Probabilistic Program Analysis

GTTSE part 3

Matthew B. Dwyer

Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, Nebraska USA

August 2015

Adapting program analyses

As we have seen there are several well-developed program analysis frameworks

Researchers have been exploring how to (minimally) adapt them to take probabilistic information into account

- reuse abstract domains and transformers in data flow analysis
- reuse path generation and constraint optimization in sym. exe.

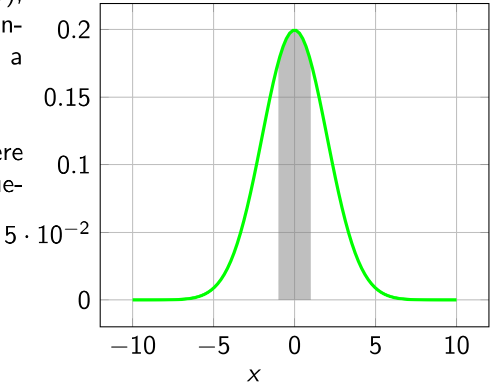
Just enough probability

For simplicity we'll look at just the continuous case.

A probability *density* function (pdf), f, defines the probability that a continuous random variable takes on a specific value.

$Pr[a \leq X \leq b] = \int_a^b f(x)dx$ where f is non-negative and Lebesgue-integrable

For x drawn from $N(0,2)$:
$Pr[-1 \leq x \leq 1]$



Breaking down the PPA literature

Conceptually, quantifying the probability of a set of values requires

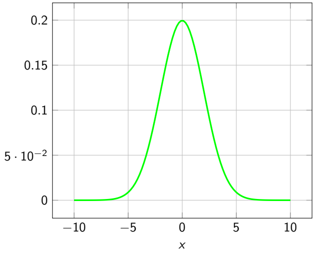
- the number of elements in the set, e.g., $b - a$
- the probability for each element, e.g., f

Probabilistic program analyses can be broken down in several dimensions

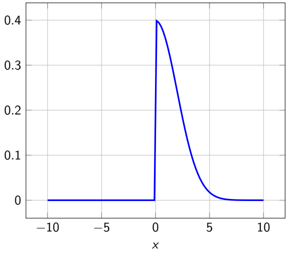
- approximating f vs. approximating $Pr[p(X)]$ for some predicate p
- approximating from above (below) vs. a “close” approximation
- explicit choice probability vs. implicit choice probability

Programs as pdf transformers

The seminal work on probabilistic data flow was Monniaux



```
double abs(int x) {
  if (x<0)
    return -x;
  else
    return x;
}
```



Computed upper bounds on the pdf using discrete approximations.

Abstract domain combined a given underlying domain with a *bounding weight* on f over the domain

Abstract transformers operate on the underlying domain and *shift* weight, when branching, to other parts of the domain

Bounding abstract domains

For an abstract domain $\mathcal{A}$ the probabilistic abstract domain is:

$$\mathcal{P}(\mathcal{A} \times [0,1])$$

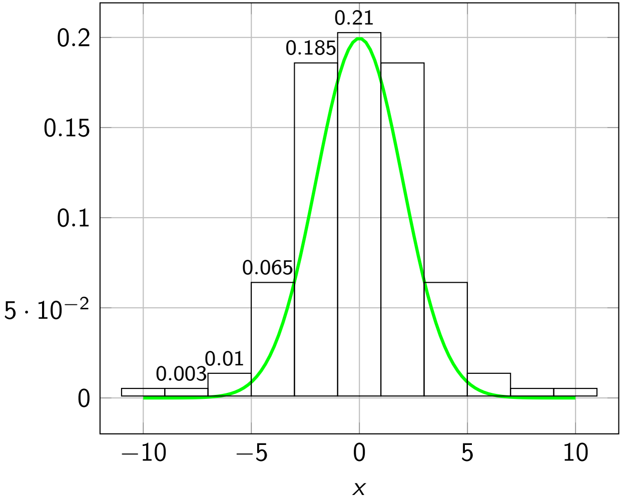
A sequence of pairs (a, w) where w is a *weight* that bounds the probability of elements in $\gamma(a)$

If you want to know the probability of a state given by p at a location with $\langle (a_1, w_1), \ldots \rangle$

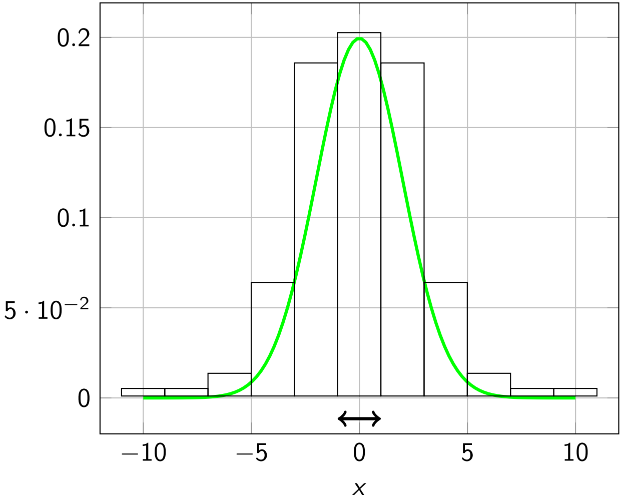
- find all pairs where $p \cap a_i \neq \emptyset$
- record the indices of those pairs in I
- $\forall c \in \gamma(p) : Pr(c) \leq \sum_{i \in I} w_i$

Generic, clean and modular, but can be imprecise

Bounding pdf

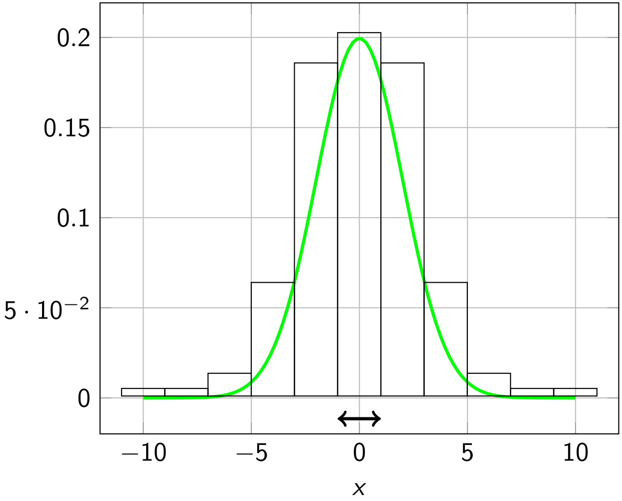


Bounding $Pr([-1, 1])$



How big is the domain?
What is the mass of each domain element?
$Pr([-1, 1]) \leq$

Bounding $Pr([-1,1])$

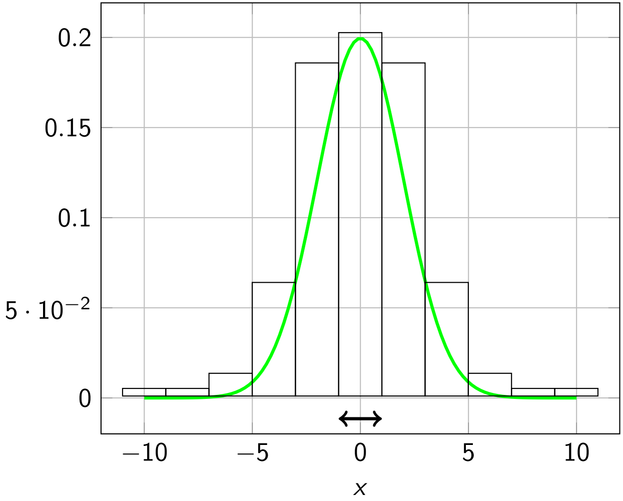


How big is the domain? 2

What is the mass of each domain element?

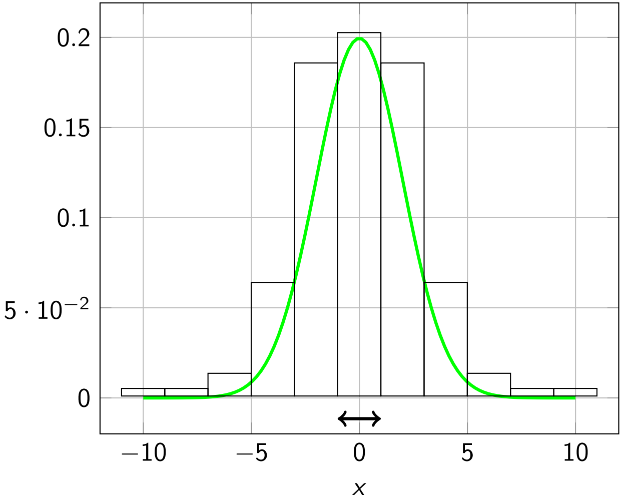
$Pr([-1,1]) \leq$

Bounding $Pr([-1,1])$



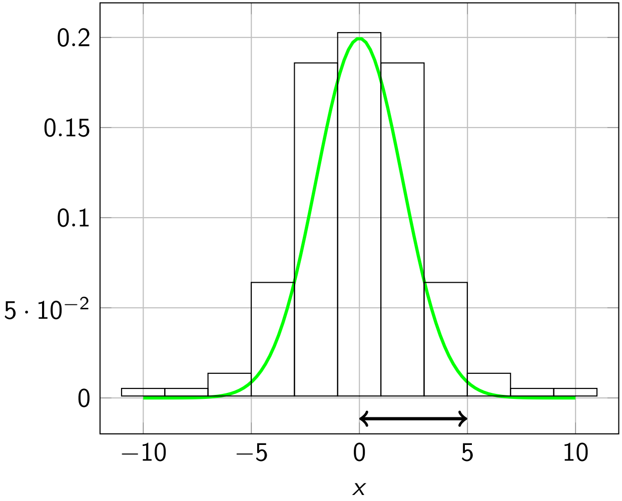
How big is the domain? 2
What is the mass of each domain element? ≤ 0.21
$Pr([-1,1]) \leq$

Bounding $Pr([-1,1])$



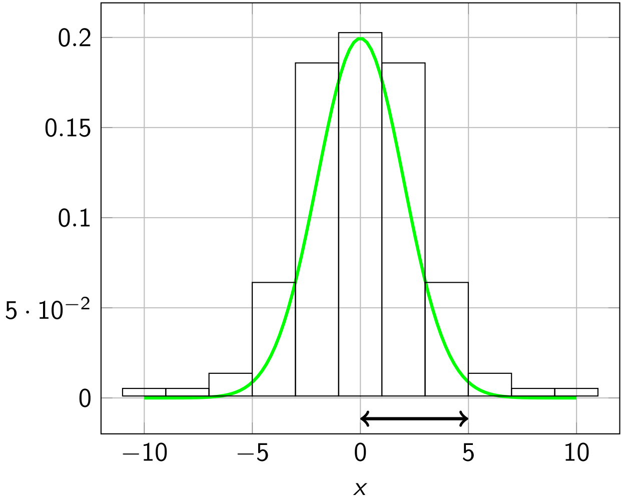
How big is the domain? 2
What is the mass of each domain element? ≤ 0.21
$Pr([-1,1]) \leq 0.42 = 2 * 0.21$

Bounding $Pr([0,5])$



How big is the domain?
What is the mass of each domain element?
$Pr([0,5]) \leq$

Bounding $Pr([0,5])$

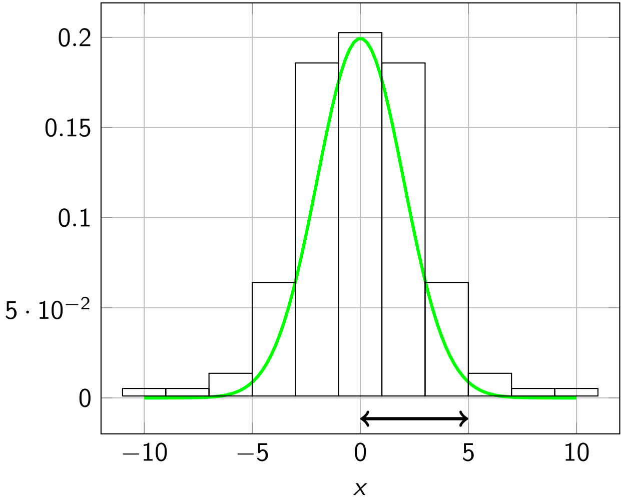


How big is the domain? 5

What is the mass of each domain element?

$Pr([0,5]) \leq$

Bounding $Pr([0,5])$

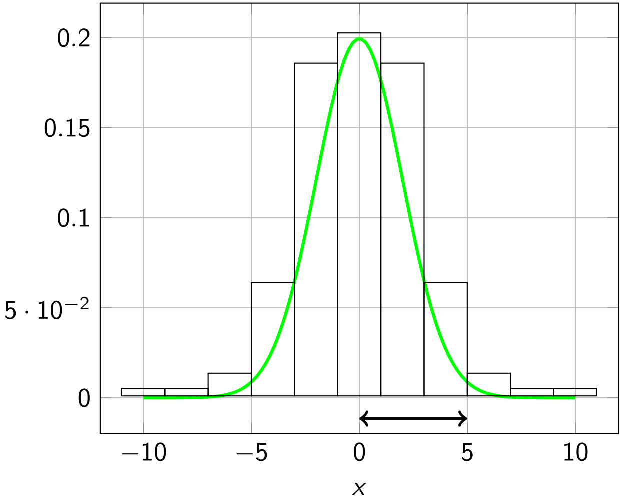


How big is the domain? 5

What is the mass of each domain element? $\leq 0.21, \leq 0.185, \leq 0.065$

$Pr([0,5]) \leq$

Bounding $Pr([0,5])$



How big is the domain? 5
What is the mass of each domain element? $\leq 0.21, \leq 0.185, \leq 0.065$
$Pr([0,5]) \leq 0.71 = 0.21 + 2 * 0.185 + 2 * 0.065$

Breaking down the PPA literature

Probabilistic program analyses can be broken down in several dimensions

- approximating f vs. approximating $Pr[p(X)]$ for some predicate p
- approximating from above (below) vs. a "close" approximation
- explicit choice probability vs. implicit choice probability

More recent work develops Monniaux's ideas further ...

Mardziel et al. (2013) develop a polyhedra domain that tracks upper and lower bounds

Adje et al. (2014) develop an affine function form that tracks bounds

Breaking down the PPA literature

Probabilistic program analyses can be broken down in several dimensions

- approximating f vs. approximating $Pr[p(X)]$ for some predicate p
- approximating from above (below) vs. a "close" approximation
- explicit choice probability vs. implicit choice probability

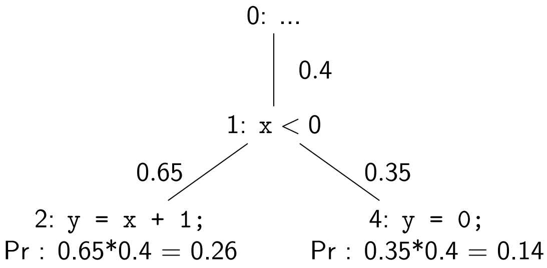
di Pierro, Wicklicky, and Hankin in a series of papers (2002-2013) develop data flow analyses to compute least-squares error approximation

Smith (2008) restricts the supported distributions to allow for precise estimation

Chakarov and Sankaranarayanan (2013-2014) compute *expectation invariants* – bounds on the long run expectation of a program expression

Explicit branch probabilities

A number of researchers have explored models where they assume that the probabilities on branches are given



This makes sense in some cases, e.g., `x = bernoulli(0.5);`

Breaking down the PPA literature

Probabilistic program analyses can be broken down in several dimensions

- approximating f vs. approximating $Pr[p(X)]$ for some predicate p
- approximating from above (below) vs. a "close" approximation
- explicit choice probability vs. implicit choice probability

A body of work makes the simplifying assumption that branch probabilities are given (this includes all work on probabilistic model checking)

Ramalingam (1996) generalized Kildall's framework to accumulate path probabilities

This is equivalent to PRISM's support for DTMCs — see Kwiatkowska et al (2011)

Wachter and Zhang (2010), Esparza and Gaiser (2011), and Kwiatkowska et al (2011) apply predicate abstraction to data to scale probabilistic model checking to software

Breaking down the PPA literature

Probabilistic program analyses can be broken down in several dimensions

- approximating f vs. approximating $Pr[p(X)]$ for some predicate p
- approximating from above (below) vs. a "close" approximation
- explicit choice probability vs. implicit choice probability

Sankaranarayanan et al (2013) estimate the property probabilities using a path selection approach to drive symbolic execution

We developed probabilistic symbolic execution in a series of papers (2012-2015) that, novely, computes the conditional choice probabilities for branches along paths

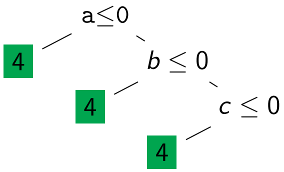
A simple example ...

```
int classify(int a, int b, int c) {
  if (a<=0 || b<=0 || c<=0) return 4;
  int type=0;
  if (a==b) type+=1;
  if (a==c) type+=2;
  if (b==c) type+=3;
  if (type==0) {
    if (a+b<=c || b+c<=a || a+c>=b) type=4;
    else type=1;
    return type;
  }
  if (type>3) type=3;
  else if (type==1 && a+b>c) type=2;
  else if (type==2 && a+c>b) type=2;
  else if (type==3 && b+c>a) type=2;
  else type=4;
  return type;
}
```

A simple example ...

```
int classify(int a, int b, int c) {
  if (a<=0 || b<=0 || c<=0) return 4;
  int type=0;
  if (a==b) type+=1;
  if (a==c) type+=2;
  if (b==c) type+=3;
  if (type==0) {
    if (a+b<=c || b+c<=a || a+c>=b) type=4;
    else type=1;
    return type;
  }
  if (type>3) type=3;
  else if (type==1 && a+b>c) type=2;
  else if (type==2 && a+c>b) type=2;
  else if (type==3 && b+c>a) type=2;
  else type=4;
  return type;
}
```

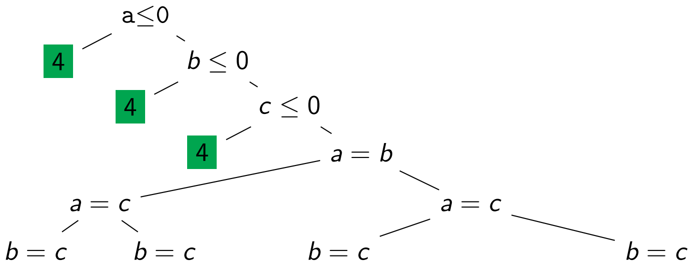
Symbolic execution tree



A simple example ...

```
int classify(int a, int b, int c) {
  if (a<=0 || b<=0 || c<=0) return 4;
  int type=0;
  if (a==b) type+=1;
  if (a==c) type+=2;
  if (b==c) type+=3;
  if (type==0) {
    if (a+b<=c || b+c<=a || a+c>=b) type=4;
    else type=1;
    return type;
  }
  if (type>3) type=3;
  else if (type==1 && a+b>c) type=2;
  else if (type==2 && a+c>b) type=2;
  else if (type==3 && b+c>a) type=2;
  else type=4;
  return type;
}
```

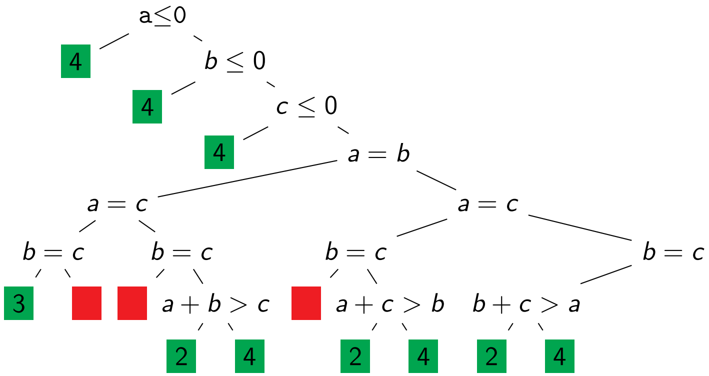
Symbolic execution tree



A simple example ...

```
int classify(int a, int b, int c) {
  if (a<=0 || b<=0 || c<=0) return 4;
  int type=0;
  if (a==b) type+=1;
  if (a==c) type+=2;
  if (b==c) type+=3;
  if (type==0) {
    if (a+b<=c || b+c<=a || a+c>=b) type=4;
    else type=1;
    return type;
  }
  if (type>3) type=3;
  else if (type==1 && a+b>c) type=2;
  else if (type==2 && a+c>b) type=2;
  else if (type==3 && b+c>a) type=2;
  else type=4;
  return type;
}
```

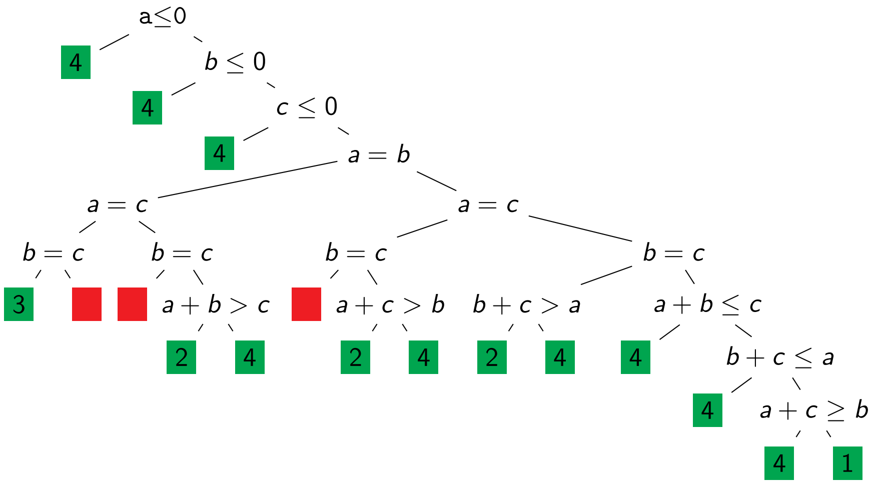
Symbolic execution tree



A simple example ...

```
int classify(int a, int b, int c) {
  if (a<=0 || b<=0 || c<=0) return 4;
  int type=0;
  if (a==b) type+=1;
  if (a==c) type+=2;
  if (b==c) type+=3;
  if (type==0) {
    if (a+b<=c || b+c<=a || a+c>=b) type=4;
    else type=1;
    return type;
  }
  if (type>3) type=3;
  else if (type==1 && a+b>c) type=2;
  else if (type==2 && a+c>b) type=2;
  else if (type==3 && b+c>a) type=2;
  else type=4;
  return type;
}
```

Symbolic execution tree



Some observations

- There are 14 distinct paths (Green): from 1 to 9 branches
- 1 path returns “scalene” (1); 3 return “isoscelese” (2); and 1 returns “equilateral” (3)
- 3 paths are pruned because constraints are unsat (Red)
- Interesting symmetries are involved in the “isoscelese” and unsat cases

Adding probabilities

The key insight here is to shift from using SAT queries to *counting* queries (#)

- cost ranges from fast, e.g., $\#([a, b])$, to exponential
- cost-effective # procedures may not be available
- use statistical estimators when necessary

Probability estimates can be very precise for state space that is analyzed

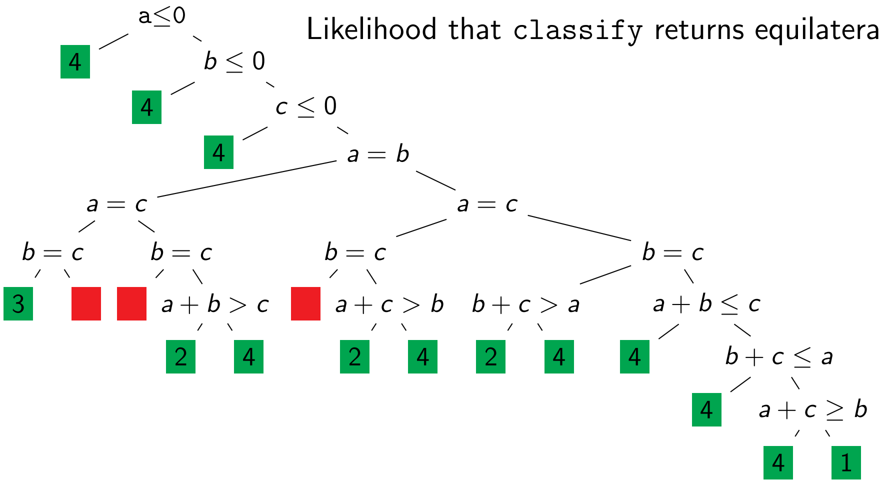
- computes an underapproximation of state probabilities
- unanalyzed state space can be quantified

Numerous layers of optimization required to make it efficient

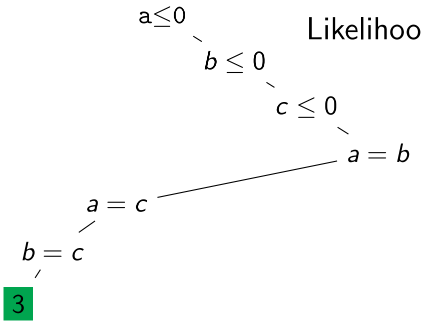
- due to optimizations initial versions ran faster then classic symbolic execution (backpatched compatible optimizations)
- still limited to programs with 10s of thousands of SLOC

Assume `ints` are drawn uniformly from $[-1000, 1000]$

Likelihood that `classify` returns equilateral?

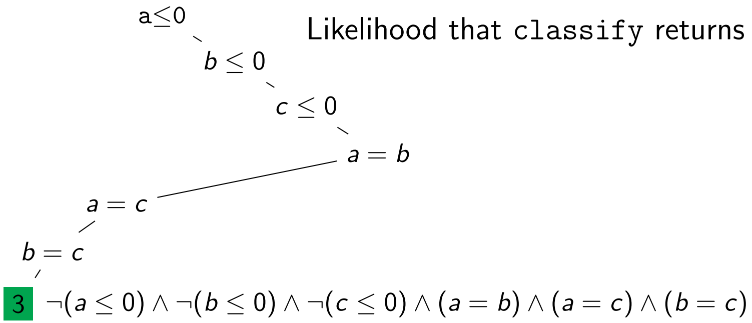


Assume `ints` are drawn uniformly from $[-1000, 1000]$



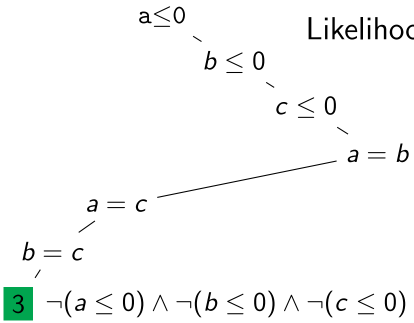
Likelihood that `classify` returns equilateral?

Assume `ints` are drawn uniformly from $[-1000, 1000]$


a≤0
$b \leq 0$
$c \leq 0$
$a = b$
$a = c$
$b = c$
3
$\neg(a \leq 0) \wedge \neg(b \leq 0) \wedge \neg(c \leq 0) \wedge (a = b) \wedge (a = c) \wedge (b = c)$

Likelihood that `classify` returns equilateral?

Assume `ints` are drawn uniformly from $[-1000, 1000]$

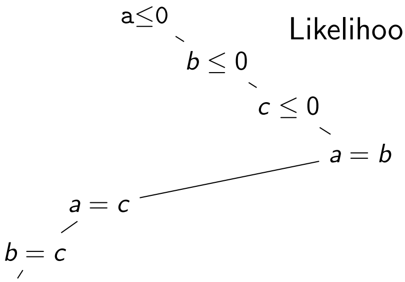


Likelihood that `classify` returns equilateral?

$\neg(a \le 0) \wedge \neg(b \le 0) \wedge \neg(c \le 0) \wedge (a = b) \wedge (a = c) \wedge (b = c)$

How many inputs satisfy this path condition?

Assume `ints` are drawn uniformly from $[-1000, 1000]$



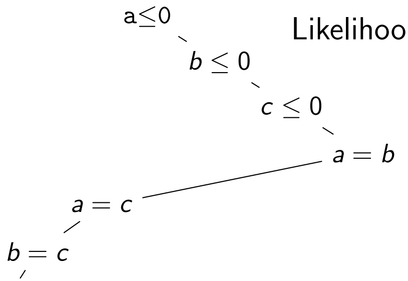
Likelihood that `classify` returns equilateral?

3 $\neg(a \leq 0) \wedge \neg(b \leq 0) \wedge \neg(c \leq 0) \wedge (a = b) \wedge (a = c) \wedge (b = c)$

How many inputs satisfy this path condition?

1000

Assume `ints` are drawn uniformly from $[-1000, 1000]$



3 $\neg(a \leq 0) \wedge \neg(b \leq 0) \wedge \neg(c \leq 0) \wedge (a = b) \wedge (a = c) \wedge (b = c)$

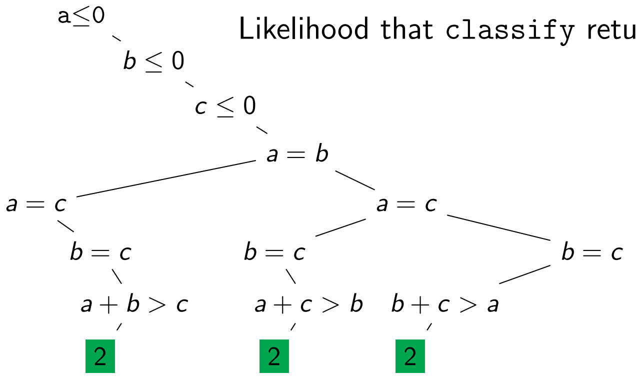
Likelihood that `classify` returns equilateral?

$1.25e^{-7}$

How many inputs satisfy this path condition?

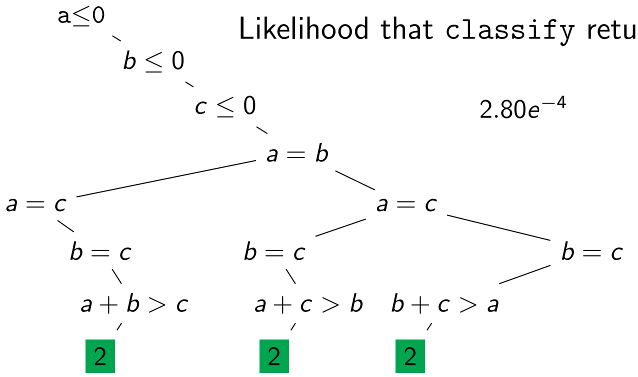
1000

Assume `int`s are drawn uniformly from $[-1000, 1000]$



Likelihood that `classify` returns isosceles?

Assume `ints` are drawn uniformly from $[-1000, 1000]$



Likelihood that `classify` returns isosceles?

$2.80e^{-4}$

Some observations

- You may be able to calculate these probabilities because you know about triangles.

Some observations

- You may be able to calculate these probabilities because you know about triangles.
- We want to calculate the probability that a program's execution ...
    - returns a value, reaches a statement, ...

Some observations

- You may be able to calculate these probabilities because you know about triangles.
- We want to calculate the probability that a program's execution ...
  - returns a value, reaches a statement, ...
- Adapting symbolic execution to perform these computations involves ...
  - calculating the paths of interest
  - calculating the probability of taking those paths
  - combining those probabilities appropriately

Probabilistic symbolic execution algorithm

Algorithm 1 $probSymEx(l, m, pc, Pr_{pc})$

```
if $stoppingPath(pc)$ then
    return $pc$
end if
while $\neg branch(l)$ do
    $m \leftarrow op(l)(m)$
    $l \leftarrow succ(l)$
end while
$c \leftarrow cond(l)(m)$
$pc' \leftarrow slice(pc, c)$
$Pr_c \leftarrow prob(pc' \wedge c)/prob(pc')$
if $Pr_c > 0$ then
    $probSymEx(succ_t(l), pc \wedge c, m, Pr_{pc} * Pr_c)$
end if
if $Pr_c < 1$ then
    $probSymEx(succ_f(l), pc \wedge \neg c, m, Pr_{pc} * (1 - Pr_c))$
end if
```

Key algorithmic features (Geldenhuys et al ISSTA'12)

Slicing the path condition, i.e., $slice(pc, c)$

- reduces formula size which reduces cost of $prob(\cdot)$
- exposes opportunities for reusing computation in $prob(\cdot)$

Key algorithmic features (Geldenhuys et al ISSTA'12)

Slicing the path condition, i.e., $\mathit{slice}(pc, c)$

- reduces formula size which reduces cost of $\mathit{prob}(\cdot)$
- exposes opportunities for reusing computation in $\mathit{prob}(\cdot)$

Calculating the conditional probability Pr_c of c

- requires model counting of path condition
- determines satisfiability of branches, i.e., $Pr_c > 0 \implies SAT(pc \wedge c)$
- allows inference of off-branch probability, i.e., $Pr_{pc} * (1 - Pr_c)$
- depth-first nature of symbolic execution ensures that $\mathit{prob}(pc')$ will be reused
- slicing assures independence in computing Pr_c, i.e., $pc - pc'$ factored out

Key algorithmic features (Geldenhuys et al ISSTA'12)

Slicing the path condition, i.e., $\mathit{slice}(pc, c)$

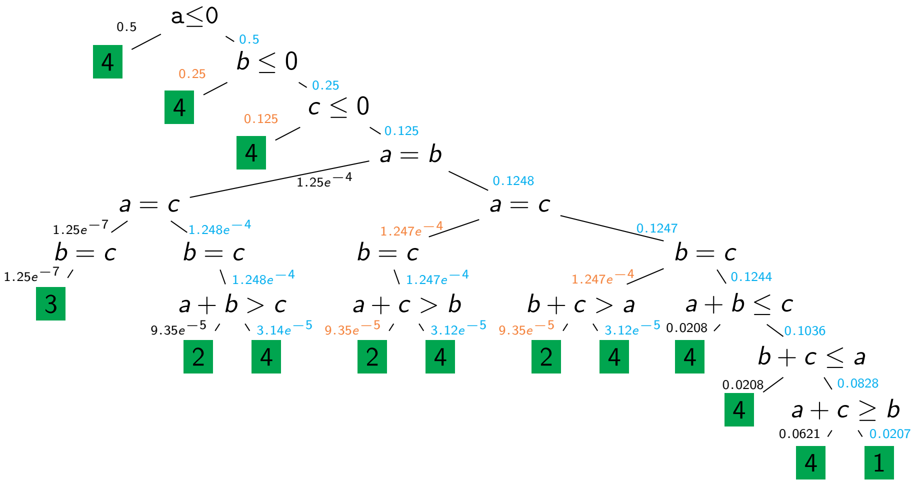
- reduces formula size which reduces cost of $\mathit{prob}(\cdot)$
- exposes opportunities for reusing computation in $\mathit{prob}(\cdot)$

Calculating the conditional probability Pr_c of c

- requires model counting of path condition
- determines satisfiability of branches, i.e., $Pr_c > 0 \implies SAT(pc \wedge c)$
- allows inference of off-branch probability, i.e., $Pr_{pc} * (1 - Pr_c)$
- depth-first nature of symbolic execution ensures that $\mathit{prob}(pc')$ will be reused
- slicing assures independence in computing Pr_c, i.e., $pc - pc'$ factored out

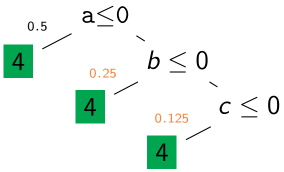
This allows the algorithm to compute path probabilities cost-effectively.

Probabilistic symbolic execution tree

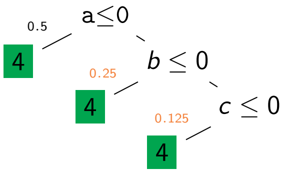


29 branches: 8 counting queries, 6 reused, 15 inferred

Probabilistic symbolic execution tree

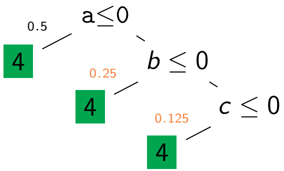


Probabilistic symbolic execution tree



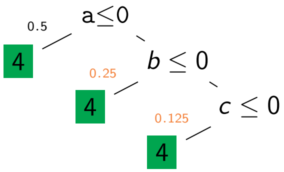
$$\mathit{slice}(\mathit{true}, a \leq 0) = \mathit{true}$$

Probabilistic symbolic execution tree



$$slice(true, a \leq 0) = true$$
$$Pr_c = prob(a \leq 0)/prob(true)$$

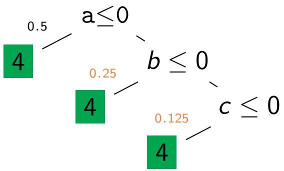
Probabilistic symbolic execution tree



$$slice(true, a \leq 0) = true$$
$$Pr_c = prob(a \leq 0)/prob(true)$$

$$slice(a \leq 0 \wedge b \leq 0, c \leq 0) = true$$

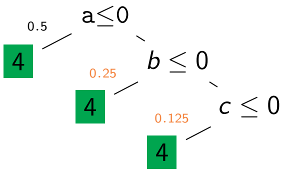
Probabilistic symbolic execution tree



$$slice(true, a \leq 0) = true$$
$$Pr_c = prob(a \leq 0)/prob(true)$$

$$slice(a \leq 0 \wedge b \leq 0, c \leq 0) = true$$
$$Pr_c = prob(c \leq 0)/prob(true)$$

Probabilistic symbolic execution tree



$$slice(true, a \leq 0) = true$$
$$Pr_c = prob(a \leq 0)/prob(true)$$

$$slice(a \leq 0 \land b \leq 0, c \leq 0) = true$$
$$Pr_c = prob(c \leq 0)/prob(true)$$

Normalization of constraints, e.g., $a \mapsto v_1$, $c \mapsto v_1$, enables reuse in calculating Pr_c

Calculating *prob*$(\cdot)$

Linear integer arithmetic (LIA) constraints can be counted using LattE

- computes the number of *lattice* points in a convex polytope;
- constraints encoded as system of inequalities, $Ax \leq B$;
- does not support disjunction or disequality constraints, i.e., $x \neq c$

Calculating $prob(\cdot)$

Linear integer arithmetic (LIA) constraints can be counted using LattE

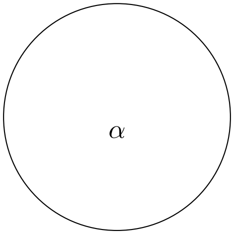
- computes the number of *lattice* points in a convex polytope;
- constraints encoded as system of inequalities, $Ax \leq B$;
- does not support disjunction or disequality constraints, i.e., $x \neq c$

Calculation relies on "counting" the number of solutions of a set of related constraints using LattE and combining the results.

- $count = count_{\wedge}(\bigwedge_{ineqSet}) - count_{\vee}(\bigvee_{exSet})$
- `return` $count / \prod_{v \in vars} dom(v)$

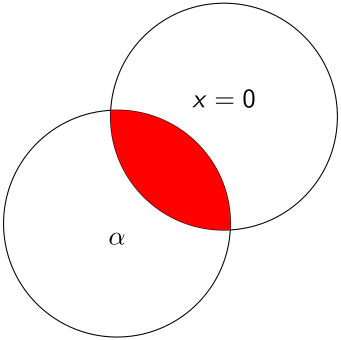
$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly

$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly



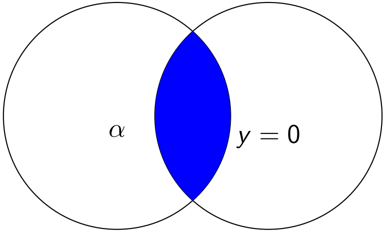
Count the solutions to α

$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly



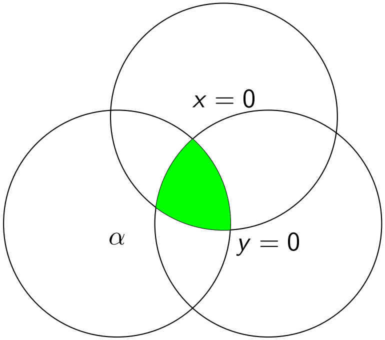
Remove the count of solutions to $\alpha \wedge (x = 0)$

$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly



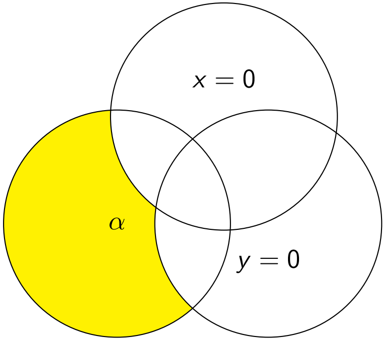
Remove the count of solutions to $\alpha \wedge (y = 0)$

$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly



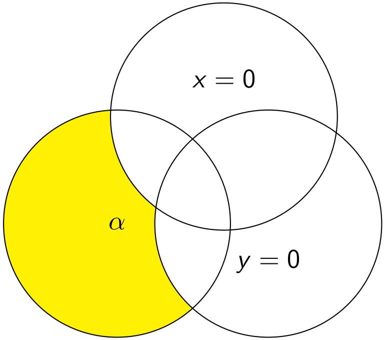
Add back the count of solutions to $\alpha \wedge (x = 0) \wedge (y = 0)$

$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly



This results in the count of $\alpha \wedge (x \neq 0) \wedge (y \neq 0)$

$\alpha \wedge (x \neq 0) \wedge (y \neq 0)$ cannot be expressed directly



This results in the count of $\alpha \wedge (x \neq 0) \wedge (y \neq 0)$

Complexity is exponential in number of disequality constraints

Optimizing $count_{\wedge}(\cdot)$

experience with LattE revealed that its execution time

- is not dependent on the size of variable domains;
- is highly dependent on the number of variables (dimension of the polytope);
- is highly dependent on the number of constraints (faces of the polytope);

Optimizing $count_{\wedge}(\cdot)$

experience with LattE revealed that its execution time

- is not dependent on the size of variable domains;
- is highly dependent on the number of variables (dimension of the polytope);
- is highly dependent on the number of constraints (faces of the polytope);

Experience with sliced PCs revealed that

- a significant portion of the PC, and many variables, can be eliminated;
- sliced PCs, if normalized, recur throughout the symbolic execution tree

Optimizing $count_{\wedge}(\cdot)$

experience with LattE revealed that its execution time

- is not dependent on the size of variable domains;
- is highly dependent on the number of variables (dimension of the polytope);
- is highly dependent on the number of constraints (faces of the polytope);

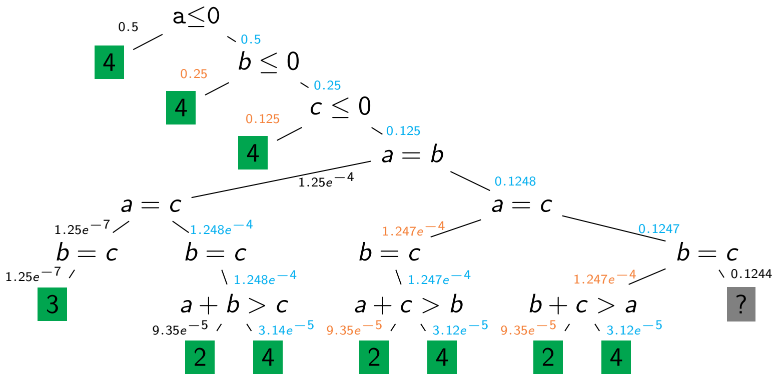
Experience with sliced PCs revealed that

- a significant portion of the PC, and many variables, can be eliminated;
- sliced PCs, if normalized, recur throughout the symbolic execution tree

Implementation of $count_{\wedge}(\cdot)$: Visser et al (FSE'12)

- normalizes the inequality system
- caches the counts computed for each system; and
- checks the cache before invoking LattE.

Partial Probabilistic symbolic execution



Probabilistic Symbolic Execution for Concurrency

View program's behavior as a tree-structured Markov Decision Process

- symbolic execution tree with non-deterministic choice nodes

Probabilistic Symbolic Execution for Concurrency

View program's behavior as a tree-structured Markov Decision Process

- symbolic execution tree with non-deterministic choice nodes

Randomly sample paths, but unlike Monte Carlo methods

- each sampled path, p, contributes mass proportional to $count(PC_p)$
- a sampled path is biased against being revisited

Probabilistic Symbolic Execution for Concurrency

View program's behavior as a tree-structured Markov Decision Process

- symbolic execution tree with non-deterministic choice nodes

Randomly sample paths, but unlike Monte Carlo methods

- each sampled path, p, contributes mass proportional to $count(PC_p)$
- a sampled path is biased against being revisited

We compute the maximum probability of an event (e.g., `assert`)

- resolve non-deterministic choice to use max probability of child
- calculation is bottom up like value-iteration

Probabilistic Symbolic Execution for Concurrency

View program's behavior as a tree-structured Markov Decision Process

- symbolic execution tree with non-deterministic choice nodes

Randomly sample paths, but unlike Monte Carlo methods

- each sampled path, p, contributes mass proportional to $count(PC_p)$
- a sampled path is biased against being revisited

We compute the maximum probability of an event (e.g., `assert`)

- resolve non-deterministic choice to use max probability of child
- calculation is bottom up like value-iteration

MDP support works for any source of non-determinism, e.g., abstract post, unknown input values

Probabilistic Symbolic Execution for Concurrency

View program's behavior as a tree-structured Markov Decision Process

- symbolic execution tree with non-deterministic choice nodes

Randomly sample paths, but unlike Monte Carlo methods

- each sampled path, p, contributes mass proportional to $count(PC_p)$
- a sampled path is biased against being revisited

We compute the maximum probability of an event (e.g., `assert`)

- resolve non-deterministic choice to use max probability of child
- calculation is bottom up like value-iteration

MDP support works for any source of non-determinism, e.g., abstract post, unknown input values

Fastest and most precise method to date by exploiting tree structure

Probabilistic symbolic execution algorithm

The briefing paper describes a broader family of such algorithms

Alg. 2 `pse`(l, m, pc)

```
repeat
    $p \leftarrow$ symsample$(l_0, m_0, true)$
    $processPath(p)$
until $stoppingSearch(p)$
```

Alg. 3 `symsample`(l, m, pc)

```
if $stoppingPath(pc)$ then
    return $pc$
end if
while $\neg branch(l)$ do
    $m \leftarrow op(l)(m)$
    $l \leftarrow succ(l)$
end while
$c \leftarrow cond(l)(m)$
if $selectBranch(c, pc)$ then
    return
    symsample$(succ_t(l), m, pc \wedge c)$
else
    return
    symsample$(succ_f(l), m, pc \wedge \neg c)$
end if
```

Solution space quantification

In the 1990s “propositional” SAT solver technology advanced

- increased in performance by orders of magnitude
- was subsequently applied to many many problems

Solution space quantification

In the 1990s “propositional” SAT solver technology advanced

- increased in performance by orders of magnitude
- was subsequently applied to many many problems

In the 2000s satisfiability modulo-theories (SMT) solver technology advanced

- increased in performance by orders of magnitude
- was subsequently applied to many many problems
- subsumes propositional SAT, so jump straight into SMT
- tons of tools, e.g., Z3, Yices, CVC4, ...

Solution space quantification

In the 1990s “propositional” SAT solver technology advanced

- increased in performance by orders of magnitude
- was subsequently applied to many many problems

In the 2000s satisfiability modulo-theories (SMT) solver technology advanced

- increased in performance by orders of magnitude
- was subsequently applied to many many problems
- subsumes propositional SAT, so jump straight into SMT
- tons of tools, e.g., Z3, Yices, CVC4, ...

The 2010s is the decade of model counting

Get on board early and leverage it in your research!

Exact Counting Methods

Believe it or not you can count the **exact** number of solutions of very complex systems of constraints.

Exact Counting Methods

Believe it or not you can count the **exact** number of solutions of very complex systems of constraints.

Linear constraints over integer program variables

- every constraint introduces a cutting plane
- together these form a convex polyhedra
- tools exist to exactly count the number of integer values in the interior of the polyhedra, e.g., Latte, barvinok, ...
- scales to 10s of dimensions
- efficiency is **not** dependent on the size of the domain, i.e., you can solve constraints over $[MININT, MAXINT]$

Exact Counting Methods

Believe it or not you can count the **exact** number of solutions of very complex systems of constraints.

Linear constraints over integer program variables

- every constraint introduces a cutting plane
- together these form a convex polyhedra
- tools exist to exactly count the number of integer values in the interior of the polyhedra, e.g., Latte, barvinok, ...
- scales to 10s of dimensions
- efficiency is **not** dependent on the size of the domain, i.e., you can solve constraints over $[MININT, MAXINT]$

Techniques for strings, data structures, and more

- see the work of Luu et al (PLDI'14), Fredrickson et al (LICS'14), Freemont et al (SMT'14), Filieri et al (SPIN'15), Aydin et al (CAV'15)

Approximate Counting Methods

Building on the significant literature in Monte Carlo estimation from Statistics, Physics, AI, ...

Approximate Counting Methods

Building on the significant literature in Monte Carlo estimation from Statistics, Physics, AI, ...

Sampling techniques serve to estimate the parameters of a distribution

- rich collection of techniques for handling uniform distributions
- number of samples needed to achieve a level of accuracy can be computed
- a wide range of distributions can be reduced to uniform through inverse transform sampling
- sample uniform distribution then use hill climbing on the monotone CDF^{-1}

Approximate Counting Methods

Building on the significant literature in Monte Carlo estimation from Statistics, Physics, AI, ...

Sampling techniques serve to estimate the parameters of a distribution

- rich collection of techniques for handling uniform distributions
- number of samples needed to achieve a level of accuracy can be computed
- a wide range of distributions can be reduced to uniform through inverse transform sampling
- sample uniform distribution then use hill climbing on the monotone CDF^{-1}

Techniques that exploit the structure of programs are starting to appear

- see the work of Borges et al (PLDI'14), Borges et al (ESEC/FSE'15), ...

Approximate Counting Methods

Building on the significant literature in Monte Carlo estimation from Statistics, Physics, AI, ...

Sampling techniques serve to estimate the parameters of a distribution

- rich collection of techniques for handling uniform distributions
- number of samples needed to achieve a level of accuracy can be computed
- a wide range of distributions can be reduced to uniform through inverse transform sampling
- sample uniform distribution then use hill climbing on the monotone CDF^{-1}

Techniques that exploit the structure of programs are starting to appear

- see the work of Borges et al (PLDI'14), Borges et al (ESEC/FSE'15), ...

Lots of room for creative blending of techniques ...

- model counting + search + numerical optimization : Dingel at al (FSE'14)