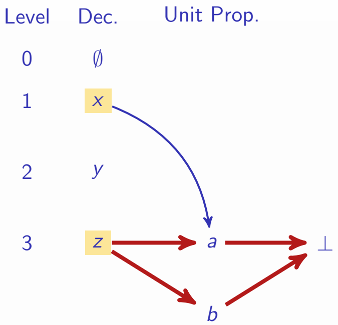
$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$



- Analyze conflict [MSS96a,MSS96b,MSS96c,MSS96d,MSS99]
    - Reasons: x and z
        - Decision variable & literals assigned at decision levels less than current

Clause learning

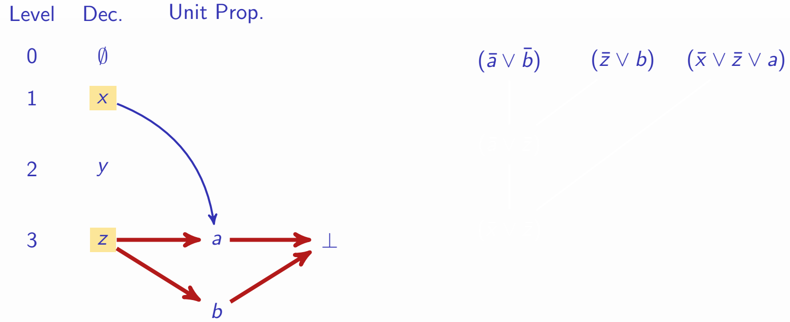
$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$



- Analyze conflict [MSS96a,MSS96b,MSS96c,MSS96d,MSS99]
  - Reasons: x and z
    - Decision variable & literals assigned at decision levels less than current
  - Create **new** clause: $(\bar{x} \vee \bar{z})$

Clause learning

$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$



Level Dec. Unit Prop.

0 $\emptyset$

1 x

2 y

3 z

a

$\perp$

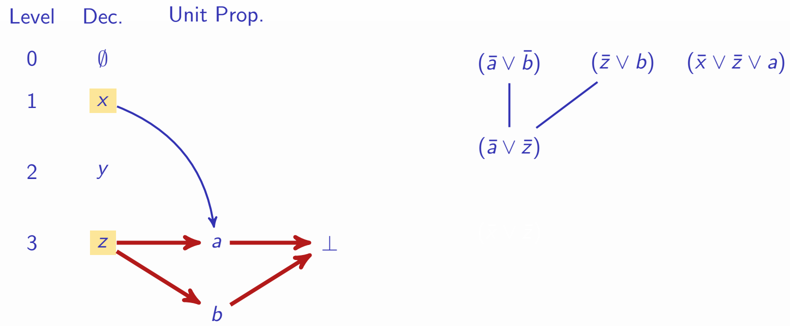
b

$(\bar{a} \vee \bar{b})$ $(\bar{z} \vee b)$ $(\bar{x} \vee \bar{z} \vee a)$

- Analyze conflict [MSS96a,MSS96b,MSS96c,MSS96d,MSS99]
  - Reasons: x and z
    - Decision variable & literals assigned at decision levels less than current
  - Create **new** clause: $(\bar{x} \vee \bar{z})$
- Can relate clause learning with resolution

Clause learning

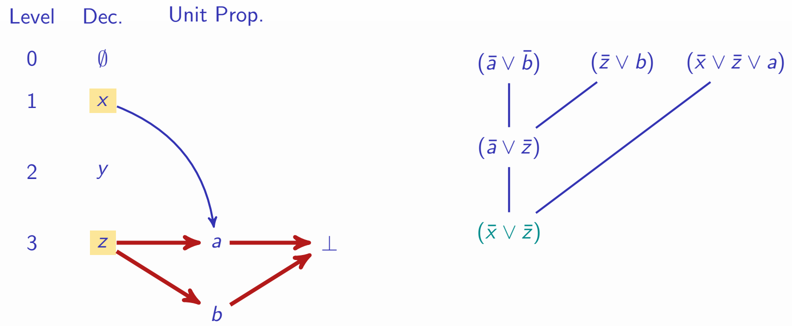
$(\bar{a} \lor \bar{b}) \land (\bar{z} \lor b) \land (\bar{x} \lor \bar{z} \lor a) \land (y \lor b)$



- Analyze conflict [MSS96a,MSS96b,MSS96c,MSS96d,MSS99]
  - Reasons: x and z
    - Decision variable & literals assigned at decision levels less than current
  - Create **new** clause: $(\bar{x} \lor \bar{z})$
- Can relate clause learning with resolution

Clause learning

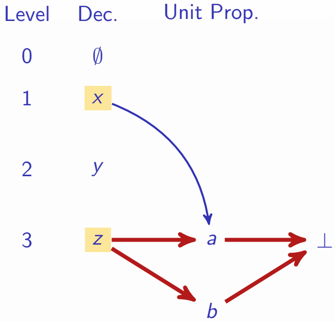
$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$

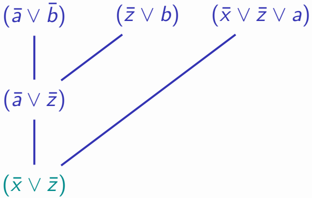


- Analyze conflict [MSS96a,MSS96b,MSS96c,MSS96d,MSS99]
  - Reasons: x and z
    - Decision variable & literals assigned at decision levels less than current
  - Create **new** clause: $(\bar{x} \vee \bar{z})$
- Can relate clause learning with resolution

Clause learning

$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$



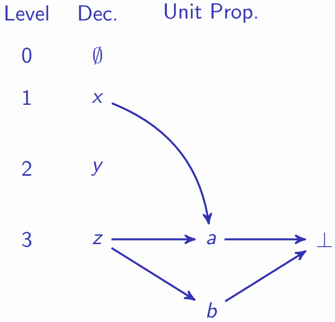


- Analyze conflict

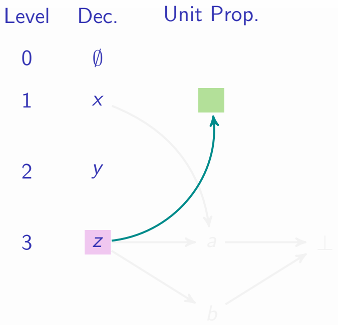
[MSS96a,MSS96b,MSS96c,MSS96d,MSS99]

  - Reasons: x and z
    - Decision variable & literals assigned at decision levels less than current
  - Create **new** clause: $(\bar{x} \vee \bar{z})$
- Can relate clause learning with resolution
  - Learned clauses result from (**selected**) resolution operations

Clause learning – after backtracking

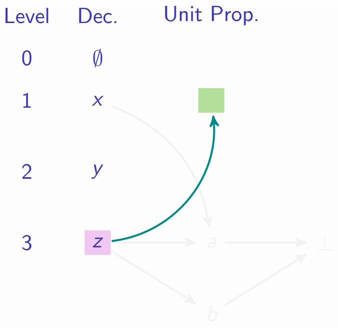


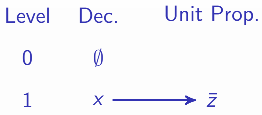
Clause learning – after backtracking



- Clause $(\bar{x} \vee \bar{z})$ is asserting at decision level 1

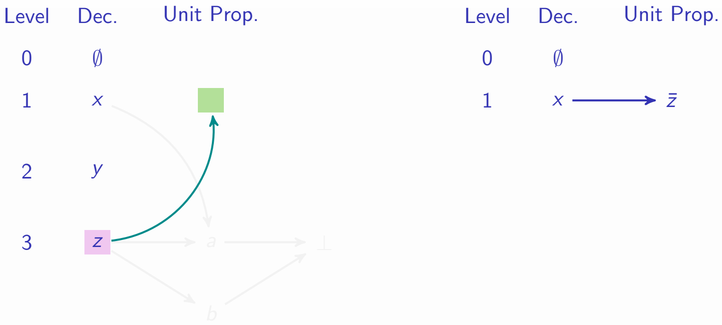
Clause learning – after backtracking





- Clause $(\bar{x} \vee \bar{z})$ is asserting at decision level 1

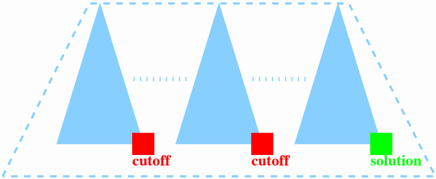
Clause learning – after backtracking



- Clause $(\bar{x} \vee \bar{z})$ is asserting at decision level 1
- Learned clauses are asserting (with exceptions) [MSS96a,MSS99]
- Backtracking differs from plain DPLL:
  - Always bactrack after a conflict [ZMMM01]

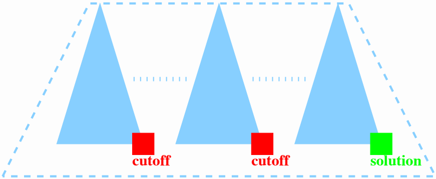
Search restarts

- Restart search after a number of conflicts



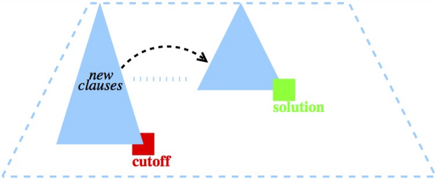
Search restarts

- Restart search after a number of conflicts
- Increase cutoff after each restart
  - Guarantees completeness
  - Different policies exist



Search restarts

- Restart search after a number of conflicts
- Increase cutoff after each restart
    - Guarantees completeness
    - Different policies exist
- Learned clauses effective after restart(s)



Data structures basics

- Each literal l should access clauses containing l
  - Why?

Data structures basics

- Each literal l should access clauses containing l
  - Why? Unit propagation

Data structures basics

- Each literal l should access clauses containing l
    - Why? Unit propagation
- Clause with k literals results in k references, from literals to the clause

Data structures basics

- Each literal l should access clauses containing l
  - Why? Unit propagation
- Clause with k literals results in k references, from literals to the clause
- Number of clause references **equals** number of literals, L

Data structures basics

- Each literal l should access clauses containing l
  - Why? Unit propagation
- Clause with k literals results in k references, from literals to the clause
- Number of clause references **equals** number of literals, L
  - Clause learning can generate **large** clauses
    - Worst-case size: $\mathcal{O}(n)$

Data structures basics

- Each literal l should access clauses containing l
    - Why? Unit propagation
- Clause with k literals results in k references, from literals to the clause
- Number of clause references **equals** number of literals, L
    - Clause learning can generate **large** clauses
        - Worst-case size: $\mathcal{O}(n)$
    - Worst-case number of literals: $\mathcal{O}(m\,n)$

Data structures basics

- Each literal l should access clauses containing l
  - Why? Unit propagation

- Clause with k literals results in k references, from literals to the clause
- Number of clause references **equals** number of literals, L
  - Clause learning can generate **large** clauses
    - Worst-case size: $\mathcal{O}(n)$
  - Worst-case number of literals: $\mathcal{O}(m\,n)$
  - In practice,

    Unit propagation slow-down worse than linear as clauses are learned **!**

Data structures basics

- Each literal l should access clauses containing l
  - Why? Unit propagation
- Clause with k literals results in k references, from literals to the clause
- Number of clause references **equals** number of literals, L
  - Clause learning can generate **large** clauses
    - Worst-case size: $\mathcal{O}(n)$
  - Worst-case number of literals: $\mathcal{O}(m\,n)$
  - In practice,

    Unit propagation slow-down worse than linear as clauses are learned **!**
- Clause learning to be effective requires a more efficient representation:

Data structures basics

- Each literal l should access clauses containing l
  - Why? Unit propagation

- Clause with k literals results in k references, from literals to the clause
- Number of clause references **equals** number of literals, L
  - Clause learning can generate **large** clauses
    - Worst-case size: $\mathcal{O}(n)$
  - Worst-case number of literals: $\mathcal{O}(m\,n)$
  - In practice,

    Unit propagation slow-down worse than linear as clauses are learned **!**

- Clause learning to be effective requires a more efficient representation: **Watched Literals** [MMZZM01]

Data structures basics

- Each literal l should access clauses containing l
  - Why? Unit propagation
- Clause with k literals results in k references, from literals to the clause
- Number of clause references **equals** number of literals, L
  - Clause learning can generate **large** clauses
    - Worst-case size: $\mathcal{O}(n)$
  - Worst-case number of literals: $\mathcal{O}(m\,n)$
  - In practice,

    Unit propagation slow-down worse than linear as clauses are learned **!**
- Clause learning to be effective requires a more efficient representation: **Watched Literals** [MMZZM01]
  - Keep track of only two literals per clause

Data structures basics

- Each literal l should access clauses containing l
  - Why? Unit propagation
- Clause with k literals results in k references, from literals to the clause
- Number of clause references **equals** number of literals, L
  - Clause learning can generate **large** clauses
    - Worst-case size: $\mathcal{O}(n)$
  - Worst-case number of literals: $\mathcal{O}(m\, n)$
  - In practice,

    Unit propagation slow-down worse than linear as clauses are learned **!**
- Clause learning to be effective requires a more efficient representation: **Watched Literals** [MMZZM01]
  - Keep track of only two literals per clause
  - Watched literals are one example of lazy data structures
    - But there are others

Additional key techniques

- Lightweight branching [MMZZM01]
  - Use conflict to bias variables to branch on, associate score with each variable
  - Prefer recent bias by regularly decreasing variable scores
  - Recent promising ML-based branching [LGPC16a,LGPC16b]

Additional key techniques

- Lightweight branching [MMZZM01]
    - Use conflict to bias variables to branch on, associate score with each variable
    - Prefer recent bias by regularly decreasing variable scores
    - Recent promising ML-based branching [LGPC16a,LGPC16b]
- Clause deletion policies
    - Not practical to keep all learned clauses
    - Delete larger clauses [E.g. MSS96a,MSS99]
    - Delete less used clauses [E.g. GN02,ES03]

Additional key techniques

- Lightweight branching [MMZZM01]
    - Use conflict to bias variables to branch on, associate score with each variable
    - Prefer recent bias by regularly decreasing variable scores
    - Recent promising ML-based branching [LGPC16a,LGPC16b]
- Clause deletion policies
    - Not practical to keep all learned clauses
    - Delete larger clauses [E.g. MSS96a,MSS99]
    - Delete less used clauses [E.g. GN02,ES03]
- Other effective techniques:
    - Phase saving [S00,PD07]
    - Luby restarts [H07]
    - Literal blocks distance [AS09]
    - Preprocessing/inprocessing [E.g. JHB12,HJLSB15]

Getting the maximum mileage from CDCL Solvers

Oracle vs Solver SAT Solvers $\neq$ SAT oracle; The performance of solver depends on the formulas

Getting the maximum mileage from CDCL Solvers

Oracle vs Solver SAT Solvers $\neq$ SAT oracle; The performance of solver depends on the formulas

Incremental Solving It is often easier to solve F followed by G if we G can be written as $G = F \wedge H$

- Clause Learning: If $F \rightarrow C$ then $(F \wedge H) \implies C$

Getting the maximum mileage from CDCL Solvers

Oracle vs Solver SAT Solvers $\neq$ SAT oracle; The performance of solver depends on the formulas

Incremental Solving It is often easier to solve F followed by G if we G can be written as $G = F \wedge H$

- Clause Learning: If $F \rightarrow C$ then $(F \wedge H) \implies C$

Beyond CDCL Solver Just CDCL is not sufficient

- Need to handle CNF+XOR formulas
- XORs can be solved by Gaussian elimination
- CryptoMiniSAT: Solver designed to perform CDCL and Gaussian Elimination in tandem

Part IV

Hashing-based Approach for Uniform Distribution

Outline

1. Uniform Constrained Counting
2. Uniform Constrained Sampling

Uniform Constrained Counting

- Given
  - Boolean variables $X_1, X_2, \cdots X_n$
  - Formula F over $X_1, X_2, \cdots X_n$
  - Weight Function W: $\{0, 1\}^n \mapsto \{1\}$
  - $W(F) = |\mathrm{Sol}(F)|$
- ExactCount(F): Compute $|\mathrm{Sol}(F)|$?
  - #P-complete (Valiant 1979)

Uniform Constrained Counting

- Given
  - Boolean variables $X_1, X_2, \cdots X_n$
  - Formula F over $X_1, X_2, \cdots X_n$
  - Weight Function W: $\{0,1\}^n \mapsto \{1\}$
  - $W(F) = |\mathsf{Sol}(F)|$
- ExactCount(F): Compute $|\mathsf{Sol}(F)|$?
  - #P-complete (Valiant 1979)
- ApproxCount(F, ε, δ): Compute C such that

$$\Pr[\frac{|\mathsf{Sol}(F)|}{1+\varepsilon} \leq C \leq |\mathsf{Sol}(F)|(1+\varepsilon)] \geq 1-\delta$$

Counting in Stockholm

How many people in Stockholm like coffee?

- Population of Stockholm = 952K
- Assign every person a unique ($n =$) 21 bit identifier ($2^n = 952$K)

Counting in Stockholm

How many people in Stockholm like coffee?

- Population of Stockholm = 952K
- Assign every person a unique ($n =$) 21 bit identifier ($2^n = 952K$)
- Attempt #1: Pick 50 people and count how many of them like coffee and multiple by 952K/50

Counting in Stockholm

How many people in Stockholm like coffee?

- Population of Stockholm = 952K
- Assign every person a unique ($n =$) 21 bit identifier ($2^n = 952$K)
- Attempt #1: Pick 50 people and count how many of them like coffee and multiple by 952K/50
  - If only 5 people like coffee, it is unlikely that we will find anyone who likes coffee in our sample of 50

Counting in Stockholm

How many people in Stockholm like coffee?

- Population of Stockholm = 952K
- Assign every person a unique ($n =$) 21 bit identifier ($2^n = 952K$)
- Attempt #1: Pick 50 people and count how many of them like coffee and multiple by 952K/50
  - If only 5 people like coffee, it is unlikely that we will find anyone who likes coffee in our sample of 50
- NP Query: Find a person who likes coffee

Counting in Stockholm

How many people in Stockholm like coffee?

- Population of Stockholm = 952K
- Assign every person a unique ($n =$) 21 bit identifier ($2^n = 952K$)
- Attempt #1: Pick 50 people and count how many of them like coffee and multiple by 952K/50
  - If only 5 people like coffee, it is unlikely that we will find anyone who likes coffee in our sample of 50
- NP Query: Find a person who likes coffee
- A SAT solver can answer queries like:
  - Q1: Find a person who likes coffee
  - Q2: Find a person who likes coffee and is not person y

Counting in Stockholm

How many people in Stockholm like coffee?

- Population of Stockholm = 952K
- Assign every person a unique ($n =$) 21 bit identifier ($2^n = 952$K)
- Attempt #1: Pick 50 people and count how many of them like coffee and multiple by 952K/50
  - If only 5 people like coffee, it is unlikely that we will find anyone who likes coffee in our sample of 50
- NP Query: Find a person who likes coffee
- A SAT solver can answer queries like:
  - Q1: Find a person who likes coffee
  - Q2: Find a person who likes coffee and is not person y
- Attempt #2: Enumerate every person who likes coffee

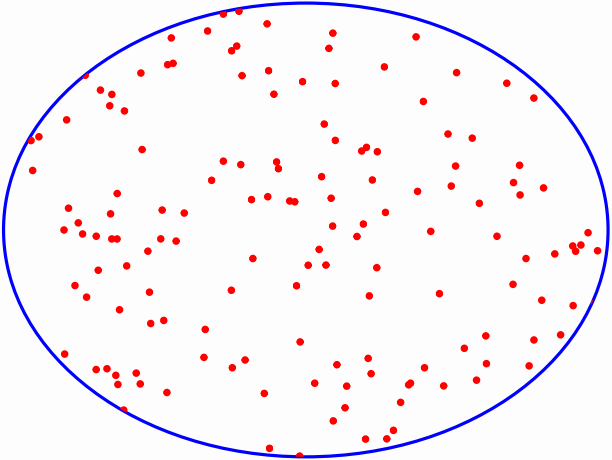
Counting in Stockholm

How many people in Stockholm like coffee?

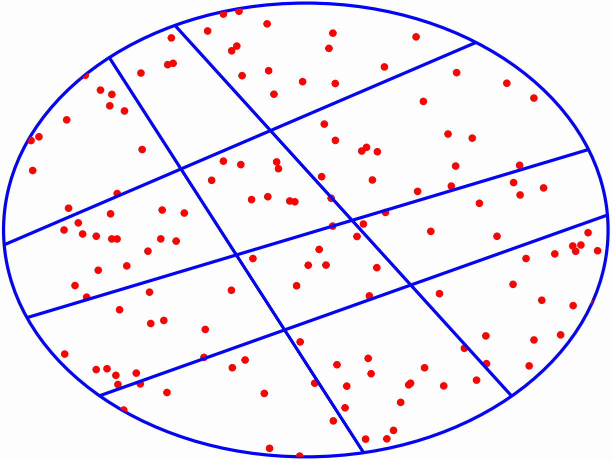
- Population of Stockholm = 952K
- Assign every person a unique ($n =$) 21 bit identifier ($2^n = 952$K)
- Attempt #1: Pick 50 people and count how many of them like coffee and multiple by 952K/50
  - If only 5 people like coffee, it is unlikely that we will find anyone who likes coffee in our sample of 50
- NP Query: Find a person who likes coffee
- A SAT solver can answer queries like:
  - Q1: Find a person who likes coffee
  - Q2: Find a person who likes coffee and is not person y
- Attempt #2: Enumerate every person who likes coffee
  - Potentially 2^n queries

Can we do with lesser # of SAT queries – $\mathcal{O}(n)$ or $\mathcal{O}(\log n)$?

As Simple as Counting Dots

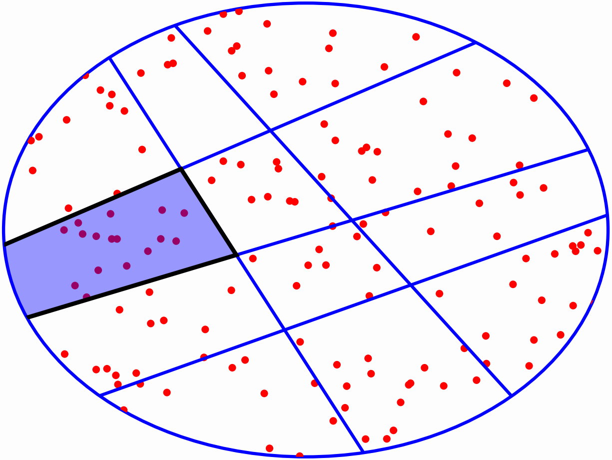


As Simple as Counting Dots



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 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: $\mathsf{Sol}(F) \cap \{y \mid h(y) = \alpha\}$

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: $\mathsf{Sol}(F) \cap \{y \mid h(y) = \alpha\}$
- 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: $\mathsf{Sol}(F) \cap \{y \mid h(y) = \alpha\}$
- Deterministic h unlikely to work
- Choose h randomly from a large family H of hash functions

Universal Hashing (Carter and Wegman 1977)

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 $y \in \mathrm{Sol}(F)$, we define $I_y = \begin{cases} 1 & h(y) = \alpha (\text{y is in cell}) \\ 0 & \text{otherwise} \end{cases}$
- $Z = \sum_{y \in \mathrm{Sol}(F)} I_y$
  - Desired: $\mathrm{E}[Z] = \frac{|\mathrm{Sol}(F)|}{2^m}$ 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 $\mathsf{E}[Z]$ and how much does Z deviate from $\mathsf{E}[Z]$?
- For every $y \in \mathsf{Sol}(F)$, we define $I_y = \begin{cases} 1 & h(y) = \alpha (\text{y is in cell}) \\ 0 & \text{otherwise} \end{cases}$
- $Z = \sum_{y \in \mathsf{Sol}(F)} I_y$
  - Desired: $\mathsf{E}[Z] = \frac{|\mathsf{Sol}(F)|}{2^m}$ and $\sigma^2[Z] \leq \mathsf{E}[Z]$
  - $\Pr\left[\frac{\mathsf{E}[Z]}{2} \leq Z \leq \mathsf{E}[Z] \cdot 2\right] \geq 1 - \frac{4\sigma^2[Z]}{(\mathsf{E}[Z])^2} \geq 1 - \frac{4}{(\mathsf{E}[Z])}$

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 $y \in \mathsf{Sol}(F)$, we define $I_y = \begin{cases} 1 & h(y) = \alpha (\text{y is in cell}) \\ 0 & \text{otherwise} \end{cases}$
- $Z = \sum_{y \in \mathsf{Sol}(F)} I_y$
  - Desired: $\mathsf{E}[Z] = \frac{|\mathsf{Sol}(F)|}{2^m}$ and $\sigma^2[Z] \leq \mathsf{E}[Z]$
  - $\Pr\left[\frac{\mathsf{E}[Z]}{2} \leq Z \leq \mathsf{E}[Z] \cdot 2\right] \geq 1 - \frac{4\sigma^2[Z]}{(\mathsf{E}[Z])^2} \geq 1 - \frac{4}{(\mathsf{E}[Z])}$
  - Having $\mathsf{E}[Z] \geq 4 \cdot k$ provides $1 - \frac{1}{k}$ lower bound
- What kind of H would ensure the above properties

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 $y \in \mathsf{Sol}(F)$, we define $I_y = \begin{cases} 1 & h(y) = \alpha (\text{y is in cell}) \\ 0 & \text{otherwise} \end{cases}$
- $Z = \sum_{y \in \mathsf{Sol}(F)} I_y$
  - Desired: $\mathsf{E}[Z] = \frac{|\mathsf{Sol}(F)|}{2^m}$ and $\sigma^2[Z] \leq \mathsf{E}[Z]$
  - $\Pr\left[\frac{\mathsf{E}[Z]}{2} \leq Z \leq \mathsf{E}[Z] \cdot 2\right] \geq 1 - \frac{4\sigma^2[Z]}{(\mathsf{E}[Z])^2} \geq 1 - \frac{4}{(\mathsf{E}[Z])}$
  - Having $\mathsf{E}[Z] \geq 4 \cdot k$ provides $1 - \frac{1}{k}$ lower bound
- What kind of H would ensure the above properties
- 2-universal hash functions

2-Universal Hashing

- Let H be family of 2-universal hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$

$$\forall y_1, y_2 \in \{0,1\}^n, \alpha_1, \alpha_2 \in \{0,1\}^m, h \overset{R}{\leftarrow} H$$
$$\Pr[h(y_1) = \alpha_1] = \Pr[h(y_2) = \alpha_2] = \left(\frac{1}{2^m}\right)$$

$$\Pr[h(y_1) = \alpha_1 \wedge h(y_2) = \alpha_2] = \left(\frac{1}{2^m}\right)^2$$

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; and XOR 1 with prob. $\frac{1}{2}$
  - $X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2} \oplus 1$
  - 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; and XOR 1 with prob. $\frac{1}{2}$
  - $X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2} \oplus 1$
  - Expected size of each XOR: $\frac{n}{2}$
- To choose $\alpha \in \{0,1\}^m$, set every XOR equation to 0 or 1 randomly

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

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

The explanation for 2-universality

- To construct $h : \{0,1\}^n \rightarrow \{0,1\}^m$, choose m random XORs
- Since every XOR is independently constructed, let us focus on the first XOR (denoted by h^1) and the first bit of the cell: α^1
- We can view construction of h^1 as choosing $a_1, a_2 \ldots a_n, b$ randomly with prob $\frac{1}{2}$ and then writing XOR as $a_1 \cdot x_1 \oplus a_2 \cdot x_2 \oplus \ldots a_n \cdot x_n \oplus b$
- 1-universality, i.e. $\Pr[h^1(y) = \alpha^1]$
  - For every choice of $a_1, a_2, \ldots a_n$, there is a unique b such that $h^1(y) = \alpha^1$. $Pr[h^1(y) = \alpha^1] = \frac{1}{2}$

The explanation for 2-universality

- We can view construction of h^1 as $a_1, a_2 \ldots a_n, b$ randomly with prob $\frac{1}{2}$ and then writing XOR as $a_1 \cdot x_1 \oplus a_2 \cdot x_2 \oplus \ldots a_n \cdot x_n \oplus b$
- 2-universality, i.e., $\Pr[h^1(y) = \alpha^1 \mid h^1(z) = \alpha^1]$
  - $\Pr[h^1(y) = \alpha^1 \mid h^1(z) = \alpha^1] \equiv \Pr[h^1(y - z) = 0]$
  - Let us consider $y - z = [1, 0, 0, \ldots 0]$

The explanation for 2-universality

- We can view construction of h^1 as $a_1, a_2 \dots a_n, b$ randomly with prob $\frac{1}{2}$ and then writing XOR as $a_1 \cdot x_1 \oplus a_2 \cdot x_2 \oplus \dots a_n \cdot x_n \oplus b$
- 2-universality, i.e., $\Pr[h^1(y) = \alpha^1 \mid h^1(z) = \alpha^1]$
  - $\Pr[h^1(y) = \alpha^1 \mid h^1(z) = \alpha^1] \equiv \Pr[h^1(y - z) = 0]$
  - Let us consider $y - z = [1, 0, 0, \dots 0]$
  - $\Pr[h^1([1, 0, 0, \dots 0]) = 0] \equiv \Pr[a_1 = 0] =$

The explanation for 2-universality

- We can view construction of h^1 as $a_1, a_2 \dots a_n, b$ randomly with prob $\frac{1}{2}$ and then writing XOR as $a_1 \cdot x_1 \oplus a_2 \cdot x_2 \oplus \dots a_n \cdot x_n \oplus b$
- 2-universality, i.e., $\Pr[h^1(y) = \alpha^1 \mid h^1(z) = \alpha^1]$
  - $\Pr[h^1(y) = \alpha^1 \mid h^1(z) = \alpha^1] \equiv \Pr[h^1(y - z) = 0]$
  - Let us consider $y - z = [1, 0, 0, \dots 0]$
  - $\Pr[h^1([1, 0, 0, \dots 0]) = 0] \equiv \Pr[a_1 = 0] = \frac{1}{2}$
  - Now observe that set of all possible h^1 is unchanged under rotation and origin shift operation.

The explanation for 2-universality

- We can view construction of h^1 as $a_1, a_2 \dots a_n, b$ randomly with prob $\frac{1}{2}$ and then writing XOR as $a_1 \cdot x_1 \oplus a_2 \cdot x_2 \oplus \dots a_n \cdot x_n \oplus b$
- 2-universality, i.e., $\Pr[h^1(y) = \alpha^1 \mid h^1(z) = \alpha^1]$
  - $\Pr[h^1(y) = \alpha^1 \mid h^1(z) = \alpha^1] \equiv \Pr[h^1(y - z) = 0]$
  - Let us consider $y - z = [1, 0, 0, \dots 0]$
  - $\Pr[h^1([1, 0, 0, \dots 0]) = 0] \equiv \Pr[a_1 = 0] = \frac{1}{2}$
  - Now observe that set of all possible h^1 is unchanged under rotation and origin shift operation. Therefore, for all $y, z \in \{0, 1\}^n$, one can perform series of transformations such that $T(y) - T(z) = [1, 0, 0, \dots 0]$

The explanation for 2-universality

- We can view construction of h^1 as $a_1, a_2 \ldots a_n, b$ randomly with prob $\frac{1}{2}$ and then writing XOR as $a_1 \cdot x_1 \oplus a_2 \cdot x_2 \oplus \ldots a_n \cdot x_n \oplus b$
- 2-universality, i.e., $\Pr[h^1(y) = \alpha^1 \mid h^1(z) = \alpha^1]$
  - $\Pr[h^1(y) = \alpha^1 \mid h^1(z) = \alpha^1] \equiv \Pr[h^1(y - z) = 0]$
  - Let us consider $y - z = [1, 0, 0, \ldots 0]$
  - $\Pr[h^1([1, 0, 0, \ldots 0]) = 0] \equiv \Pr[a_1 = 0] = \frac{1}{2}$
  - Now observe that set of all possible h^1 is unchanged under rotation and origin shift operation. Therefore, for all $y, z \in \{0, 1\}^n$, one can perform series of transformations such that $T(y) - T(z) = [1, 0, 0, \ldots 0]$
  - $\Pr[h^1(y - z) = 0] = \frac{1}{2}$

Challenges

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

- Choose *h* randomly from a large family *H* of hash functions

  $\boxed{\text{Universal Hashing (Carter and Wegman 1977)}}$

Challenge 2 How many cells?

Question 2: How many cells?

- A cell is small if it has less than $\mathrm{thresh} = 48$ solutions

Question 2: How many cells?

- A cell is small if it has less than $\mathrm{thresh} = 48$ solutions
- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$

Question 2: How many cells?

- A cell is small if it has less than $\mathrm{thresh} = 48$ solutions
- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$
    - Check for every $m = 0, 1, \cdots n$ if the number of solutions $\leq \mathrm{thresh}$

HashCount(F, δ)

- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$
  - Query 1: Is $\#(F \wedge Q_1^1) \leq \mathrm{thresh}$
  - Query 2: Is $\#(F \wedge Q_2^1 \wedge Q_2^2 \leq \mathrm{thresh}$
  - $\cdots$
  - Query n: Is $\#(F \wedge Q_3^1 \wedge Q_3^2 \cdots \wedge Q_n^n \leq \mathrm{thresh}$
- Stop at the first m where Query m returns YES and return estimate as $\#(F \wedge Q_m^1 \wedge Q_m^2 \cdots \wedge Q_m^m) \times 2^m$

$\mathsf{HashCount}(F, \delta)$

- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$
  - Query 1: Is $\#(F \wedge Q_1^1) \leq \mathrm{thresh}$
  - Query 2: Is $\#(F \wedge Q_2^1 \wedge Q_2^2 \leq \mathrm{thresh}$
  - $\cdots$
  - Query n: Is $\#(F \wedge Q_3^1 \wedge Q_3^2 \cdots \wedge Q_n^n \leq \mathrm{thresh}$
- Stop at the first m where Query m returns YES and return estimate as $\#(F \wedge Q_m^1 \wedge Q_m^2 \cdots \wedge Q_m^m) \times 2^m$
- To obtain confidence of $1 - \delta$, repeat the above procedure $\mathcal{O}(\log \frac{1}{\delta})$

HashCount(F, δ)

- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$
  - Query 1: Is $\#(F \wedge Q_1^1) \leq \mathrm{thresh}$
  - Query 2: Is $\#(F \wedge Q_2^1 \wedge Q_2^2 \leq \mathrm{thresh}$
  - $\cdots$
  - Query n: Is $\#(F \wedge Q_3^1 \wedge Q_3^2 \cdots \wedge Q_n^n \leq \mathrm{thresh}$
- Stop at the first m where Query m returns YES and return estimate as $\#(F \wedge Q_m^1 \wedge Q_m^2 \cdots \wedge Q_m^m) \times 2^m$
- To obtain confidence of $1-\delta$, repeat the above procedure $\mathcal{O}(\log \frac{1}{\delta})$
- **Will this work? Will the "m" where we stop be close to m^*?**

HashCount

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

Lemma (1)

For (F, ε, δ), the procedure 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 factor of 8 of $|\mathsf{Sol}(F)|$ with probability $\geq$ 0.8

Theorem (Correctness)

$$\Pr\left[\frac{|\mathsf{Sol}(F)|}{8} \leq \mathit{HashCount}(F, \delta) \leq |\mathsf{Sol}(F)|(8)\right] \geq 1 - \delta$$

From Constant Factor to $(1+\varepsilon)$

- $G = F(X) \wedge F(Y)$
- $|\mathrm{Sol}(G)| = |\mathrm{Sol}(F)|^2$
- $\frac{|\mathrm{Sol}(G)|}{8} \leq C \leq 8|\mathrm{Sol}(G)| \implies \frac{|\mathrm{Sol}(G)|}{\sqrt{8}} \leq C \leq \sqrt{8}|\mathrm{Sol}(G)|$

From Constant Factor to $(1 + \varepsilon)$

- $G = F(X) \wedge F(Y)$
- $|\mathsf{Sol}(G)| = |\mathsf{Sol}(F)|^2$
- $\frac{|\mathsf{Sol}(G)|}{8} \leq C \leq 8|\mathsf{Sol}(G)| \implies \frac{|\mathsf{Sol}(G)|}{\sqrt{8}} \leq C \leq \sqrt{8}|\mathsf{Sol}(G)|$
- Make $\mathcal{O}(\frac{1}{\varepsilon})$ copies of F and then take $\frac{1}{\varepsilon}$the root of the estimate to obtain $(1 + \varepsilon)$ factor approximation

HashCount(F, ε, δ)

Theorem (Correctness)

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

Theorem (Complexity)

HashCount(F, ε, δ) *makes* $\mathcal{O}(\frac{\boldsymbol{n}\,\mathbf{log}\,\boldsymbol{n}\log(\frac{1}{\delta})}{\varepsilon})$ *calls to SAT oracle (Stockmeyer 1983)*

Scalability

HashCount fails to scale to formulas beyond few hundreds of variables

Challenges

Long XORs	Expected size of each XOR added is $n/2$
Large Formulas	HashCount is invoked on G, where $\|G\| = \frac{1}{\varepsilon} \times \|F\|$
No Incrementality	The calls to SAT oracle do not allow incremental solving
Too many calls	The number of calls to SAT oracle is $O(n \log n)$

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
  - $X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2}$
  - 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
  - $X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2}$
  - Expected size of each XOR: $\frac{n}{2}$
- To choose $\alpha \in \{0,1\}^m$, set every XOR equation to 0 or 1 randomly

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

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

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
  - $X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2}$
  - Expected size of each XOR: $\frac{n}{2}$
- To choose $\alpha \in \{0,1\}^m$, set every XOR equation to 0 or 1 randomly

$$X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{n-2} = 0 \quad (Q_1)$$

$$X_2 \oplus X_5 \oplus X_6 \cdots \oplus X_{n-1} = 1 \quad (Q_2)$$

$$\cdots \quad (\cdots)$$

$$X_1 \oplus X_2 \oplus X_5 \cdots \oplus X_{n-2} = 1 \quad (Q_m)$$

- Solutions in a cell: $F \wedge Q_1 \cdots \wedge Q_m$
- The performance of SAT solver degrades with increase in size of XORs (SAT solver $\neq$ SAT oracle)

Improved Universal Hash Functions

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

Improved Universal Hash Functions

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

Improved Universal Hash Functions

- Not all variables are required to specify solution space of F
  - $F := X_3 \iff (X_1 \vee X_2)$
  - X_1 and X_2 uniquely determines rest of the variables (i.e., X_3)
- Formally: if I is independent support, then $\forall \sigma_1, \sigma_2 \in \mathsf{Sol}(F)$, 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
- 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:
  $\forall \sigma_1, \sigma_2 \in \mathsf{Sol}(()\varphi)$, σ_1 and σ_2 agree on I then $\sigma_1 = \sigma_2$
- $F(x_1, \cdots x_n) \wedge F(y_1, \cdots y_n) \wedge \bigwedge_{i \mid x_i \in I}(x_i = y_i) \implies \bigwedge_i (x_i = y_i)$
  where $F(y_1, \cdots y_n) := F(x_1 \rightarrowtail y_1, \cdots x_n \rightarrowtail y_n)$
- $Q_{F,I} := F(x_1, \cdots x_n) \wedge F(y_1, \cdots y_n) \wedge \bigwedge_{i \mid x_i \in I}(x_i = y_i) \wedge \neg(\bigwedge_i (x_i = y_i))$

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 many cells?

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 $y \in \mathrm{Sol}(F)$, we define $I_y = \begin{cases} 1 & h(y) = \alpha(\text{y is in cell}) \\ 0 & \text{otherwise} \end{cases}$
- $Z = \sum_{y \in \mathrm{Sol}(F)} I_y$
  - Desired: $\mathrm{E}[Z] = \frac{|\mathrm{Sol}(F)|}{2^m}$ 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 $\mathsf{E}[Z]$ and how much does Z deviate from $\mathsf{E}[Z]$?
- For every $y \in \mathsf{Sol}(F)$, we define $I_y = \begin{cases} 1 & h(y) = \alpha(\text{y is in cell}) \\ 0 & \text{otherwise} \end{cases}$
- $Z = \sum_{y \in \mathsf{Sol}(F)} I_y$
  - Desired: $\mathsf{E}[Z] = \frac{|\mathsf{Sol}(F)|}{2^m}$ and $\sigma^2[Z] \leq \mathsf{E}[Z]$
  - $\Pr\left[\frac{\mathsf{E}[Z]}{1+\varepsilon} \leq Z \leq \mathsf{E}[Z](1+\varepsilon)\right] \geq 1 - \frac{\sigma^2[Z]}{(\frac{\varepsilon}{1+\varepsilon})^2(\mathsf{E}[Z])^2} \geq 1 - \frac{1}{(\frac{\varepsilon}{1+\varepsilon})^2(\mathsf{E}[Z])}$

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 $y \in \mathsf{Sol}(F)$, we define $I_y = \begin{cases} 1 & h(y) = \alpha (\text{y is in cell}) \\ 0 & \text{otherwise} \end{cases}$
- $Z = \sum_{y \in \mathsf{Sol}(F)} I_y$
  - Desired: $\mathsf{E}[Z] = \frac{|\mathsf{Sol}(F)|}{2^m}$ and $\sigma^2[Z] \leq \mathsf{E}[Z]$
  - $\Pr\left[\frac{\mathsf{E}[Z]}{1+\varepsilon} \leq Z \leq \mathsf{E}[Z](1+\varepsilon)\right] \geq 1 - \frac{\sigma^2[Z]}{(\frac{\varepsilon}{1+\varepsilon})^2(\mathsf{E}[Z])^2} \geq 1 - \frac{1}{(\frac{\varepsilon}{1+\varepsilon})^2(\mathsf{E}[Z])}$
  - Having $\mathsf{E}[Z] \geq k(\frac{1+\varepsilon}{\varepsilon})^2$ provides $1 - \frac{1}{k}$ lower bound

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 $y \in \mathsf{Sol}(F)$, we define $I_y = \begin{cases} 1 & h(y) = \alpha (\text{y is in cell}) \\ 0 & \text{otherwise} \end{cases}$
- $Z = \sum_{y \in \mathsf{Sol}(F)} I_y$
  - Desired: $\mathsf{E}[Z] = \frac{|\mathsf{Sol}(F)|}{2^m}$ and $\sigma^2[Z] \leq \mathsf{E}[Z]$
  - $\Pr\left[\frac{\mathsf{E}[Z]}{1+\varepsilon} \leq Z \leq \mathsf{E}[Z](1+\varepsilon)\right] \geq 1 - \frac{\sigma^2[Z]}{(\frac{\varepsilon}{1+\varepsilon})^2(\mathsf{E}[Z])^2} \geq 1 - \frac{1}{(\frac{\varepsilon}{1+\varepsilon})^2(\mathsf{E}[Z])}$
  - Having $\mathsf{E}[Z] \geq k(\frac{1+\varepsilon}{\varepsilon})^2$ provides $1 - \frac{1}{k}$ lower bound
- A cell is small if it has less than $\mathrm{thresh} = 5(\frac{1+\varepsilon}{\varepsilon})^2$ solutions

Question 2: How many cells?

- A cell is small if it has less than $\mathrm{thresh} = 5(1 + \frac{1}{\varepsilon})^2$ solutions

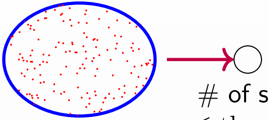
Question 2: How many cells?

- 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 $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$

Question 2: How many cells?

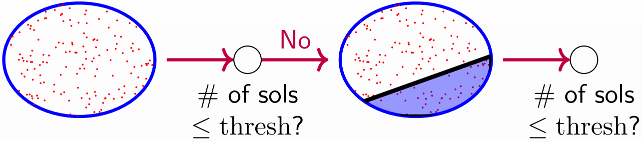
- A cell is small if it has less than $\mathrm{thresh} = 5(1 + \frac{1}{\varepsilon})^2$ solutions
- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$
  - Check for every $m = 0, 1, \cdots n$ if the number of solutions $\leq \mathrm{thresh}$

ApproxMC(F, ε, δ)

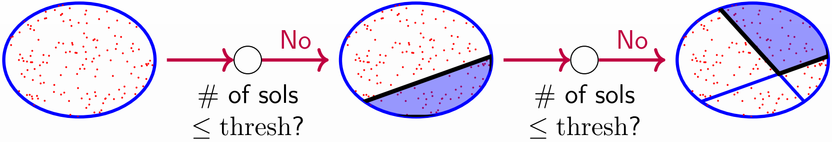


# of sols
$\leq$ thresh?

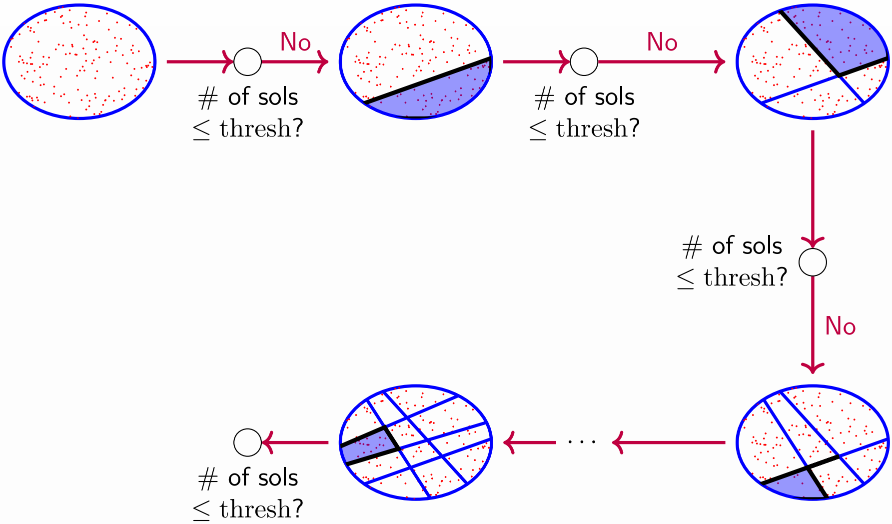
ApproxMC(F, ε, δ)



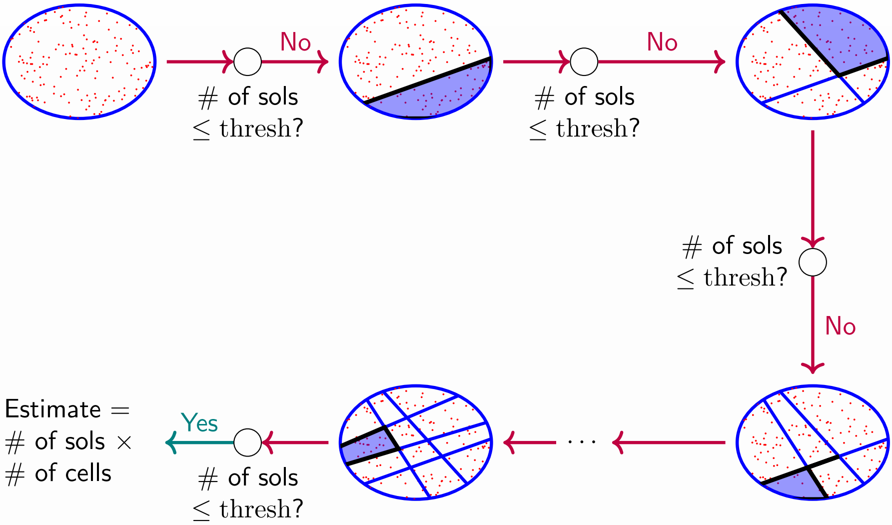
ApproxMC(F, ε, δ)



ApproxMC(F, ε, δ)



ApproxMC(F, ε, δ)



Estimate =
# of sols ×
# of cells

ApproxMC(F, ε, δ)

- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$
  - Query 1: Is $\#(F \wedge Q_1) \leq \mathrm{thresh}$
  - Query 2: Is $\#(F \wedge Q_1 \wedge Q_2) \leq \mathrm{thresh}$
  - $\cdots$
  - Query n: Is $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_n) \leq \mathrm{thresh}$
- 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

ApproxMC(F, ε, δ)

- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$
  - Query 1: Is $\#(F \wedge Q_1) \leq \mathrm{thresh}$
  - Query 2: Is $\#(F \wedge Q_1 \wedge Q_2) \leq \mathrm{thresh}$
  - $\cdots$
  - Query n: Is $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_n) \leq \mathrm{thresh}$
- 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
  - Logarithmic search (# of SAT calls: $\mathcal{O}(\log n)$)
  - Incremental search

ApproxMC(F, ε, δ)

- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$
  - Query 1: Is $\#(F \wedge Q_1) \leq \mathrm{thresh}$
  - Query 2: Is $\#(F \wedge Q_1 \wedge Q_2) \leq \mathrm{thresh}$
  - $\cdots$
  - Query n: Is $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_n) \leq \mathrm{thresh}$
- 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
  - Logarithmic search (# of SAT calls: $\mathcal{O}(\log n)$)
  - Incremental search
- **Will this work? Will the "m" where we stop be close to m^*?**

ApproxMC(F, ε, δ)

- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$
  - Query 1: Is $\#(F \wedge Q_1) \leq \mathrm{thresh}$
  - Query 2: Is $\#(F \wedge Q_1 \wedge Q_2) \leq \mathrm{thresh}$
  - $\cdots$
  - Query n: Is $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_n) \leq \mathrm{thresh}$
- 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
  - Logarithmic search (# of SAT calls: $\mathcal{O}(\log n)$)
  - Incremental search
- **Will this work? Will the "m" where we stop be close to m^*?**
  - Challenge Query i and Query j are not independent
  - Independence crucial to analysis (Stockmeyer 1983, $\cdots$)

ApproxMC(F, ε, δ)

- We want to partition into 2^{m^*} cells such that $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$
  - Query 1: Is $\#(F \wedge Q_1) \leq \mathrm{thresh}$
  - Query 2: Is $\#(F \wedge Q_1 \wedge Q_2) \leq \mathrm{thresh}$
  - $\cdots$
  - Query n: Is $\#(F \wedge Q_1 \wedge Q_2 \cdots \wedge Q_n) \leq \mathrm{thresh}$
- 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
  - Logarithmic search (# of SAT calls: $\mathcal{O}(\log n)$)
  - Incremental search
- **Will this work? Will the "m" where we stop be close to m^*?**
  - Challenge Query i and Query j are not independent
  - Independence crucial to analysis (Stockmeyer 1983, $\cdots$)
  - Key Insight: The probability of making a bad choice of Q_i is very small for $i \ll m^*$

Taming the Curse of Dependence

Let $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$ ($m^* = \log(\frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}})$)

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* $|\mathsf{Sol}(F)|$ *with probability* $\geq$ *0.8*

ApproxMC(F, ε, δ)

Theorem (Correctness)

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

Theorem (Complexity)

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

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

Summary of our Journey

HashCount fails to scale to formulas beyond few hundreds of variables

Challenges

Long XORs	Expected size of each XOR added is $n/2$
Large Formulas	HashCount is invoked on G, where $\|G\| = \frac{1}{\varepsilon} \times \|F\|$
No Incrementality	The calls to SAT oracle do not allow incremental solving
Too many calls	The number of calls to SAT oracle is $O(n \log n)$

Summary of our Journey

HashCount fails to scale to formulas beyond few hundreds of variables

Challenges

Long XORs Expected size of each XOR added is $n/2$
Independent support-based XORs

Large Formulas HashCount is invoked on G, where $|G| = \frac{1}{\varepsilon} \times |F|$

No Incrementality The calls to SAT oracle do not allow incremental solving

Too many calls The number of calls to SAT oracle is $O(n \log n)$

Summary of our Journey

HashCount fails to scale to formulas beyond few hundreds of variables

Challenges

Long XORs Expected size of each XOR added is $n/2$
Independent support-based XORs

Large Formulas HashCount is invoked on G, where $|G| = \frac{1}{\varepsilon} \times |F|$
Constant pivot to ε dependent pivot

No Incrementality The calls to SAT oracle do not allow incremental solving

Too many calls The number of calls to SAT oracle is $O(n \log n)$
Dependent XORs with new proof technique. Killed two birds with one stone!

Reliability of Critical Infrastructure Networks

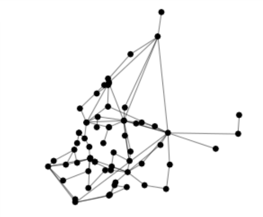
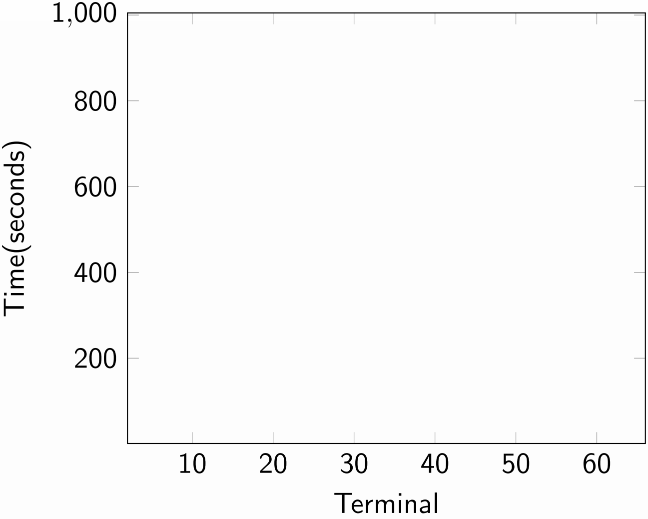


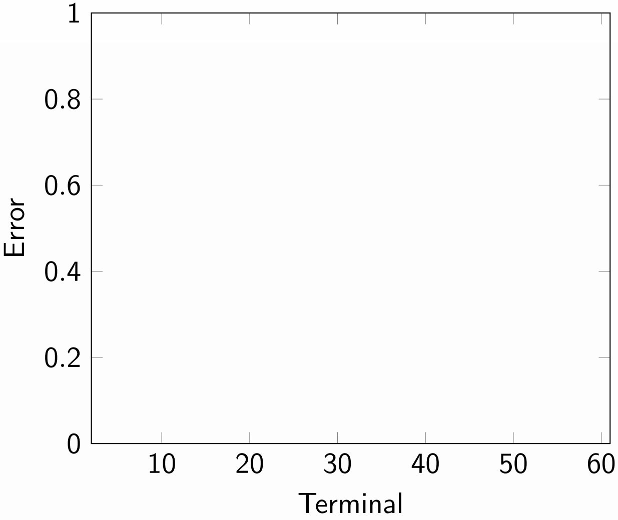
Figure: Plantersville, SC

- $G = (V, E)$; source node: s
- Compute Pr[t is disconnected]?

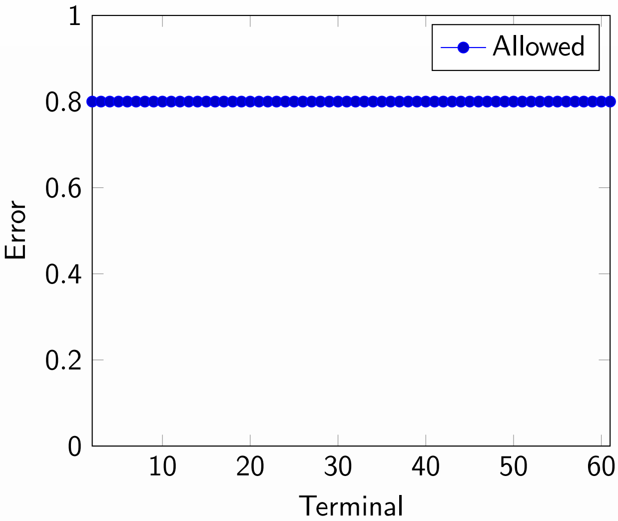
Timeout = 1000 seconds



Highly Accurate Estimates

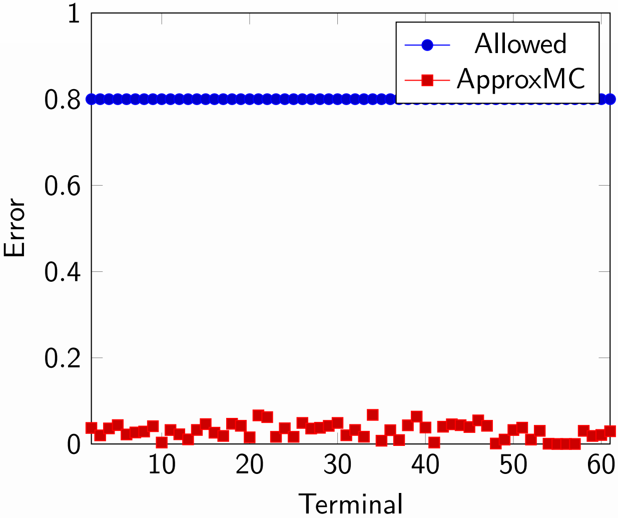


Highly Accurate Estimates



Observed Geometric mean: 0.03

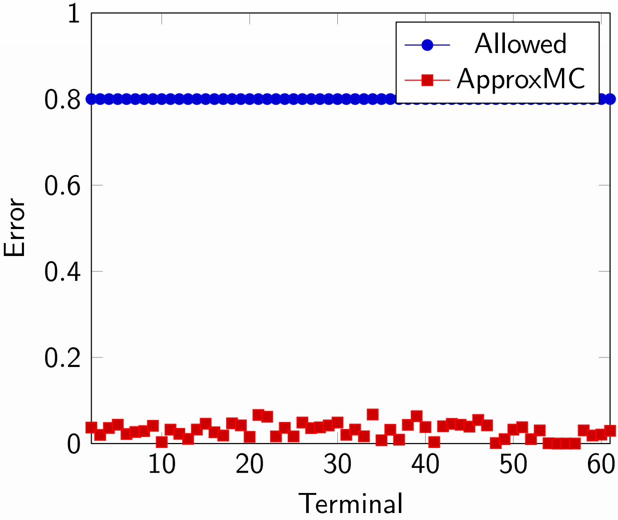
Highly Accurate Estimates



Observed Geometric mean: 0.03

These results are good

Highly Accurate Estimates



Observed Geometric mean: 0.03

These results are good problem.

Outline

1. Uniform Constrained Counting
2. Uniform Constrained Sampling

Constrained Sampling

- Given:
  - Set of Constraints F over variables $X_1, X_2, \cdots X_n$
- Uniform Sampler

$$\forall y \in \mathsf{Sol}(F), \mathsf{Pr}[\text{y is output}] = \frac{1}{|\mathsf{Sol}(F)|}$$

- Almost-Uniform Sampler

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

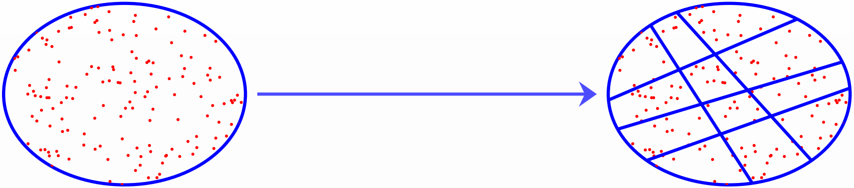
Close Cousins: Counting and Sampling

- Approximate counting and almost-uniform sampling are inter-reducible (Jerrum, Valiant and Vazirani, 1986)

Close Cousins: Counting and Sampling

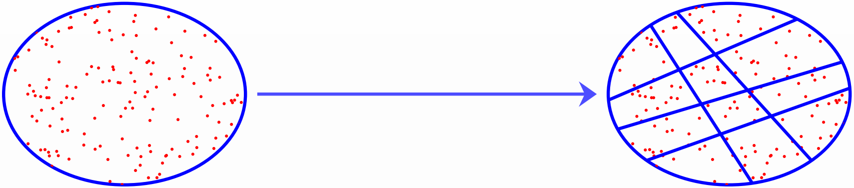
- Approximate counting and almost-uniform sampling are inter-reducible (Jerrum, Valiant and Vazirani, 1986)
- Is the reduction efficient?
  - Almost-uniform sampler (JVV) require linear number of approximate counting calls

Key Ideas



- Check if a randomly picked cell is *small*
  - If yes, pick a solution randomly from randomly picked cell

Key Ideas



- Check if a randomly picked cell is *small*
  - If yes, pick a solution randomly from randomly picked cell

Challenge: How many cells?

How many cells?

- Desired Number of cells: $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\text{thresh}}$ ($m^* = \log \frac{|\mathsf{Sol}(F)|}{\text{thresh}}$)

How many cells?

- Desired Number of cells: $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\text{thresh}}$ ($m^* = \log \frac{|\mathsf{Sol}(F)|}{\text{thresh}}$)
  - ApproxMC(F, ε, δ) returns C such that
    $$\Pr\left[\frac{|\mathsf{Sol}(F)|}{1+\varepsilon} \leq C \leq |\mathsf{Sol}(F)|(1+\varepsilon)\right] \geq 1 - \delta$$
  - $\tilde{m} = \log \frac{C}{\text{thresh}}$

How many cells?

- Desired Number of cells: $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$ ($m^* = \log \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$)
  - ApproxMC(F, ε, δ) returns C such that
$$\Pr\left[\frac{|\mathsf{Sol}(F)|}{1+\varepsilon} \leq C \leq |\mathsf{Sol}(F)|(1+\varepsilon)\right] \geq 1-\delta$$
  - $\tilde{m} = \log \frac{C}{\mathrm{thresh}}$
  - 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: $2^{m^*} = \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$ ($m^* = \log \frac{|\mathsf{Sol}(F)|}{\mathrm{thresh}}$)
  - ApproxMC(F, ε, δ) returns C such that
    $$\Pr\left[\frac{|\mathsf{Sol}(F)|}{1+\varepsilon} \leq C \leq |\mathsf{Sol}(F)|(1+\varepsilon)\right] \geq 1 - \delta$$
  - $\tilde{m} = \log \frac{C}{\mathrm{thresh}}$
  - 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

Theoretical Guarantees

Theorem (Almost-Uniformity)

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

Theoretical Guarantees

Theorem (Almost-Uniformity)

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

Theorem (Query)

For a formula F over n variables UniGen makes **one call** *to approximate counter*

Theoretical Guarantees

Theorem (Almost-Uniformity)

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

Theorem (Query)

For a formula F over n variables UniGen makes **one call** *to approximate counter*

- *Prior work required* **n** *calls to approximate counter* (*Jerrum, Valiant and Vazirani, 1986*)

Theoretical Guarantees

Theorem (Almost-Uniformity)

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

Theorem (Query)

For a formula F over n variables UniGen makes **one call** *to approximate counter*

- *Prior work required* **n** *calls to approximate counter (Jerrum, Valiant and Vazirani, 1986)*

Universality

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

Theoretical Guarantees

Theorem (Almost-Uniformity)

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

Theorem (Query)

For a formula F over n variables UniGen makes **one call** *to approximate counter*

- *Prior work required* **n** *calls to approximate counter (Jerrum, Valiant and Vazirani, 1986)*

Universality

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

Random XORs are 3-universal

	Relative Runtime
SAT Solver	1
Desired Uniform Generator	10

Experiments over 200+ benchmarks

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

Experiments over 200+ benchmarks

Three Orders of Improvement

	Relative Runtime
SAT Solver	1
Desired Uniform Generator	10
XORSample (2012 state of the art)	50000
UniGen (2015)	21

Experiments over 200+ benchmarks

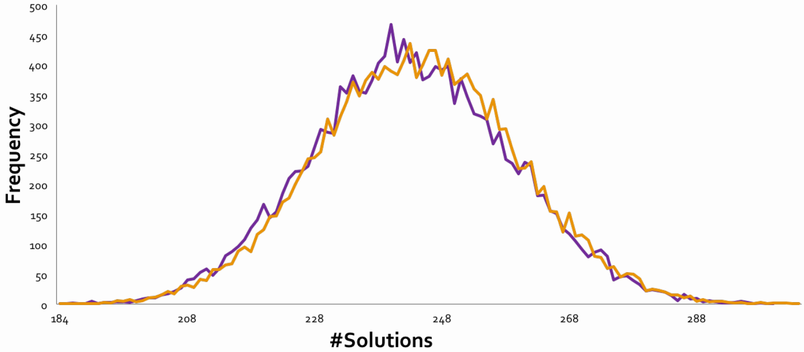
Three Orders of Improvement

	Relative Runtime
SAT Solver	1
Desired Uniform Generator	10
XORSample (2012 state of the art)	50000
UniGen (2015)	21

Experiments over 200+ benchmarks

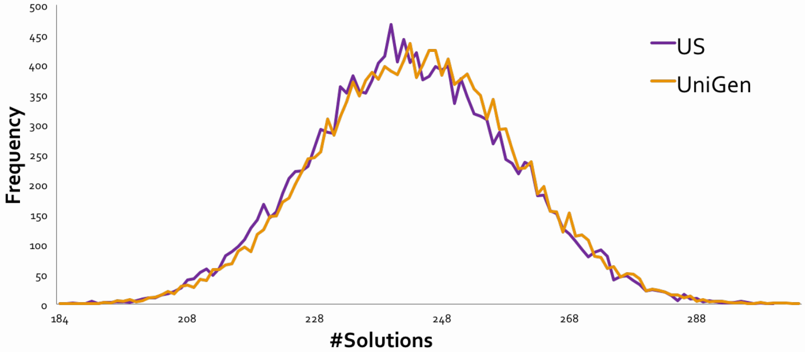
Closer to technical transfer

Quiz Time: 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

Outline

Part 1:	Applications
Part 2:	Prior Work
Part 3:	Overview of SAT Solving
Part 4:	Hashing-based Approach for Uniform Distribution
Part 5:	Beyond Propositional
Part 6:	Challenges

Part V

Beyond Propositional

Why go beyond propositional?

Why go beyond propositional?

- Lifted inference: first order (FO) logic + probabilistic reasoning (Kersting2012, Poole2003)
    - FO variables of non-binary type
    - Reasoning about FO constraints directly key to scalability
    - Inference reduces to counting models of these constraints

Why go beyond propositional?

- Lifted inference: first order (FO) logic + probabilistic reasoning (Kersting2012, Poole2003)
  - FO variables of non-binary type
  - Reasoning about FO constraints directly key to scalability
  - Inference reduces to counting models of these constraints
- Probabilistic program analysis
  - Value problem: Pr$[\text{Accepting runs}]$/Pr$[\text{Terminating runs}]$
  - Program variables of enumerated, integer or float type
  - Encoded as model-counting of integer+rational arithmetic formulas (Chistikov2015)

Why go beyond propositional?

- Lifted inference: first order (FO) logic + probabilistic reasoning (Kersting2012, Poole2003)
  - FO variables of non-binary type
  - Reasoning about FO constraints directly key to scalability
  - Inference reduces to counting models of these constraints
- Probabilistic program analysis
  - Value problem: $\Pr[\text{Accepting runs}]/\Pr[\text{Terminating runs}]$
  - Program variables of enumerated, integer or float type
  - Encoded as model-counting of integer+rational arithmetic formulas (Chistikov2015)
- Inference in continuous & hybrid Markov networks
  - Mix of discrete and continuous random variables
  - Encoded as model counting in theory of rationals + Booleans

How do we go beyond propositional?

- For finite domains, binary encoding + propositional counting often used
  - \+ Leverage advances in propositional model counting
  - \- Fails to exploit domain-specific propeties (e.g. linear algebraic identities)
  - \- Scalability a concern
    - ▸ Count of variables and constraints increases with domain size
  - \- Infinite domains out of reach

How do we go beyond propositional?

- For finite domains, binary encoding + propositional counting often used
  - \+ Leverage advances in propositional model counting
  - \- Fails to exploit domain-specific propeties (e.g. linear algebraic identities)
  - \- Scalability a concern
    - Count of variables and constraints increases with domain size
  - \- Infinite domains out of reach
- **Can we do better?**

How do we go beyond propositional?

- For finite domains, binary encoding + propositional counting often used
    + Leverage advances in propositional model counting
    - Fails to exploit domain-specific propeties (e.g. linear algebraic identities)
    - Scalability a concern
        - Count of variables and constraints increases with domain size
    - Infinite domains out of reach
- **Can we do better?**
    - Yes in some cases
    - Not yet in general

Overview: Three different approaches

Overview: Three different approaches

- Domain-specific universal hash functions
    - Not always easy to design
    - Bit-vector model counting

Overview: Three different approaches

- Domain-specific universal hash functions
  - Not always easy to design
  - Bit-vector model counting
- Domain-specific decomposition + prop model counting
  - Estimating model volume in bounded integer+rational linear arithmetic (Chistikov2015)

Overview: Three different approaches

- Domain-specific universal hash functions
  - Not always easy to design
  - Bit-vector model counting
- Domain-specific decomposition + prop model counting
  - Estimating model volume in bounded integer+rational linear arithmetic (Chistikov2015)
- Weighted model integration
  - Generalizes weighted model counting
  - Bootstraps on advances in SMT solvers & abstraction techniques (Belle2015, Morettin2017)

Bit-vector Model Counting

- Given constraint $\varphi(x_1, \dots x_n)$, where

Bit-vector Model Counting

- Given constraint $\varphi(x_1, \dots x_n)$, where
  - $x_1, \dots x_n$ are bit-vector variables
    - Simplifying assumption: all k-bits wide
    - Domain of $x_i = \{0, 1\}^k$

Bit-vector Model Counting

- Given constraint $\varphi(x_1, \ldots x_n)$, where
  - $x_1, \ldots x_n$ are bit-vector variables
    - Simplifying assumption: all k-bits wide
    - Domain of $x_i = \{0, 1\}^k$
  - Functions and predicates from theory of bit-vectors
    - extract, concat, leftshift, $+_{[k]}$, $\times_{k}$...

Bit-vector Model Counting

- Given constraint $\varphi(x_1, \ldots x_n)$, where
  - $x_1, \ldots x_n$ are bit-vector variables
    - Simplifying assumption: all k-bits wide
    - Domain of $x_i = \{0,1\}^k$
  - Functions and predicates from theory of bit-vectors
    - extract, concat, leftshift, $+_{[k]}$, $\times_{k}$...
- Example:
  - $\varphi(x_1, x_2) \equiv (x +_{[3]} y = 000) \vee (\text{extract}(x, 1, 1) = 0)$
  - x_1, x_2: all 3-bits wide

Bit-vector Model Counting

- Given constraint $\varphi(x_1, \ldots x_n)$, where
  - $x_1, \ldots x_n$ are bit-vector variables
    - Simplifying assumption: all k-bits wide
    - Domain of $x_i = \{0,1\}^k$
  - Functions and predicates from theory of bit-vectors
    - extract, concat, leftshift, $+_{[k]}$, $\times_{k}$...
- Example:
  - $\varphi(x_1, x_2) \equiv (x +_{[3]} y = 000) \vee (\text{extract}(x, 1, 1) = 0)$
  - x_1, x_2: all 3-bits wide
  - How many satisfying assignments does φ have?

Bit-vector Model Counting

- Given constraint $\varphi(x_1, \dots x_n)$, where
  - $x_1, \dots x_n$ are bit-vector variables
    - Simplifying assumption: all k-bits wide
    - Domain of $x_i = \{0, 1\}^k$
  - Functions and predicates from theory of bit-vectors
    - extract, concat, leftshift, $+_{[k]}$, $\times_{k} \dots$
- Example:
  - $\varphi(x_1, x_2) \equiv (x +_{[3]} y = 000) \vee (\mathrm{extract}(x, 1, 1) = 0)$
  - x_1, x_2: all 3-bits wide
  - How many satisfying assignments does φ have?
  - $\mathrm{Sol}(\varphi) = \{(x_1 = 000, x_2 = 000), (x_1 = 001, x_2 = 111)\}$
  - $|\mathrm{Sol}(\varphi)| = 2$

Bit-vector Model Counting

- Key idea: New 2-universal hash function h_{BV} for bit-vectors

Bit-vector Model Counting

- Key idea: New 2-universal hash function h_{BV} for bit-vectors
- Recall from propositional case
  - Prop variables $\{x_1, \ldots x_n\}$
  - Example: $h(x_1, ...) = x_1 \oplus x_4 \oplus \ldots \oplus 1$

Bit-vector Model Counting

- Key idea: New 2-universal hash function h_{BV} for bit-vectors
- Recall from propositional case
  - Prop variables $\{x_1, \ldots x_n\}$
  - Example: $h(x_1, ...) = x_1 \oplus x_4 \oplus \ldots \oplus 1$
  - Alternatively, $h(x_1, ...) = (1 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 + 1 \cdot x_4 + \ldots + 1)$ mod 2

Bit-vector Model Counting

- Key idea: New 2-universal hash function h_{BV} for bit-vectors
- Recall from propositional case
    - Prop variables $\{x_1, \ldots x_n\}$
    - Example: $h(x_1, ...) = x_1 \oplus x_4 \oplus \ldots \oplus 1$
    - Alternatively, $h(x_1, ...) = (1 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 + 1 \cdot x_4 + \ldots + 1)$ mod 2
    - Family of hash functions to choose from $\mathcal{H} = \{(a_1 \cdot x_1 + \ldots + a_n \cdot x_n + b) \mod 2 \mid a_1, \ldots a_n, b$ randomly chosen from $\mathbb{Z}_2 = \{0, 1\}\}$

Bit-vector Model Counting

- Key idea: New 2-universal hash function h_{BV} for bit-vectors
- Recall from propositional case
  - Prop variables $\{x_1, \ldots x_n\}$
  - Example: $h(x_1, ...) = x_1 \oplus x_4 \oplus \ldots \oplus 1$
  - Alternatively, $h(x_1, ...) = (1 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 + 1 \cdot x_4 + \ldots + 1)$ mod 2
  - Family of hash functions to choose from $\mathcal{H} = \{(a_1 \cdot x_1 + \ldots + a_n \cdot x_n + b) \mod 2 \mid a_1, \ldots a_n, b$ randomly chosen from $\mathbb{Z}_2 = \{0, 1\}\}$
- Generalizing to bit-vectors
  - Bit-vector variables $\{x_1, \ldots x_n\}$

Bit-vector Model Counting

- Key idea: New 2-universal hash function h_{BV} for bit-vectors
- Recall from propositional case
  - Prop variables $\{x_1, \ldots x_n\}$
  - Example: $h(x_1, ...) = x_1 \oplus x_4 \oplus \ldots \oplus 1$
  - Alternatively, $h(x_1, ...) = (1 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 + 1 \cdot x_4 + \ldots + 1) \mod 2$
  - Family of hash functions to choose from $\mathcal{H} = \{(a_1 \cdot x_1 + \ldots + a_n \cdot x_n + b) \mod 2 \mid a_1, \ldots a_n, b$ randomly chosen from $\mathbb{Z}_2 = \{0, 1\}\}$
- Generalizing to bit-vectors
  - Bit-vector variables $\{x_1, \ldots x_n\}$
  - Use a suitable *prime* p instead of 2 for modulus
    - Smallest p such that $2^k \leq p < 2^{nk}$

Bit-vector Model Counting

- Key idea: New 2-universal hash function h_{BV} for bit-vectors
- Recall from propositional case
  - Prop variables $\{x_1, \ldots x_n\}$
  - Example: $h(x_1, ...) = x_1 \oplus x_4 \oplus \ldots \oplus 1$
  - Alternatively, $h(x_1, ...) = (1 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 + 1 \cdot x_4 + \ldots + 1)$ mod 2
  - Family of hash functions to choose from $\mathcal{H} = \{(a_1 \cdot x_1 + \ldots + a_n \cdot x_n + b) \mod 2 \mid a_1, \ldots a_n, b$ randomly chosen from $\mathbb{Z}_2 = \{0, 1\}\}$
- Generalizing to bit-vectors
  - Bit-vector variables $\{x_1, \ldots x_n\}$
  - Use a suitable *prime* p instead of 2 for modulus
    - Smallest p such that $2^k \leq p < 2^{nk}$
  - First-cut $\mathcal{H}_{BV}$ (linear modular hash functions): $\{(a_1 \cdot x_1 + \ldots + a_n \cdot x_n \ldots b) \mod p \mid a_i, \ldots a_n, b$ randomly chosen from $\mathbb{Z}_p = \{0, 1, \ldots p - 1\}\}$

Closer look at $\mathcal{H}_{BV}$

$\varphi(x_1, \ldots x_n)$: Bit-vector formula

- Randomly choose $h(x_1, \ldots) : \{0,1\}^{nk} \rightarrow \mathbb{Z}_p$ from $\mathcal{H}_{BV}$
  - Partitions $\{0,1\}^{nk}$ into p cells
  - Expected # solutions per cell $= |\mathrm{Sol}(\varphi)|/p$

Closer look at $\mathcal{H}_{BV}$

$\varphi(x_1, \ldots x_n)$: Bit-vector formula

- Randomly choose $h(x_1, ...) : \{0,1\}^{nk} \rightarrow \mathbb{Z}_p$ from $\mathcal{H}_{BV}$
  - Partitions $\{0,1\}^{nk}$ into p cells
  - Expected # solutions per cell = $|\mathsf{Sol}(\varphi)|/p$
- What if p is too small compared to $|\mathsf{Sol}(\varphi)|$?
  - Recall we'd like each cell to have "few" solutions of φ

Closer look at $\mathcal{H}_{BV}$

$\varphi(x_1, \ldots x_n)$: Bit-vector formula

- Randomly choose $h(x_1, \ldots) : \{0,1\}^{nk} \rightarrow \mathbb{Z}_p$ from $\mathcal{H}_{BV}$
  - Partitions $\{0,1\}^{nk}$ into p cells
  - Expected # solutions per cell $= |\mathrm{Sol}(\varphi)|/p$
- What if p is too small compared to $|\mathrm{Sol}(\varphi)|$?
  - Recall we'd like each cell to have "few" solutions of φ
  - Choose $h_1, h_2, \ldots h_c$ independently at random from $\mathcal{H}_{BV}$

Closer look at $\mathcal{H}_{BV}$

$\varphi(x_1, \ldots x_n)$: Bit-vector formula

- Randomly choose $h(x_1, \ldots) : \{0,1\}^{nk} \rightarrow \mathbb{Z}_p$ from $\mathcal{H}_{BV}$
  - Partitions $\{0,1\}^{nk}$ into p cells
  - Expected # solutions per cell $= |\mathsf{Sol}(\varphi)|/p$
- What if p is too small compared to $|\mathsf{Sol}(\varphi)|$?
  - Recall we'd like each cell to have "few" solutions of φ
  - Choose $h_1, h_2, \ldots h_c$ independently at random from $\mathcal{H}_{BV}$
  - Choose $\alpha_1, \ldots \alpha_c$ independently at random from $\mathbb{Z}_p$

Closer look at $\mathcal{H}_{BV}$

$\varphi(x_1, \ldots x_n)$: Bit-vector formula

- Randomly choose $h(x_1, \ldots) : \{0,1\}^{nk} \rightarrow \mathbb{Z}_p$ from $\mathcal{H}_{BV}$
  - Partitions $\{0,1\}^{nk}$ into p cells
  - Expected # solutions per cell = $|\mathsf{Sol}(\varphi)|/p$
- What if p is too small compared to $|\mathsf{Sol}(\varphi)|$?
  - Recall we'd like each cell to have "few" solutions of φ
  - Choose $h_1, h_2, \ldots h_c$ independently at random from $\mathcal{H}_{BV}$
  - Choose $\alpha_1, \ldots \alpha_c$ independently at random from $\mathbb{Z}_p$
  - Expected # models of $\varphi_{BV}(\ldots) \wedge (h_1(\ldots) = \alpha_1) \wedge \cdots (h_c(\ldots) = \alpha_c)$ is $|\mathsf{Sol}(\varphi)|/p^c$
- Works if p^c is within a small factor of $|\mathsf{Sol}(\varphi)|$.

Closer look at $\mathcal{H}_{BV}$

- What if $\mathrm{Sol}(\varphi)/p^c$ is < 1, but $\mathrm{Sol}(\varphi_{BV})/p^{c-1}$ is too large?
  - Can happen for large p
  - Can we reduce p arbitrarily?

Closer look at $\mathcal{H}_{BV}$

- What if $\mathsf{Sol}(\varphi)/p^c$ is < 1, but $\mathsf{Sol}(\varphi_{BV})/p^{c-1}$ is too large?
  - Can happen for large p
  - Can we reduce p arbitrarily? Need $2^k \leq p < 2^{nk}$

Closer look at $\mathcal{H}_{BV}$

- What if $\mathsf{Sol}(\varphi)/p^c$ is < 1, but $\mathsf{Sol}(\varphi_{BV})/p^{c-1}$ is too large?
  - Can happen for large p
  - Can we reduce p arbitrarily? Need $2^k \leq p < 2^{nk}$
- Solution: Slice each $x_1, \ldots x_n$ into two equal slices
  - Effectively halves k and doubles n
  - Allows smaller prime q ($< p$) for modulus in additional h_i's
  - Expected # models in each cell is now $\mathsf{Sol}(\varphi)/(\mathbf{p^c.q})$
- Recursively apply this technique until cells are “small” enough

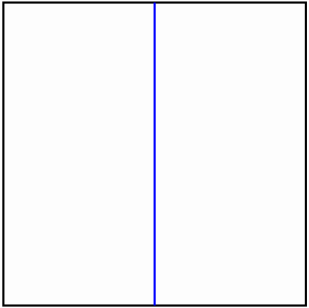
Closer look at $\mathcal{H}_{BV}$

- What if $\mathsf{Sol}(\varphi)/p^c$ is < 1, but $\mathsf{Sol}(\varphi_{BV})/p^{c-1}$ is too large?
  - Can happen for large p
  - Can we reduce p arbitrarily? Need $2^k \leq p < 2^{nk}$
- Solution: Slice each $x_1, \ldots x_n$ into two equal slices
  - Effectively halves k and doubles n
  - Allows smaller prime q $(< p)$ for modulus in additional h_i's
  - Expected # models in each cell is now $\mathsf{Sol}(\varphi)/(\mathbf{p^c.q})$
- Recursively apply this technique until cells are "small" enough

Illustration with non-prime modulus

Closer look at $\mathcal{H}_{BV}$

- What if $\mathrm{Sol}(\varphi)/p^c$ is < 1, but $\mathrm{Sol}(\varphi_{BV})/p^{c-1}$ is too large?
  - Can happen for large p
  - Can we reduce p arbitrarily? Need $2^k \leq p < 2^{nk}$
- Solution: Slice each $x_1, \ldots x_n$ into two equal slices
  - Effectively halves k and doubles n
  - Allows smaller prime q ($< p$) for modulus in additional h_i's
  - Expected # models in each cell is now $\mathrm{Sol}(\varphi)/(\mathbf{p^c.q})$
- Recursively apply this technique until cells are “small” enough



— h_1 with k

Illustration with non-prime modulus

Closer look at $\mathcal{H}_{BV}$

- What if $\mathrm{Sol}(\varphi)/p^c$ is < 1, but $\mathrm{Sol}(\varphi_{BV})/p^{c-1}$ is too large?
  - Can happen for large p
  - Can we reduce p arbitrarily? Need $2^k \leq p < 2^{nk}$
- Solution: Slice each $x_1, \ldots x_n$ into two equal slices
  - Effectively halves k and doubles n
  - Allows smaller prime q $(< p)$ for modulus in additional h_i's
  - Expected # models in each cell is now $\mathrm{Sol}(\varphi)/(\mathbf{p^c.q})$
- Recursively apply this technique until cells are "small" enough

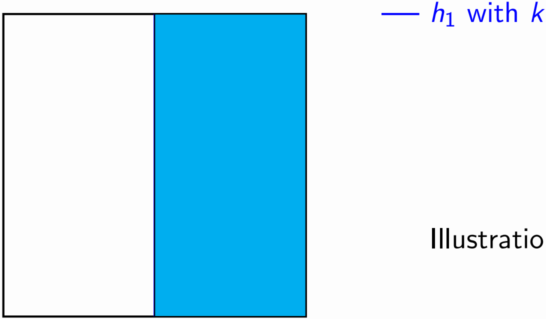
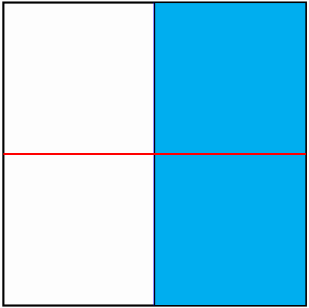


Illustration with non-prime modulus

Closer look at $\mathcal{H}_{BV}$

- What if $\mathrm{Sol}(\varphi)/p^c$ is < 1, but $\mathrm{Sol}(\varphi_{BV})/p^{c-1}$ is too large?
  - Can happen for large p
  - Can we reduce p arbitrarily? Need $2^k \leq p < 2^{nk}$
- Solution: Slice each $x_1, \ldots x_n$ into two equal slices
  - Effectively halves k and doubles n
  - Allows smaller prime q ($< p$) for modulus in additional h_i's
  - Expected # models in each cell is now $\mathrm{Sol}(\varphi)/(\mathbf{p^c.q})$
- Recursively apply this technique until cells are “small” enough



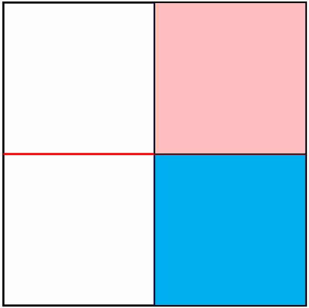
— h_1 with k

— h_2 with k

Illustration with non-prime modulus

Closer look at $\mathcal{H}_{BV}$

- What if $\mathsf{Sol}(\varphi)/p^c$ is < 1, but $\mathsf{Sol}(\varphi_{BV})/p^{c-1}$ is too large?
  - Can happen for large p
  - Can we reduce p arbitrarily? Need $2^k \leq p < 2^{nk}$
- Solution: Slice each $x_1, \ldots x_n$ into two equal slices
  - Effectively halves k and doubles n
  - Allows smaller prime q ($< p$) for modulus in additional h_i's
  - Expected # models in each cell is now $\mathsf{Sol}(\varphi)/(\mathbf{p^c.q})$
- Recursively apply this technique until cells are "small" enough



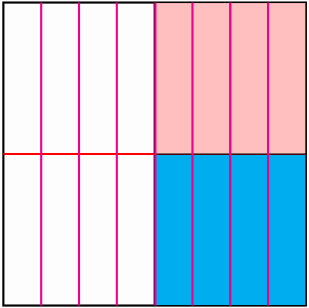
— h_1 with k

— h_2 with k

Illustration with non-prime modulus

Closer look at $\mathcal{H}_{BV}$

- What if $\mathsf{Sol}(\varphi)/p^c$ is < 1, but $\mathsf{Sol}(\varphi_{BV})/p^{c-1}$ is too large?
  - Can happen for large p
  - Can we reduce p arbitrarily? Need $2^k \leq p < 2^{nk}$
- Solution: Slice each $x_1, \ldots x_n$ into two equal slices
  - Effectively halves k and doubles n
  - Allows smaller prime q ($< p$) for modulus in additional h_i's
  - Expected # models in each cell is now $\mathsf{Sol}(\varphi)/(\mathbf{p^c.q})$
- Recursively apply this technique until cells are “small” enough



— h_1 with k

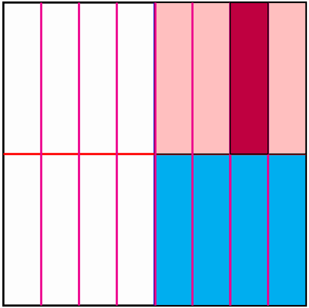
— h_2 with k

— h_3 with $k/2$

Illustration with non-prime modulus

Closer look at $\mathcal{H}_{BV}$

- What if $\mathsf{Sol}(\varphi)/p^c$ is < 1, but $\mathsf{Sol}(\varphi_{BV})/p^{c-1}$ is too large?
  - Can happen for large p
  - Can we reduce p arbitrarily? Need $2^k \leq p < 2^{nk}$
- Solution: Slice each $x_1, \ldots x_n$ into two equal slices
  - Effectively halves k and doubles n
  - Allows smaller prime q $(< p)$ for modulus in additional h_i's
  - Expected # models in each cell is now $\mathsf{Sol}(\varphi)/(\mathbf{p^c.q})$
- Recursively apply this technique until cells are “small” enough



— h_1 with k

— h_2 with k

— h_3 with $k/2$

Illustration with non-prime modulus

Closer look at $\mathcal{H}_{BV}$

- What if $\text{Sol}(\varphi)/p^c$ is < 1, but $\text{Sol}(\varphi_{BV})/p^{c-1}$ is too large?
  - Can happen for large p
  - Can we reduce p arbitrarily? Need $2^k \leq p < 2^{nk}$
- Solution: Slice each $x_1, \ldots x_n$ into two equal slices
  - Effectively halves k and doubles n
  - Allows smaller prime q ($< p$) for modulus in additional h_i's
  - Expected # models in each cell is now $\text{Sol}(\varphi)/(\mathbf{p^c.q})$
- Recursively apply this technique until cells are "small" enough

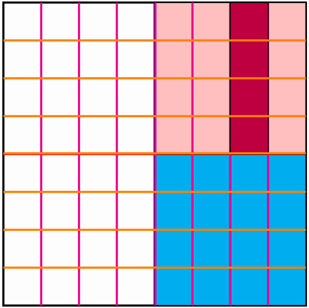


Illustration with non-prime modulus

Closer look at $\mathcal{H}_{BV}$

- What if $\mathsf{Sol}(\varphi)/p^c$ is < 1, but $\mathsf{Sol}(\varphi_{BV})/p^{c-1}$ is too large?
  - Can happen for large p
  - Can we reduce p arbitrarily? Need $2^k \leq p < 2^{nk}$
- Solution: Slice each $x_1, \ldots x_n$ into two equal slices
  - Effectively halves k and doubles n
  - Allows smaller prime q ($< p$) for modulus in additional h_i's
  - Expected # models in each cell is now $\mathsf{Sol}(\varphi)/(\mathbf{p^c.q})$
- Recursively apply this technique until cells are "small" enough

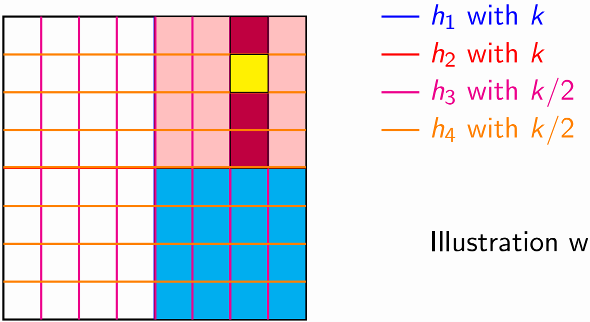


Illustration with non-prime modulus

Closer look at $\mathcal{H}_{BV}$

- Let $M = p_1^{c_1} \cdot p_2^{c_2} \cdots p_r^{c_r}$, where
  - $p_1, \ldots p_r$ are primes such that
    - $2^{k-i} \leq p_i < 2^{nk}$ for all $i \in \{1, \ldots r\}$
  - $1 < 2^{nk}/M$

Closer look at $\mathcal{H}_{BV}$

- Let $M = p_1^{c_1} \cdot p_2^{c_2} \cdots p_r^{c_r}$, where
  - $p_1, \ldots p_r$ are primes such that
    - $2^{k-i} \leq p_i < 2^{nk}$ for all $i \in \{1, \ldots r\}$
  - $1 < 2^{nk}/M$
- Final version of $\mathcal{H}_{BV}$
  Every hash function in $\mathcal{H}_{BV}$ is a tuple of $c_1 + c_2 + \ldots c_r$ linear modular hash functions
  - c_1 hash functions with modulus p_1
  - c_2 hash functions with modulus p_2
  - ...
  - c_r hash functions with modulus p_r

Closer look at $\mathcal{H}_{BV}$

- Let $M = p_1^{c_1} \cdot p_2^{c_2} \cdots p_r^{c_r}$, where
  - $p_1, \ldots p_r$ are primes such that
    - $2^{k-i} \leq p_i < 2^{nk}$ for all $i \in \{1, \ldots r\}$
  - $1 < 2^{nk}/M$
- Final version of $\mathcal{H}_{BV}$
  Every hash function in $\mathcal{H}_{BV}$ is a tuple of $c_1 + c_2 + \ldots c_r$ linear modular hash functions
  - c_1 hash functions with modulus p_1
  - c_2 hash functions with modulus p_2
  - ...
  - c_r hash functions with modulus p_r
- Every hash function $h_{BV} \in \mathcal{H}_{BV}$ maps
  $$\{0,1\}^{nk} \text{ to } (\mathbb{Z}_{p_1})^{c_1} \times (\mathbb{Z}_{p_1})^{c_1} \times \cdots \times (\mathbb{Z}_{p_r})^{c_r}$$

Properties of $\mathcal{H}_{BV}$

Theorem: $\mathcal{H}_{BV}$ is 2-universal

For every $\alpha_1, \alpha_2 \in (\mathbb{Z}_{p_1})^{c_1} \times \cdots \times (\mathbb{Z}_{p_r})^{c_r}$, every $\mathbf{X}_1, \mathbf{X}_2 \in \{0,1\}^{nk}$, and every hash function h chosen randomly from $\mathcal{H}_{BV}$, $\Pr[h(\mathbf{X}_1) = \alpha_1 \ \wedge \ h(\mathbf{X}_2) = \alpha_2] = \Pr[h(\mathbf{X}_1) = \alpha_1] \cdot \Pr[h(\mathbf{X}_2) = \alpha_2] = (1/p_1)^{2c_1} \cdot (1/p_2)^{2c_2} \cdots (1/p_r)^{2c_r}$.

Properties of $\mathcal{H}_{BV}$

Theorem: $\mathcal{H}_{BV}$ is 2-universal

For every $\alpha_1, \alpha_2 \in (\mathbb{Z}_{p_1})^{c_1} \times \cdots \times (\mathbb{Z}_{p_r})^{c_r}$, every $\mathbf{X}_1, \mathbf{X}_2 \in \{0,1\}^{nk}$, and every hash function h chosen randomly from $\mathcal{H}_{BV}$, $\Pr[h(\mathbf{X}_1) = \alpha_1 \ \wedge \ h(\mathbf{X}_2) = \alpha_2] = \Pr[h(\mathbf{X}_1) = \alpha_1] \cdot \Pr[h(\mathbf{X}_2) = \alpha_2] = (1/p_1)^{2c_1} \cdot (1/p_2)^{2c_2} \cdots (1/p_r)^{2c_r}$.

$\mathcal{H}_{BV}$ can be used for bit-vector model counting

Putting it all together: SMTApproxMC

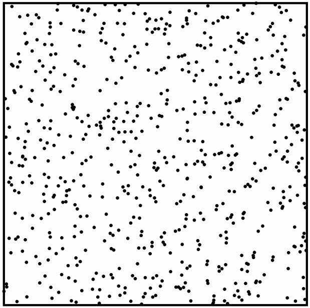
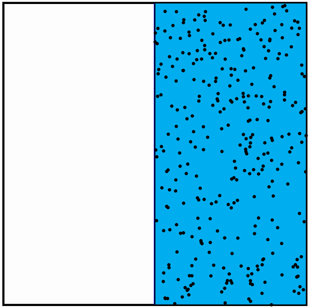


Illustration with non-prime modulus

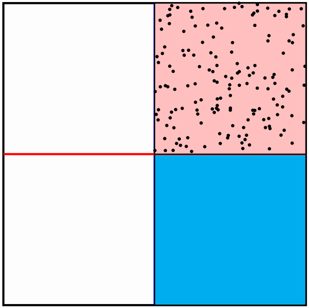
Putting it all together: SMTApproxMC



— (h_1 with p_1)

Illustration with non-prime modulus

Putting it all together: SMTApproxMC

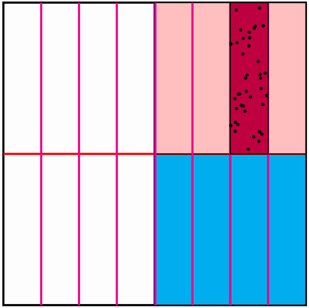


— (h_1 with p_1)

— (h_1, h_2 with p_1)

Illustration with non-prime modulus

Putting it all together: SMTApproxMC



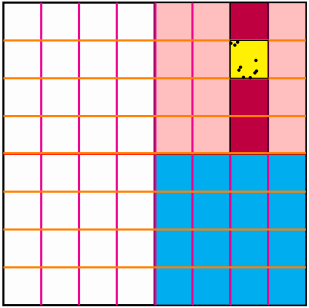
— (h_1 with p_1)

— (h_1, h_2 with p_1)

— (h_1, h_2 with p_1; h_3 with p_2)

Illustration with non-prime modulus

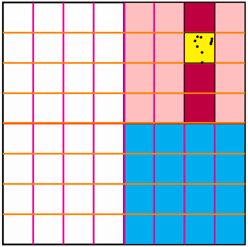
Putting it all together: SMTApproxMC



- — (h_1 with p_1)
- — (h_1, h_2 with p_1)
- — (h_1, h_2 with p_1; h_3 with p_2)
- — (h_1, h_2 with p_1; h_3, h_4 with p_2)

Illustration with non-prime modulus

Putting it all together: SMTApproxMC



— (h_1 with p_1)
— (h_1, h_2 with p_1)
— (h_1, h_2 with p_1; h_3 with p_2)
— (h_1, h_2 with p_1; h_3, h_4 with p_2)

Illustration with non-prime modulus

- Given bit-vector constraint φ, ε (> 0), and $\delta \in (0, 1]$
  (1) Determine pivot from ε, repCount from δ and initial $\mathcal{H}_{BV}$
  (2) Randomly choose $h \in \mathcal{H}_{BV}$ and $\alpha \in \text{range}(h)$
  (3) Let $\kappa = |\text{Sol}(\varphi(\mathbf{X}) \wedge (h(\mathbf{X}) = \alpha))|$
  (4) If $\kappa \notin (0, \text{pivot}]$ then
    - Update $\mathcal{H}_{BV}$ with next linear modular hash function
    - Go to (2)
  (5) Else, AddToListOfSolns(κ) and repeat (2)-(4) repCount times
  (6) Return median of ListOfSolns

Need for SMT solver

$$\boxed{\text{Step (3): Count \# solutions of } \varphi(\mathbf{X}) \wedge (h(\mathbf{X}) = \alpha)}$$

Need for SMT solver

$$\boxed{\text{Step (3): Count } \# \text{ solutions of } \varphi(\mathbf{X}) \wedge (h(\mathbf{X}) = \alpha)}$$

- Solution: Use Satisfiability Modulo Theories (SMT) solver for theory of bit-vectors

Need for SMT solver

$\boxed{\text{Step (3): Count \# solutions of } \varphi(\mathbf{X}) \wedge (h(\mathbf{X}) = \alpha)}$

- Solution: Use Satisfiability Modulo Theories (SMT) solver for theory of bit-vectors
- Uses axioms and inference rules from first-order theory of bit-vectors as much as possible
  - $x_{[l]} + 0_{[l]} = x_{[l]}$
  - $\mathrm{concat}(\mathrm{extract}(\mathrm{x_{[l]}}, 0, \mathrm{m}), \mathrm{extract}(\mathrm{x_{[l]}}, \mathrm{m}+1, \mathrm{l}-1) = x_{[l]}$, if $0 \leq m < l - 1$
  - $\mathrm{leftshift}(x_{[l]}, t) = x/2^t$
  - ... plenty of well-studied rules

Need for SMT solver

Step (3): Count # solutions of $\varphi(\mathbf{X}) \wedge (h(\mathbf{X}) = \alpha)$

- Solution: Use Satisfiability Modulo Theories (SMT) solver for theory of bit-vectors
- Uses axioms and inference rules from first-order theory of bit-vectors as much as possible
  - $x_{[l]} + 0_{[l]} = x_{[l]}$
  - $\mathrm{concat}(\mathrm{extract}(\mathrm{x}_{[\mathrm{l}]}, 0, \mathrm{m}), \mathrm{extract}(\mathrm{x}_{[\mathrm{l}]}, \mathrm{m}+1, \mathrm{l}-1) = x_{[l]}$, if $0 \leq m < l - 1$
  - $\mathrm{leftshift}(x_{[l]}, t) = x/2^t$
  - ... plenty of well-studied rules
- Bit-blast only if no rule applies
- Desirable: efficient reasoning about φ + linear constraints modulo primes

Need for SMT solver

Step (3): Count # solutions of $\varphi(\mathbf{X}) \wedge (h(\mathbf{X}) = \alpha)$

- Solution: Use Satisfiability Modulo Theories (SMT) solver for theory of bit-vectors
- Uses axioms and inference rules from first-order theory of bit-vectors as much as possible
  - $x_{[l]} + 0_{[l]} = x_{[l]}$
  - $\mathrm{concat}(\mathrm{extract}(\mathrm{x_{[l]}}, 0, \mathrm{m}), \mathrm{extract}(\mathrm{x_{[l]}}, \mathrm{m}+1, \mathrm{l}-1) = x_{[l]}$, if $0 \leq m < l - 1$
  - $\mathrm{leftshift}(x_{[l]}, t) = x/2^t$
  - ... plenty of well-studied rules
- Bit-blast only if no rule applies
- Desirable: efficient reasoning about φ + linear constraints modulo primes
  - Linear constraints modulo primes admit Gaussian elimination
  - Need to integrate Gaussian elimination within existing SMT solvers
    - Yet to be fully solved

Theoretical guarantees and Performance

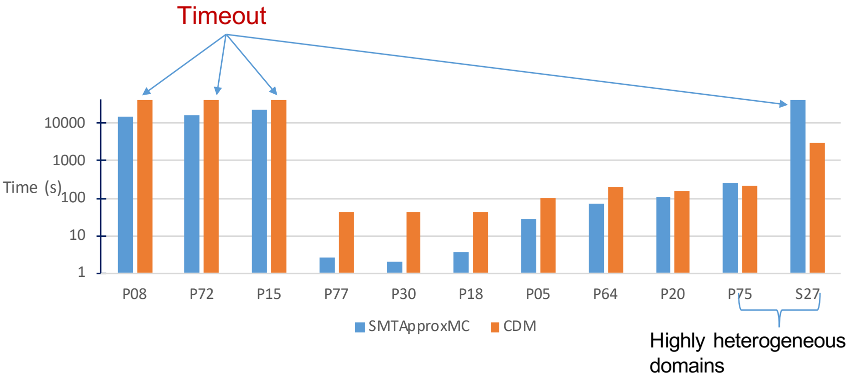
Theorem

- $\Pr[\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon} \leq \mathsf{SMTApproxMC}(\varphi, \varepsilon, \delta) \leq (1+\varepsilon) \cdot |\mathsf{Sol}(\varphi)|] \geq 1 - \delta$
- SMTApproxMC$(\varphi, \varepsilon, \delta)$ runs in time polynomial in $|\varphi|$, $1/\varepsilon$ and $\log(1/\delta)$.

Theoretical guarantees and Performance

Theorem

- $\Pr[\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon} \leq \mathsf{SMTApproxMC}(\varphi,\varepsilon,\delta) \leq (1+\varepsilon)\cdot|\mathsf{Sol}(\varphi)|] \geq 1-\delta$
- $\mathsf{SMTApproxMC}(\varphi,\varepsilon,\delta)$ runs in time polynomial in $|\varphi|$, $1/\varepsilon$ and $\log(1/\delta)$.



Timeout

Highly heterogeneous domains

Figure

Domain-specific decomposition + Propositional MC

Key idea:

- Decompose domain into finite union of hyper-rectangles
- Ensure that only a "small" number (ν) of hyper-rectangles are "cut" by the solution space
  - For most hyper-rectangles, either all points are solutions, or all points are non-solutions
- Let M = number of hyper-rectangles with at least one solution
- Let V = uniform measure weight of each hyper-rectaangle
- Then $(M - \nu) \times V \leq$ Required Count $\leq M \times V$

Counting in Bounded Integer+Rational Arithmetic [Chistikov et al 2015]

- Constraints of the form $\varphi(x) = \exists u, \Phi(x, u)$
  - Allows top-level existential quantifiers (projection)
- k free variables, each takes values in interval $[0, M]$

Counting in Bounded Integer+Rational Arithmetic [Chistikov et al 2015]

- Constraints of the form $\varphi(x) = \exists u, \Phi(x, u)$
  - Allows top-level existential quantifiers (projection)
- k free variables, each takes values in interval $[0, M]$
- Choose a "large" integer s and divide $[0, M]^k$ into s^k sub-cubes of side $\rho = M/s$

Counting in Bounded Integer+Rational Arithmetic [Chistikov et al 2015]

- Constraints of the form $\varphi(x) = \exists u, \Phi(x, u)$
  - Allows top-level existential quantifiers (projection)
- k free variables, each takes values in interval $[0, M]$
- Choose a "large" integer s and divide $[0, M]^k$ into s^k sub-cubes of side $\rho = M/s$
- $y_1, \ldots y_k$: new bounded integer variables, each with domain $\{0, 1, \ldots s - 1\}$
  - Each valuation of $y_1, \ldots y_k$ identifies a unique small cube $C(y_1, \ldots y_k)$

Counting in Bounded Integer+Rational Arithmetic [Chistikov et al 2015]

- Constraints of the form $\varphi(x) = \exists u, \Phi(x, u)$
  - Allows top-level existential quantifiers (projection)
- k free variables, each takes values in interval $[0, M]$
- Choose a "large" integer s and divide $[0, M]^k$ into s^k sub-cubes of side $\rho = M/s$
- $y_1, \ldots y_k$: new bounded integer variables, each with domain $\{0, 1, \ldots s-1\}$
  - Each valuation of $y_1, \ldots y_k$ identifies a unique small cube $C(y_1, \ldots y_k)$
- Define $\psi(y_1, \ldots y_k)$ as follows:
  - $\psi(y_1, \ldots y_k) \equiv \exists x \left(\varphi(x) \bigwedge_{i=1}^{k} (y_i.\rho \leq x_i \leq (y_j + 1) \cdot \rho)\right)$
  - $\psi(y_1, \ldots y_k) =$ true iff at least one point in $C(y_1, \ldots y_k)$ satisfies $\varphi(x)$.

Counting in Bounded Integer+Rational Arithmetic [Chistikov et al 2015]

- Constraints of the form $\varphi(x) = \exists u, \Phi(x, u)$
  - Allows top-level existential quantifiers (projection)
- k free variables, each takes values in interval $[0, M]$
- Choose a "large" integer s and divide $[0, M]^k$ into s^k sub-cubes of side $\rho = M/s$
- $y_1, \ldots y_k$: new bounded integer variables, each with domain $\{0, 1, \ldots s-1\}$
  - Each valuation of $y_1, \ldots y_k$ identifies a unique small cube $C(y_1, \ldots y_k)$
- Define $\psi(y_1, \ldots y_k)$ as follows:
  - $\psi(y_1, \ldots y_k) \equiv \exists x \left(\varphi(x) \bigwedge_{i=1}^{k} (y_i.\rho \leq x_i \leq (y_j + 1) \cdot \rho)\right)$
  - $\psi(y_1, \ldots y_k) =$ true iff at least one point in $C(y_1, \ldots y_k)$ satisfies $\varphi(x)$.
- Assign uniform measure $\rho = M/s$ to each $y_i \in \{0, \ldots s-1\}$

Counting in Bounded Integer+Rational Arithmetic [Chistikov et al 2015]

- If at most J small cubes are "cut" by solution space, then $(|\mathsf{Sol}(\psi)| - J) \cdot \delta^k \;\leq\; \mathit{ModelCount} \;\leq\; |\mathsf{Sol}(\psi)| \cdot \delta^k$

Counting in Bounded Integer+Rational Arithmetic [Chistikov et al 2015]

- If at most J small cubes are "cut" by solution space, then $(|\mathsf{Sol}(\psi)| - J) \cdot \delta^k \;\leq\; \mathit{ModelCount} \;\leq\; |\mathsf{Sol}(\psi)| \cdot \delta^k$
- Using a result from Dyer & Frieze 1998, Chistikov et al showed
  - If $s \geq \lceil 2^{m+2k}.k^2.M^k/(\varepsilon/2) \rceil$, then $J \leq (1/\delta^k) \cdot (\varepsilon/2)$, for $\varepsilon > 0$

Counting in Bounded Integer+Rational Arithmetic [Chistikov et al 2015]

- If at most J small cubes are "cut" by solution space, then $(|\mathsf{Sol}(\psi)| - J) \cdot \delta^k \ \leq \ \mathit{ModelCount} \ \leq \ |\mathsf{Sol}(\psi)| \cdot \delta^k$
- Using a result from Dyer & Frieze 1998, Chistikov et al showed
  - If $s \geq \lceil 2^{m+2k}.k^2.M^k/(\varepsilon/2) \rceil$, then $J \leq (1/\delta^k) \cdot (\varepsilon/2)$, for $\varepsilon > 0$
- Finally, $|\mathsf{Sol}(\psi)(y_1, \ldots y_k)|$ is computed by
  - Propositional encoding of finite domain
  - Propositional universal hashing
  - Invoking SMT solver (theory of integer + rational linear arithmetic) to determine if $\psi(y_1, \ldots y_k)$ is true for a given $y_1, \ldots y_k$.

Weighted model integration (Belle2015, Belle2017)

- Generalizes weighted model counting

Weighted model integration

(Belle2015, Belle2017)

- Generalizes weighted model counting
- Formula $\varphi(\mathbf{x}, \mathbf{A})$, where
  - $\mathbf{x} = (x_1, \ldots x_n)$: real valued variables
  - $\mathbf{A} = (A_1, \ldots A_m)$: atomic propositions

Weighted model integration

(Belle2015, Belle2017)

- Generalizes weighted model counting
- Formula $\varphi(\mathbf{x}, \mathbf{A})$, where
    - $\mathbf{x} = (x_1, \dots x_n)$: real valued variables
    - $\mathbf{A} = (A_1, \dots A_m)$: atomic propositions
- Weight function $w : \mathbb{R}^n \times \{0, 1\}^k \to \mathbb{R}$

Weighted model integration

(Belle2015, Belle2017)

- Generalizes weighted model counting
- Formula $\varphi(\mathbf{x}, \mathbf{A})$, where
    - $\mathbf{x} = (x_1, \dots x_n)$: real valued variables
    - $\mathbf{A} = (A_1, \dots A_m)$: atomic propositions
- Weight function $w : \mathbb{R}^n \times \{0, 1\}^k \rightarrow \mathbb{R}$
- $WMI(\varphi, w) = \Sigma_{\sigma \in \{0,1\}^m} \int_{\varphi(\mathbf{x},\sigma)} w(\mathbf{x}, \sigma) \mathrm{d}\mathbf{x}$.

Weighted model integration

(Belle2015, Belle2017)

- Generalizes weighted model counting
- Formula $\varphi(\mathbf{x}, \mathbf{A})$, where
  - $\mathbf{x} = (x_1, \ldots x_n)$: real valued variables
  - $\mathbf{A} = (A_1, \ldots A_m)$: atomic propositions
- Weight function $w : \mathbb{R}^n \times \{0, 1\}^k \rightarrow \mathbb{R}$
- $WMI(\varphi, w) = \Sigma_{\sigma \in \{0,1\}^m} \int_{\varphi(\mathbf{x},\sigma)} w(\mathbf{x}, \sigma) \mathrm{d}\mathbf{x}$.

Example (Belle2017):

- $\varphi(x, A) \equiv \leftrightarrow (x \geq 0)) \wedge (x \geq -1) \wedge (x \leq 1)$
- $w(x, A) =$ if (A) then x else $-x$

Weighted model integration (Belle2015, Belle2017)

- Generalizes weighted model counting
- Formula $\varphi(\mathbf{x}, \mathbf{A})$, where
  - $\mathbf{x} = (x_1, \dots x_n)$: real valued variables
  - $\mathbf{A} = (A_1, \dots A_m)$: atomic propositions
- Weight function $w : \mathbb{R}^n \times \{0,1\}^k \to \mathbb{R}$
- $WMI(\varphi, w) = \Sigma_{\sigma \in \{0,1\}^m} \int_{\varphi(\mathbf{x},\sigma)} w(\mathbf{x}, \sigma) \mathrm{d}\mathbf{x}$.

Example (Belle2017):

- $\varphi(x, A) \equiv \leftrightarrow (x \geq 0)) \wedge (x \geq -1) \wedge (x \leq 1)$
- $w(x, A) =$ if (A) then x else $-x$
- $WMI(\varphi, w) = \int_{[-1,0)} (-x) \mathrm{d}x + \int_{[0,1]} (x) \mathrm{d}x = \frac{1}{2} + \frac{1}{2} = 1$

Outline

Part 1:	Applications
Part 2:	Prior Work
Part 3:	Overview of SAT Solving
Part 4:	Hashing-based Approach for Uniform Distribution
Part 5:	Beyond Propositional
Part 6:	Challenges

Part VI

Challenges

Constrained Counting

- Given
    - Boolean variables $X_1, X_2, \cdots X_n$
    - Formula F over $X_1, X_2, \cdots X_n$
    - Weight Function W: $\{0,1\}^n \mapsto [0,1]$
- ExactWeightedCount(F): Compute $W(F)$?
    - #P-complete (Valiant 1979)

Constrained Counting

- Given
  - Boolean variables $X_1, X_2, \cdots X_n$
  - Formula F over $X_1, X_2, \cdots X_n$
  - Weight Function W: $\{0,1\}^n \mapsto [0,1]$
- ExactWeightedCount(F): Compute $W(F)$?
  - #P-complete (Valiant 1979)
- ApproxWeightedCount($F, W, \varepsilon, \delta$): Compute C such that

$$\Pr[\frac{W(F)}{1+\varepsilon} \le C \le W(F)(1+\varepsilon)] \ge 1-\delta$$

From Weighted to Uniform Counting

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

Boolean Formula F'

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

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

From Weighted to Uniform Counting

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

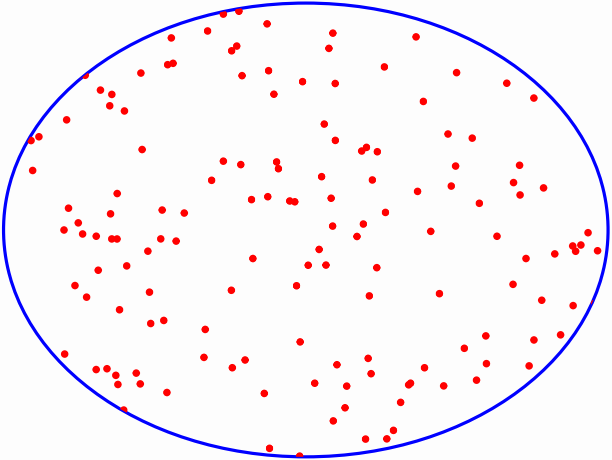
Boolean Formula F'

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

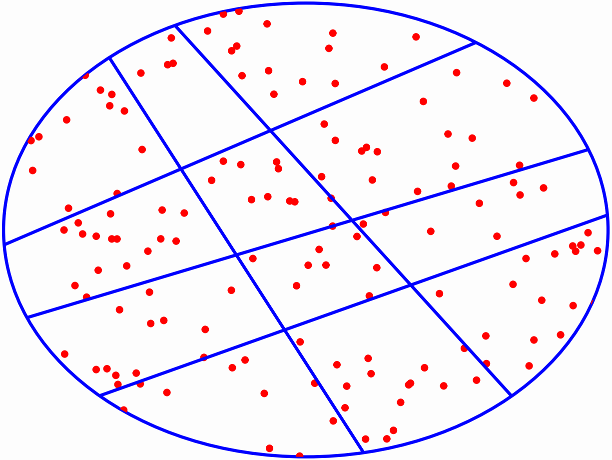
- Key Idea: Encode weight function as a set of constraints
- Caveat: $|F'| = O(|F| + |W|)$
- Increase in the number of variables $\implies$ Increase in the size of XORs
- $|\mathsf{Sol}(F')| > |\mathsf{Sol}(F)|$: Increase in number of solutions $\implies$ Increase in the number of XORs

Challenge Design better reductions that are amenable to hashing-based approximate techniques.

Summing up Mass of Dots

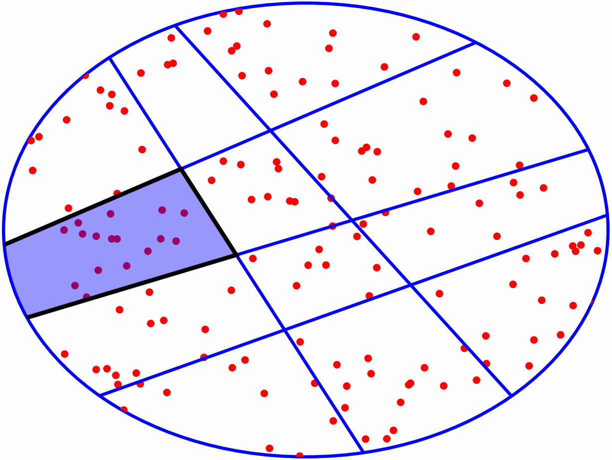


Summing up Mass of Dots



Summing up Mass of Dots

Pick a random cell



Estimate = Mass in a cell × Number of cells

Hashing-based Approach

How does equal number of solutions translate to equal weight?

It does not!

- Let w_{max}: maximum weight of a solution;
  w_{min}: minimum weight of a solution
- Two cells with equal number of solutions, say t, can have weights $w_{max} \times t$ and $w_{min} \times t$.

Hashing-based Approach

How does equal number of solutions translate to equal weight?

It does not!

- Let w_{max}: maximum weight of a solution;
  w_{min}: minimum weight of a solution
- Two cells with equal number of solutions, say t, can have weights $w_{max} \times t$ and $w_{min} \times t$.
- tilt (F) $= \frac{w_{max}}{w_{min}}$

Hashing-based Approach

How does equal number of solutions translate to equal weight?

It does not!

- Let w_{max}: maximum weight of a solution;
  w_{min}: minimum weight of a solution
- Two cells with equal number of solutions, say t, can have weights $w_{max} \times t$ and $w_{min} \times t$.
- tilt (F) $= \frac{w_{max}}{w_{min}}$
- The number of SAT calls increase by a factor of tilt

Hashing-based Approach

How does equal number of solutions translate to equal weight?

It does not!

- Let w_{max}: maximum weight of a solution;
  w_{min}: minimum weight of a solution
- Two cells with equal number of solutions, say t, can have weights $w_{max} \times t$ and $w_{min} \times t$.
- tilt (F) $= \frac{w_{max}}{w_{min}}$
- The number of SAT calls increase by a factor of tilt

Divide into multiple problems with each of the problems with small tilt

Hashing-based Approach

How does equal number of solutions translate to equal weight?

It does not!

- Let w_{max}: maximum weight of a solution;
  w_{min}: minimum weight of a solution
- Two cells with equal number of solutions, say t, can have weights $w_{max} \times t$ and $w_{min} \times t$.
- tilt (F) $= \frac{w_{max}}{w_{min}}$
- The number of SAT calls increase by a factor of tilt

Divide into multiple problems with each of the problems with small tilt

- Generate log(tilt) formulas: $F^i = F \bigwedge \frac{w_{max}}{2^i} \leq w(\sigma) \leq \frac{w_{max}}{2^{i+1}}$

Hashing-based Approach

How does equal number of solutions translate to equal weight?

It does not!

- Let w_{max}: maximum weight of a solution;
  w_{min}: minimum weight of a solution
- Two cells with equal number of solutions, say t, can have weights $w_{max} \times t$ and $w_{min} \times t$.
- tilt (F) $= \frac{w_{max}}{w_{min}}$
- The number of SAT calls increase by a factor of tilt

Divide into multiple problems with each of the problems with small tilt

- Generate log(tilt) formulas: $F^i = F \bigwedge \frac{w_{max}}{2^i} \leq w(\sigma) \leq \frac{w_{max}}{2^{i+1}}$
- tilt $(F^i) = 2$

Hashing-based Approach

How does equal number of solutions translate to equal weight?

It does not!

- Let w_{max}: maximum weight of a solution; w_{min}: minimum weight of a solution
- Two cells with equal number of solutions, say t, can have weights $w_{max} \times t$ and $w_{min} \times t$.
- tilt (F) $= \frac{w_{max}}{w_{min}}$
- The number of SAT calls increase by a factor of tilt

Divide into multiple problems with each of the problems with small tilt

- Generate log(tilt) formulas: $F^i = F \bigwedge \frac{w_{max}}{2^i} \leq w(\sigma) \leq \frac{w_{max}}{2^{i+1}}$
- tilt $(F^i) = 2$
- Use Pseudo Boolean (PB) constraints to encode $\frac{w_{max}}{2^i} \leq w(\sigma) \leq \frac{w_{max}}{2^{i+1}}$ when weight function is implicitly described

Hashing-based Approach

How does equal number of solutions translate to equal weight?

It does not!

- Let w_{max}: maximum weight of a solution;
  w_{min}: minimum weight of a solution
- Two cells with equal number of solutions, say t, can have weights $w_{max} \times t$ and $w_{min} \times t$.
- tilt (F) $= \frac{w_{max}}{w_{min}}$
- The number of SAT calls increase by a factor of tilt

Divide into multiple problems with each of the problems with small tilt

- Generate log(tilt) formulas: $F^i = F \bigwedge \frac{w_{max}}{2^i} \leq w(\sigma) \leq \frac{w_{max}}{2^{i+1}}$
- tilt $(F^i) = 2$
- Use Pseudo Boolean (PB) constraints to encode $\frac{w_{max}}{2^i} \leq w(\sigma) \leq \frac{w_{max}}{2^{i+1}}$ when weight function is implicitly described

No Good CNF+PB+XOR solver

Hashing-based Approach

How does equal number of solutions translate to equal weight?

It does not!

- Let w_{max}: maximum weight of a solution;
  w_{min}: minimum weight of a solution
- Two cells with equal number of solutions, say t, can have weights $w_{max} \times t$ and $w_{min} \times t$.
- tilt (F) $= \frac{w_{max}}{w_{min}}$
- The number of SAT calls increase by a factor of tilt

Divide into multiple problems with each of the problems with small tilt

- Generate log(tilt) formulas: $F^i = F \bigwedge \frac{w_{max}}{2^i} \leq w(\sigma) \leq \frac{w_{max}}{2^{i+1}}$
- tilt $(F^i) = 2$
- Use Pseudo Boolean (PB) constraints to encode $\frac{w_{max}}{2^i} \leq w(\sigma) \leq \frac{w_{max}}{2^{i+1}}$ when weight function is implicitly described

No Good CNF+PB+XOR solver

Challenge Design solvers that can handle CNF+PB+XOR

WISH: Weighted Counting via MaxSAT

- Let all the solutions be arranged in decreasing order of their weights: $w_1, w_2, \cdots w_{|\mathsf{Sol}(F)|}$
- $W(F) = \sum_{i \in [|\mathsf{Sol}(F)|]} w_i$
- Viewing this summation as discrete Riemann sums, we observe the following

$$\frac{W(F)}{2} \leq \sum_{i \in \log|\mathsf{Sol}(F)|} w_i \times 2^{i+1} \leq 2 \times W(F)$$

- Note that we only need to identify $\log|\mathsf{Sol}(F)|$ many weights.

WISH: Weighted Counting via MaxSAT

- Let all the solutions be arranged in decreasing order of their weights: $w_1, w_2, \cdots w_{|\mathsf{Sol}(F)|}$
- $W(F) = \sum\limits_{i \in [|\mathsf{Sol}(F)|]} w_i$
- Viewing this summation as discrete Riemann sums, we observe the following

$$\frac{W(F)}{2} \leq \sum_{i \in \log|\mathsf{Sol}(F)|} w_i \times 2^{i+1} \leq 2 \times W(F)$$

- Note that we only need to identify $\log|\mathsf{Sol}(F)|$ many weights.
- Solution: Use hashing to find these weights

WISH: Weighted Counting via MaxSAT

$$\frac{W(F)}{2} \leq \sum_{i \in \log |\mathrm{Sol}(F)|} w_i \times 2^{i+1} \leq 2 \times W(F)$$

How do we get w_i?

- w_i: ith largest weighted solution
- $w_1 = MaxWeight(F, W)$
- $\mathrm{E}[MaxWeight(F \wedge \text{One Random XOR})] =$

WISH: Weighted Counting via MaxSAT

$$\frac{W(F)}{2} \leq \sum_{i \in \log|\mathrm{Sol}(F)|} w_i \times 2^{i+1} \leq 2 \times W(F)$$

How do we get w_i?

- w_i: ith largest weighted solution
- $w_1 = MaxWeight(F, W)$
- $\mathrm{E}[MaxWeight(F \wedge \text{One Random XOR})] = w_2$
- $\mathrm{E}[MaxWeight(F \wedge \text{Two Random XOR})] =$

WISH: Weighted Counting via MaxSAT

$$\frac{W(F)}{2} \leq \sum_{i \in \log|\mathrm{Sol}(F)|} w_i \times 2^{i+1} \leq 2 \times W(F)$$

How do we get w_i?

- w_i: ith largest weighted solution
- $w_1 = MaxWeight(F, W)$
- $\mathrm{E}[MaxWeight(F \wedge \text{One Random XOR})] = w_2$
- $\mathrm{E}[MaxWeight(F \wedge \text{Two Random XOR})] = w_3$
- $\mathrm{E}[MaxWeight(F \wedge \text{i Random XOR})] = w_{i+1}$

(Ermon et al 2014, 2016, Achlioptas et al 2017, 2018)

No Good solvers to handle MaxSAT+XOR

WISH: Weighted Counting via MaxSAT

$$\frac{W(F)}{2} \leq \sum_{i \in \log|\mathrm{Sol}(F)|} w_i \times 2^{i+1} \leq 2 \times W(F)$$

How do we get w_i?

- w_i: ith largest weighted solution
- $w_1 = MaxWeight(F, W)$
- $\mathrm{E}[MaxWeight(F \wedge \text{One Random XOR})] = w_2$
- $\mathrm{E}[MaxWeight(F \wedge \text{Two Random XOR})] = w_3$
- $\mathrm{E}[MaxWeight(F \wedge \text{i Random XOR})] = w_{i+1}$

(Ermon et al 2014, 2016, Achlioptas et al 2017, 2018)

No Good solvers to handle MaxSAT+XOR

Challenge: Design MaxSAT solvers that can handle XORs

2-Universal Hash Functions

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

2-Universal Hash Functions

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

$$\begin{aligned} X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{\mathcal{I}-2} \oplus 1 &= 0 && (Q_1) \\ X_2 \oplus X_5 \oplus X_6 \cdots \oplus X_{\mathcal{I}-1} &= 1 && (Q_2) \\ &\cdots && (\cdots) \\ X_1 \oplus X_2 \oplus X_5 \cdots \oplus X_{\mathcal{I}-2} \oplus 1 &= 1 && (Q_m) \end{aligned}$$

- $h(X) = AX \oplus b$
  - A: $(0,1)$ matrix with every entry is 1 with prob. $\frac{1}{2}$
  - b: $(0,1)$ vector with every entry is 1 with prob. $\frac{1}{2}$
- Solutions in a cell: $F \wedge Q_1 \cdots \wedge Q_m$

2-Universal Hash Functions

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

$$\begin{aligned} X_1 \oplus X_3 \oplus X_6 \cdots \oplus X_{\mathcal{I}-2} \oplus 1 &= 0 && (Q_1) \\ X_2 \oplus X_5 \oplus X_6 \cdots \oplus X_{\mathcal{I}-1} &= 1 && (Q_2) \\ \cdots && (\cdots) \\ X_1 \oplus X_2 \oplus X_5 \cdots \oplus X_{\mathcal{I}-2} \oplus 1 &= 1 && (Q_m) \end{aligned}$$

- $h(X) = AX \oplus b$
  - A: $(0,1)$ matrix with every entry is 1 with prob. $\frac{1}{2}$
  - b: $(0,1)$ vector with every entry is 1 with prob. $\frac{1}{2}$
- Solutions in a cell: $F \wedge Q_1 \cdots \wedge Q_m$
- Can we choose XORs with $p < \frac{1}{2}$?

Low Density Parity Constraints

$h : \{0, 1\}^{\mathcal{I}} \rightarrow \{0, 1\}^m : h(X) = AX \oplus b$, where entries in b are chosen with $p = \frac{1}{2}$

- Let entries in A be chosen with $p < \frac{1}{2}$
- $\mu = \frac{|\mathsf{Sol}(F)|}{2^m}$
- $\sigma^2 = \sum_{y,z \in \mathsf{Sol}(F)} A(y - z) = 0$
- Based on analysis from Mackay et al, one can derive $\sigma^2 \leq \mathrm{Boost}\mu^2$
- Remember for $p = \frac{1}{2}$, we had $\sigma^2 \leq \mu$ (we have $\mu > 1$)

(Ermon et al 2014, 2016, Achlioptas et al 2017, 2018)

Low Density Parity Constraints

- Chebyshev Inequality: $Pr[||X - \mu| \geq \frac{\varepsilon}{(1+\varepsilon)}\mu] \leq \frac{\sigma^2}{\frac{varepsilon^2}{(1+\varepsilon)^2}\mu^2}$
- When $\sigma^2 \leq \mu$
  - For $\varepsilon < 1$, we choose appropriate m such $\mu \times \frac{varepsilon^2}{(1+\varepsilon)^2} > c$
- For $\sigma^2 \leq \mathrm{Boost} \cdot \mu^2$
  - Boost leads to $g(\mathrm{Boost})$ factor of more SAT calls
  - The best result so far puts $g(\mathrm{Boost}) > 10,000$ for p 0.2
  - Significant slowdown due to large number of SAT calls.
- Challenge: Is there free lunch here, i.e. achieving low density without loss of runtime performance?

Summary

- Discrete Integration (Constrained Counting) and Sampling (Constrained Sampling) are important problems with wide variety of applications
- SAT revolution allows us to design techniques that can make *smart* usage of SAT solvers.
- Hashing-based paradigm provides sweet spot in terms of guarantees and performance
- For uniform distribution: From hundreds to hundreds of thousands of variables
- Future Challenges:
    1. Beyond propositional domain (take advantage of SMT solvers)
    2. Generalized weighted distributions
    3. Low density parity constraints

Thank You for being a wonderful audience this afternoon

Acknowledgments:
Joao Marques Silva (for sharing LaTeX template and slides on SAT solving)

Collaborators: Jeffrey Dudek, Leonardo Duenas-Osorio, Alexander Ivrii, Daniel Fremont, Dror Fried, William Hung, Sharad Malik, John Mellor-Crummey, Rakesh Mistry, Roger Parades, Sanjit Seshia, Mate Soos, Aditya Shrotri, and Moshe Vardi.

Researchers in Community for wonderful discussions over the years: Dimitris Achlioptas, Fahiem Bacchus, Vaishak Belle, Guy Van den Broek, Adnan Darwiche, Rina Dechter, Zayd Hammoudeh, Stefano Ermon, Carla Gomes, Rupak Majumdar, Mark Wegman, Ashish Sabharwal, and Bart Selman.

Slides will be available at `https://tinyurl.com/ijcai18tutorial`