Principles of Probabilistic Programming

Joost-Pieter Katoen

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

Marktoberdorf Summer School 2017

Overview

nature Perspective

"There are several reasons why probabilistic programming could prove to be revolutionary for machine intelligence and scientific modelling."[1]

REVIEW

doi:10.1038/nature14541

Probabilistic machine learning and artificial intelligence

Zoubin Ghahramani[1]

[1] Zoubin Ghahramani leads the Cambridge Machine Learning Group, and holds positions at CMU, UCL, and the Alan Turing Institute.

Probabilistic programs

What?

They are programs with random assignments and conditioning

Why?

- Random assignments: to describe randomised algorithms
- Conditioning: to describe stochastic decision making

Randomised algorithms

- What? Some decisions are based on coin flips

- Why?
  - Their conceptual simplicity
  - Their speed
    - mostly faster than their deterministic counterpart
    - no particular input elicits worst-case behaviour
  - Their existence
    - many solve problems that have no deterministic solution

- Types

1. Las Vegas: always produces correct results, random runtime
2. Monte Carlo: may produce wrong results, deterministic runtime

Sorting by flipping coins

Quicksort:

```
QS(A) =
  if |A| <= 1 { return A; }
  i := ceil(|A|/2);
  A< := {a in A | a < A[i]};
  A> := {a in A | a > A[i]};
  return QS(A<) ++ A[i] ++ QS(A>)
```

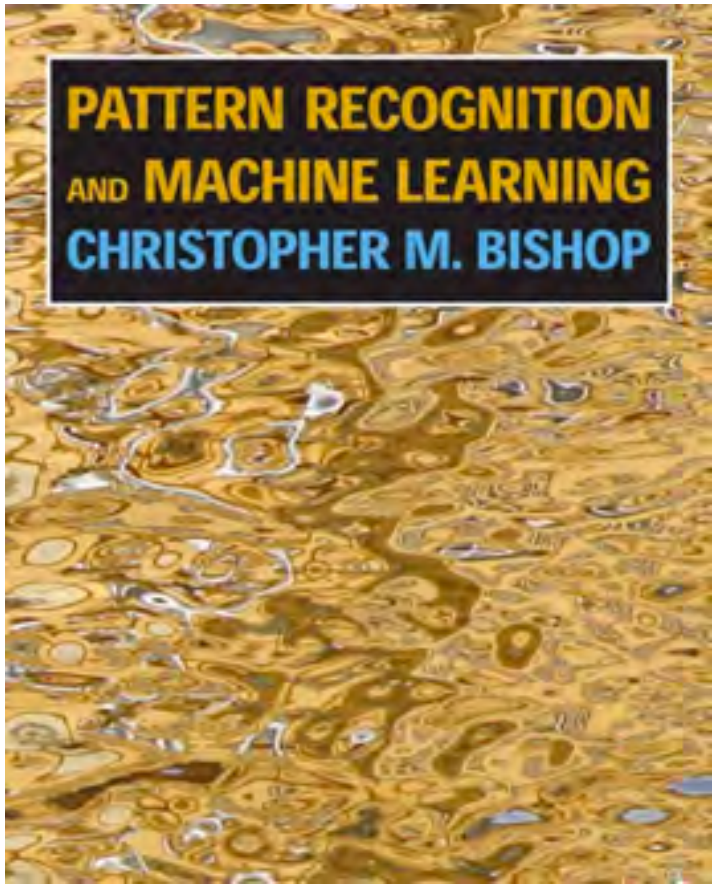
Worst case complexity:
$O(N^2)$ comparisons

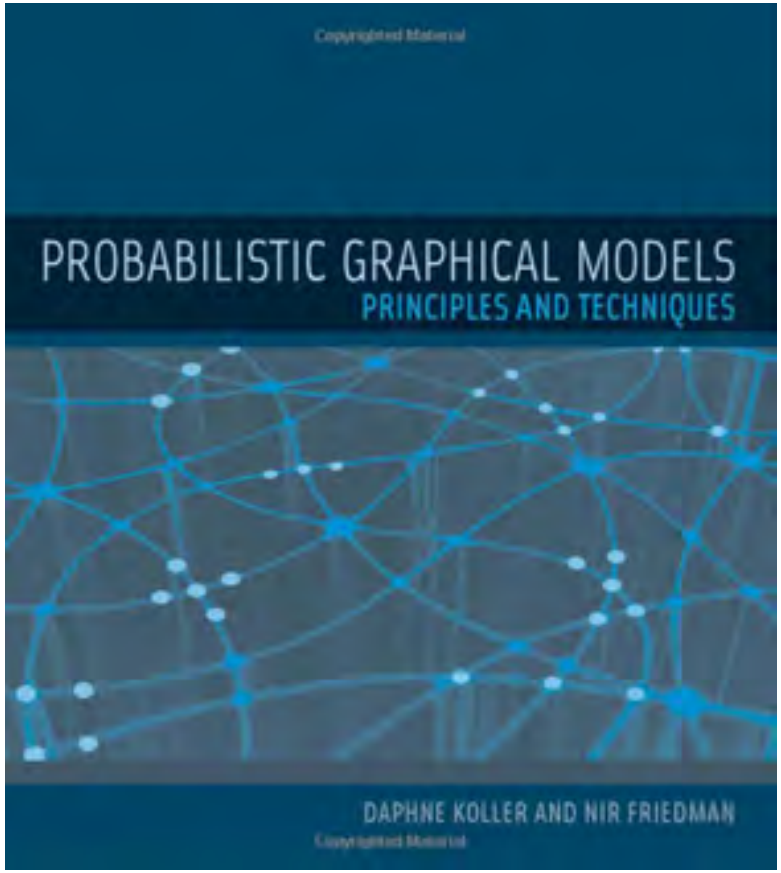
Randomised Quicksort:

```
rQS(A) =
  if |A| <= 1 { return A; }
  i := Unif[1...|A|];
  A< := {a in A | a < A[i]};
  A> := {a in A | a > A[i]};
  return rQS(A<) ++ A[i] ++ rQS(A>)
```

Worst case complexity:
$O(N \log N)$ expected comparisons

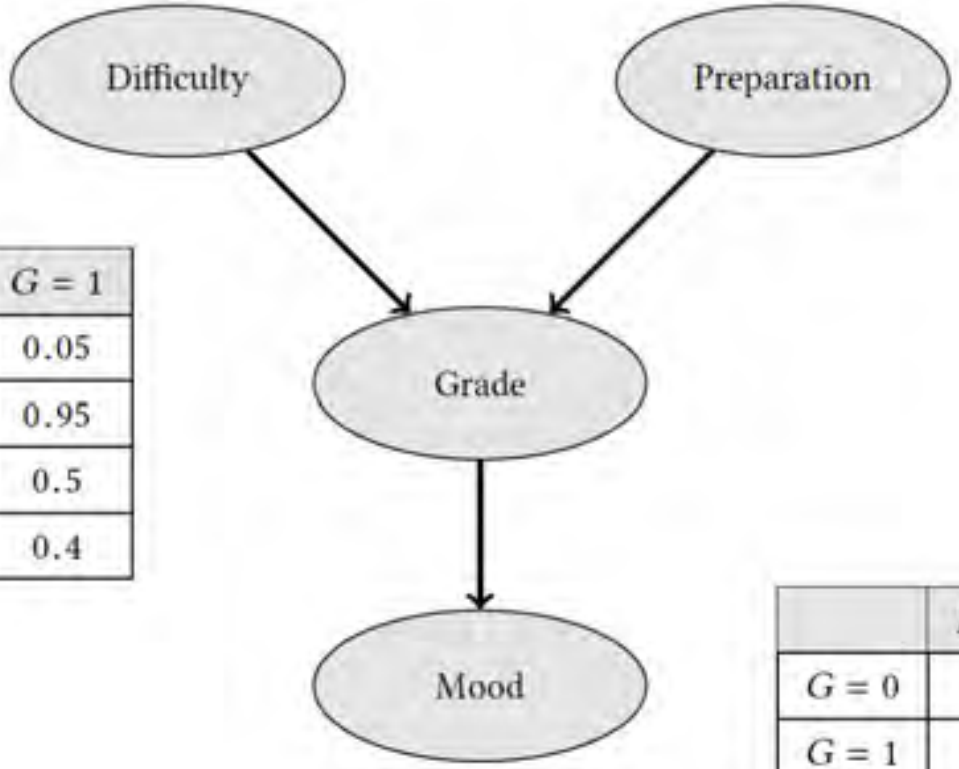
Probabilistic graphical models





Student's mood after an exam

$D = 0$	$D = 1$
0.6	0.4



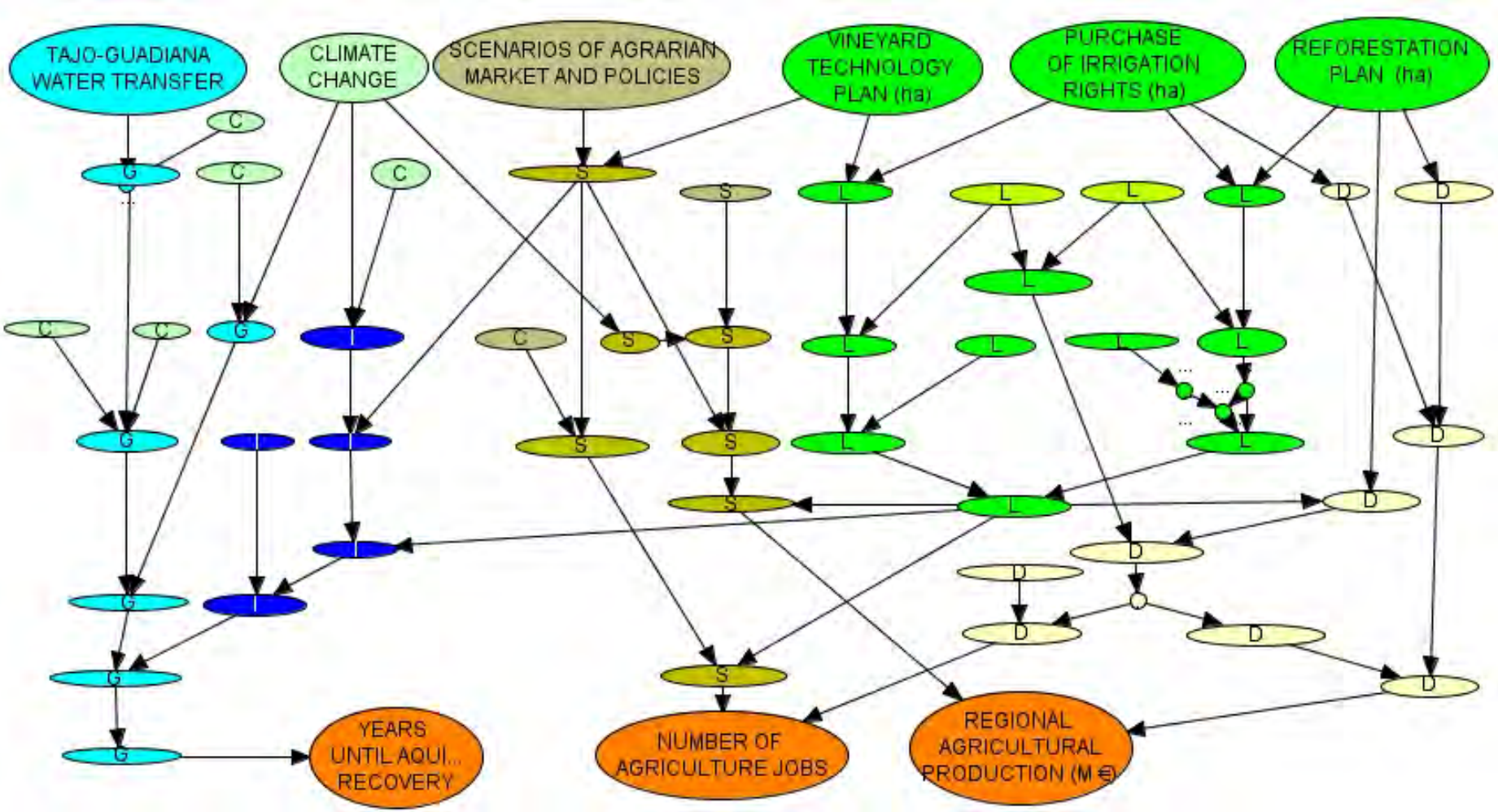
$P = 0$	$P = 1$
0.7	0.3

	$G = 0$	$G = 1$
$D = 0, P = 0$	0.95	0.05
$D = 1, P = 1$	0.05	0.95
$D = 0, P = 1$	0.5	0.5
$D = 1, P = 0$	0.6	0.4

	$M = 0$	$M = 1$
$G = 0$	0.9	0.1
$G = 1$	0.3	0.7

How likely does a well-prepared student ends up with a bad mood?

Ecology



When to purchase irrigation rights or impose pumping restrictions?

How Statisticians Found Air France Flight 447 Two Years After It Crashed Into Atlantic

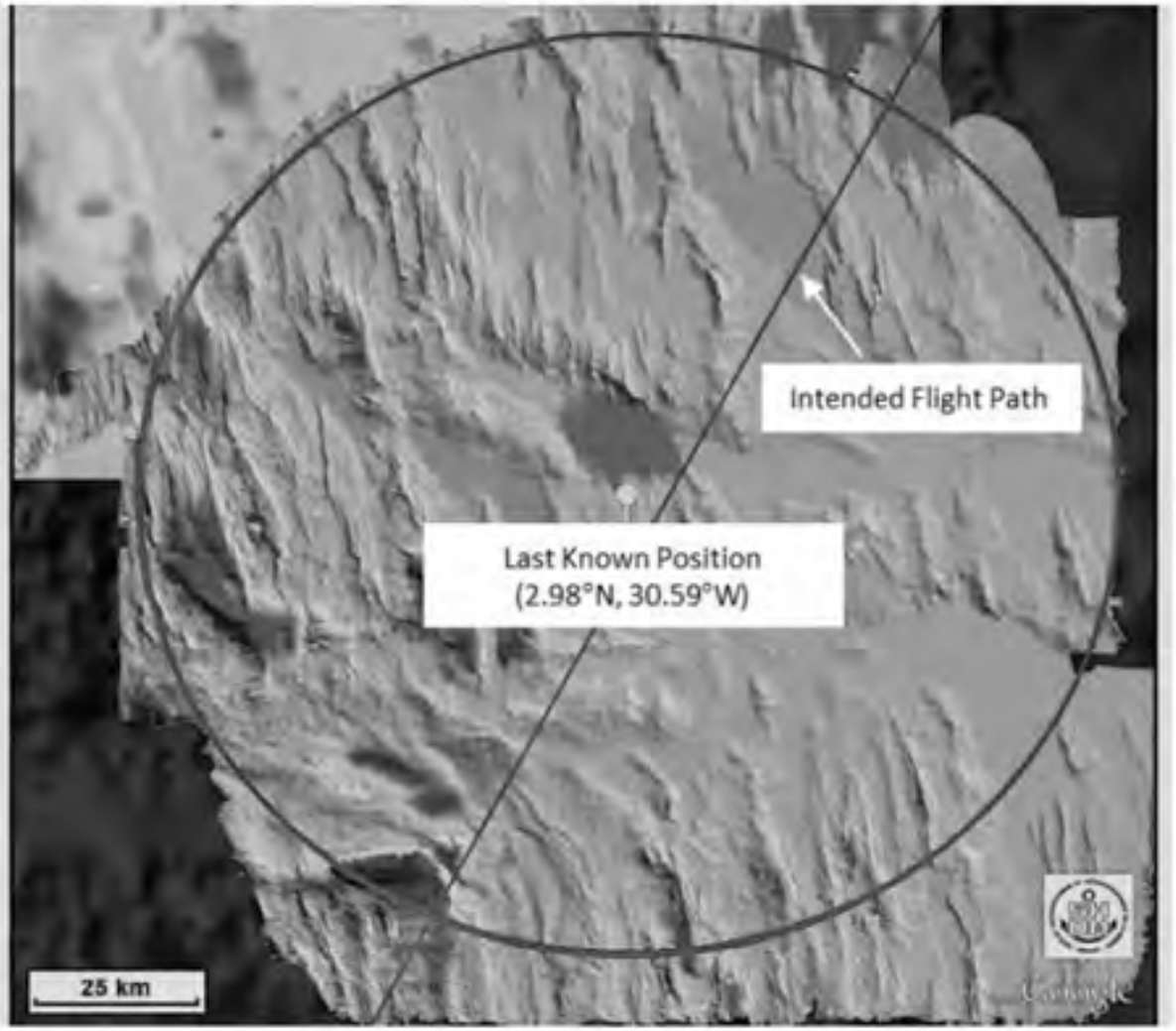
[MIT Technology Review, May 2014]

Air France flight AF-447

Airbus A-330 flight AF-447

June 1, 2009

AF447: Last position



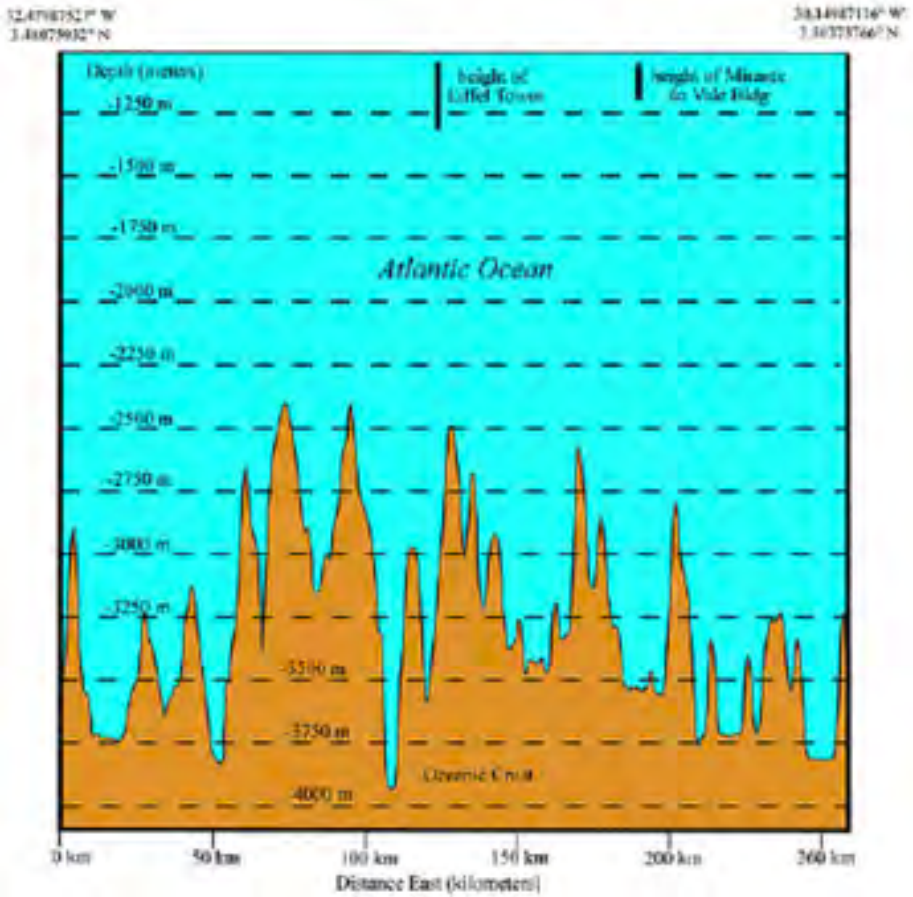
AF447: Failed search attempts

June 6, 2009

June 7, 2009

[Stone *et al.*, Statistical Science, 2013]

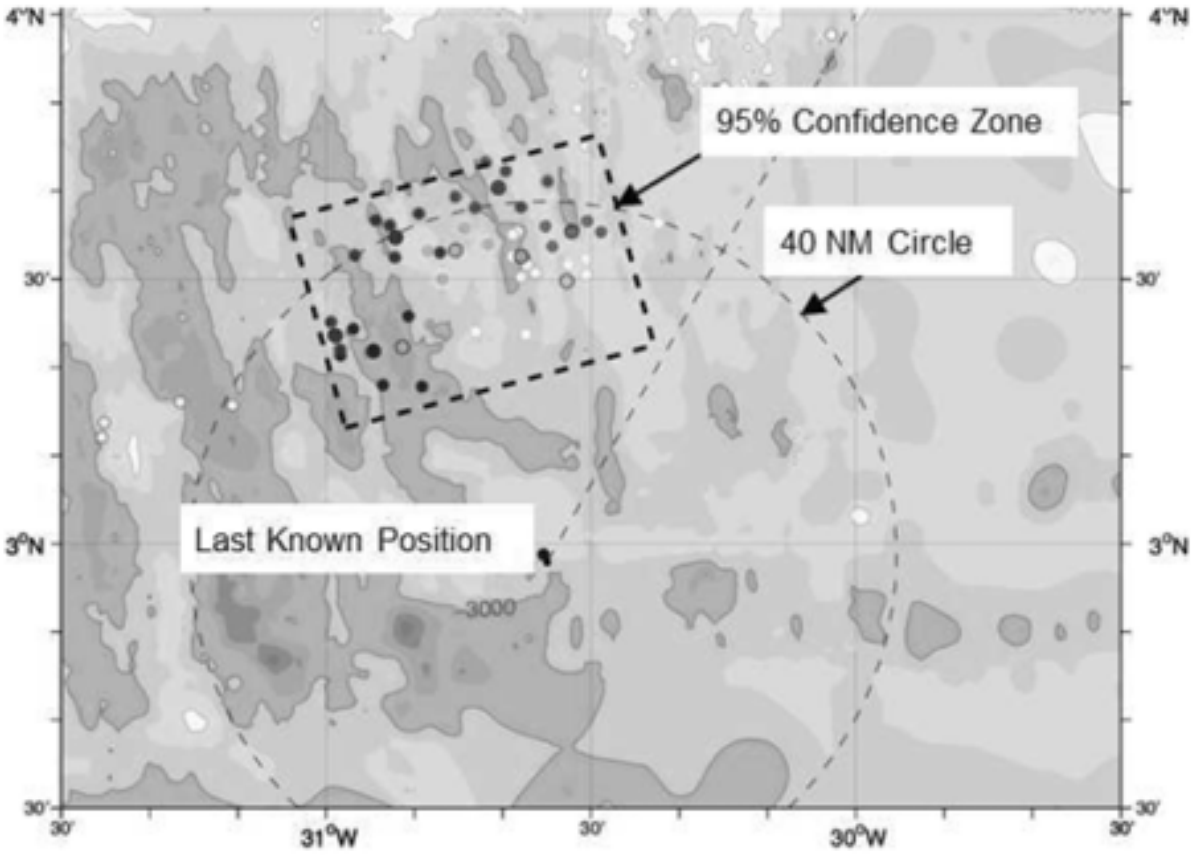
Where is the wreckage?



East-west cross section Atlantic

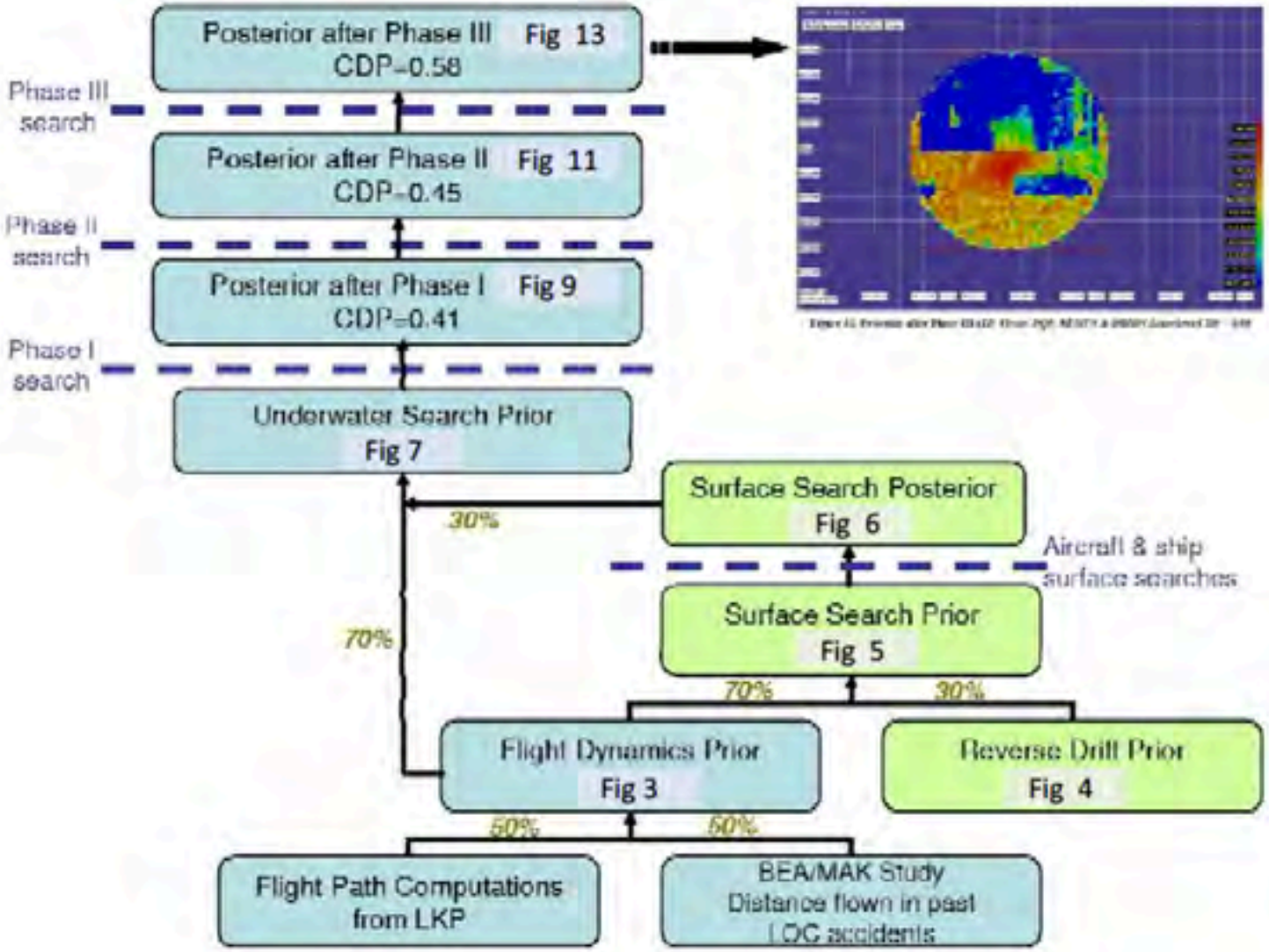
70,000 km^2 were searched, up to 4500 m depth

AF447: Guessed position

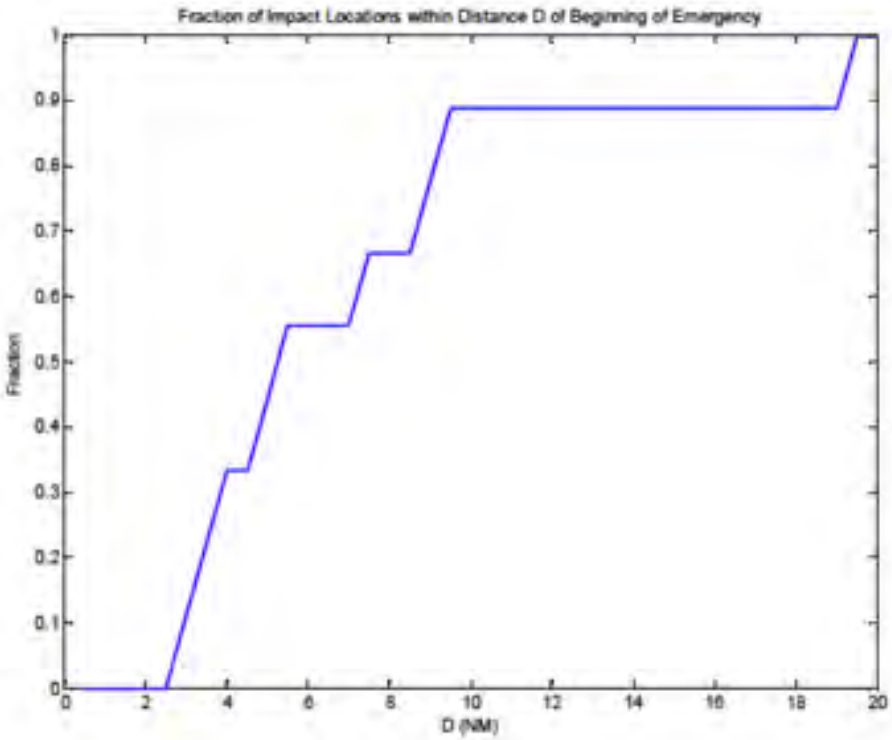


This guided the acoustic search in April 2010.

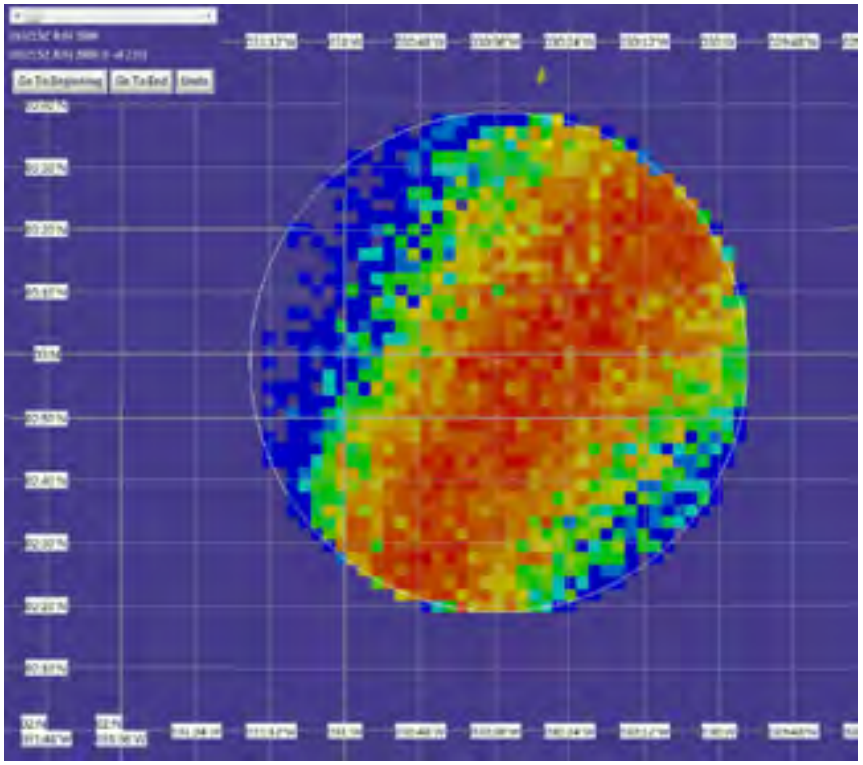
How statisticians came into the play



The priors



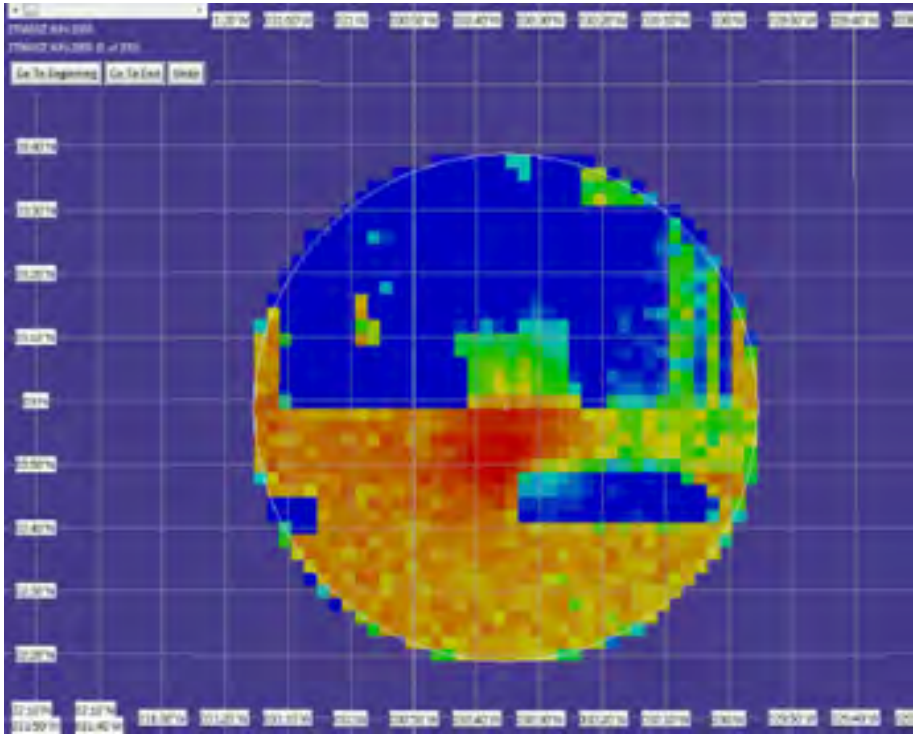
Fraction of impact locations within distance D of beginning of emergency



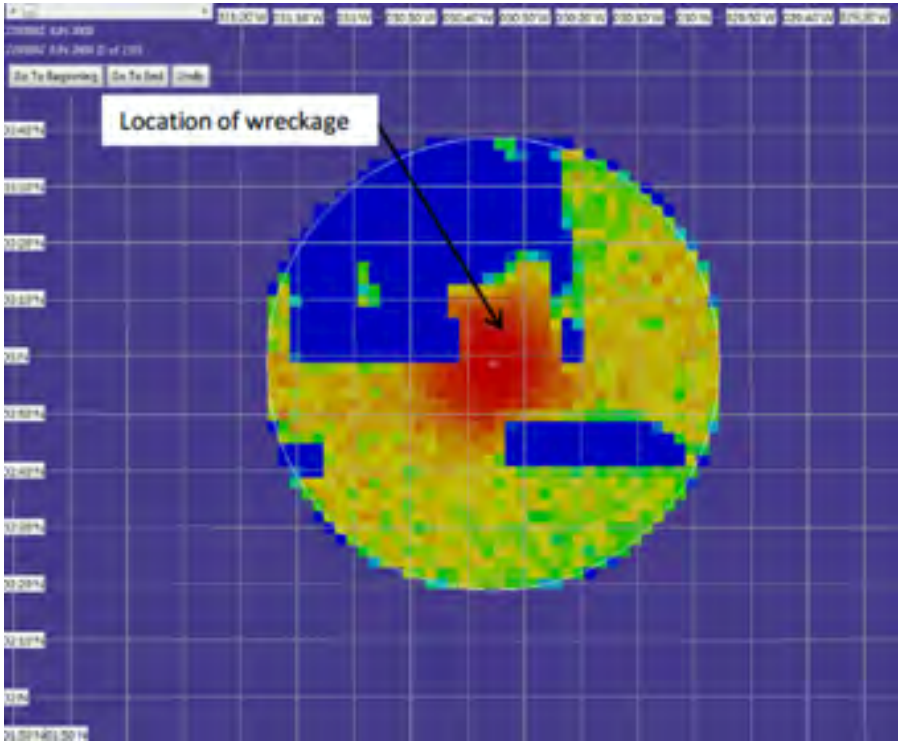
Reverse drift prior (ocean and wind drift) [a]

[a] Currents hard to estimate close to equator.

Two posteriors for location wreckage



Posterior after sonar and UAV search (2010) assuming pingers black boxes function



Posterior pdf assuming pingers of black boxes failed

Rethinking the Bayesian approach

[Daniel Roy, 2011][a]

"In particular, the graphical model formalism that ushered in an era of rapid progress in AI has proven inadequate in the face of [these] new challenges.

A promising new approach that aims to bridge this gap is probabilistic programming, which marries probability theory, statistics and programming languages"

[a] MIT/EECS George M. Sprowls Doctoral Dissertation Award

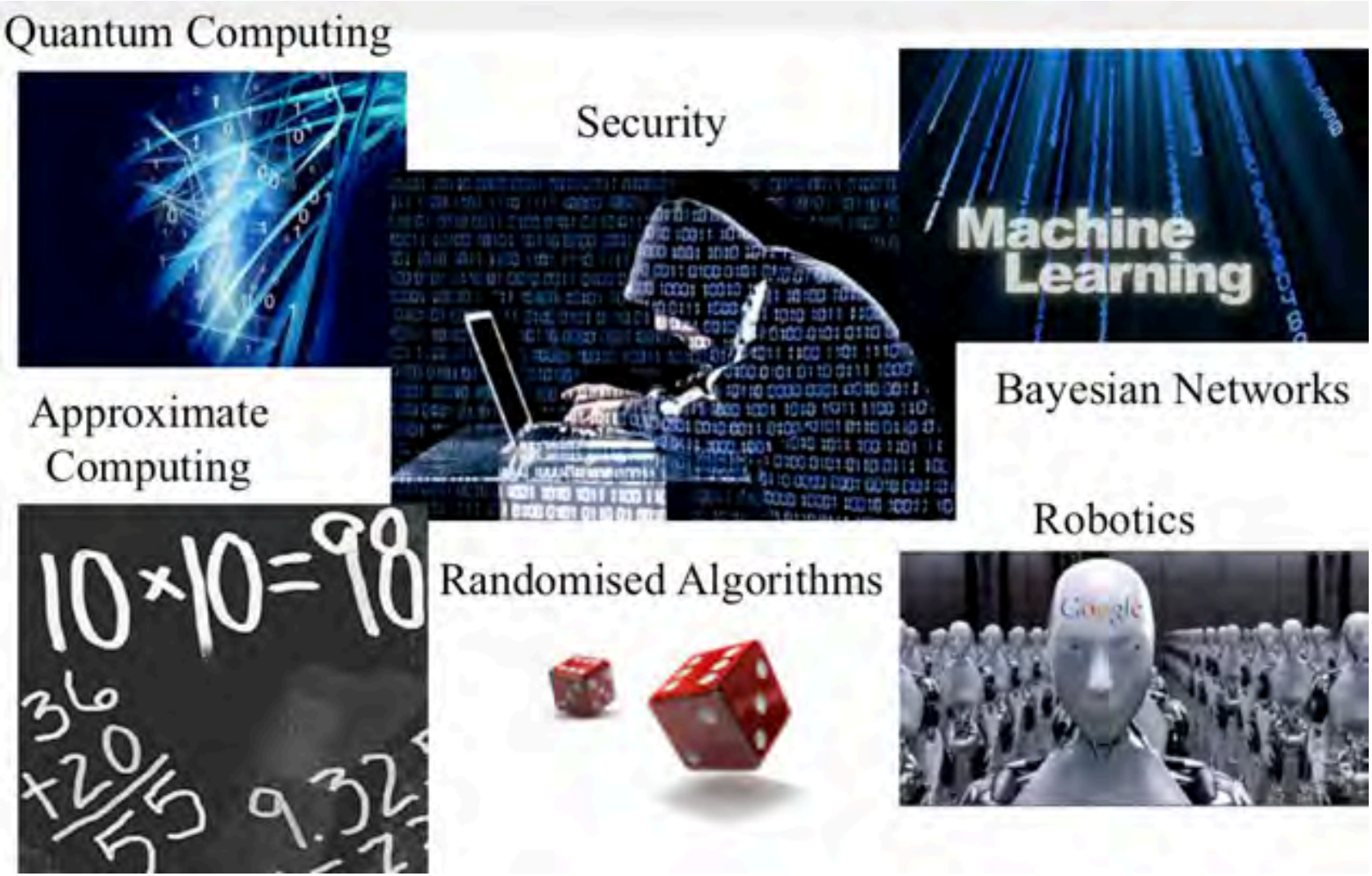
nature Perspective

"There are several reasons why probabilistic programming could prove to be revolutionary for machine intelligence and scientific modelling."

Why? Probabilistic programming

1. ... obviates the need to manually provide inference methods
2. ... enables rapid prototyping
3. ... clearly separates the model and the inference procedures

Applications



Languages


Languages:
Probabilistic C
ProbLog
Church
webPPL
Figaro
PyMC
Tabular
R2
.....
A. Pfeffer
N. Goodman
probabilistic-programming.org
Microsoft
Research

Roadmap of my lectures

Overview

Dijkstra's guarded command language

- `skip` — empty statement
- `abort` — abortion
- `x := E` — assignment
- `prog1 ; prog2` — sequential composition
- `if (G) prog1 else prog2` — choice
- `prog1 [] prog2` — non-deterministic choice
- `while (G) prog` — iteration

Probabilistic GCL

Kozen

McIver

Morgan

- `skip` — empty statement
- `abort` — abortion
- `x := E` — assignment
- `observe (G)` — **conditioning**
- `prog1 ; prog2` — sequential composition
- `if (G) prog1 else prog2` — choice
- `prog1 [p] prog2` — **probabilistic choice**
- `while (G) prog` — iteration

Let's start simple

```
x := 0 [0.5] x := 1;
y := -1 [0.5] y := 0
```

This program admits four runs and yields the outcome:

$$Pr[x=0, y=0] = Pr[x=0, y=-1] = Pr[x=1, y=0] = Pr[x=1, y=-1] = 1/4$$

A loopy program

For $0 < p < 1$ an arbitrary probability:

```
bool c := true;
int i := 0;
while (c) {
    i++;
    (c := false [p] c := true)
}
```

The loopy program models a geometric distribution with parameter p.

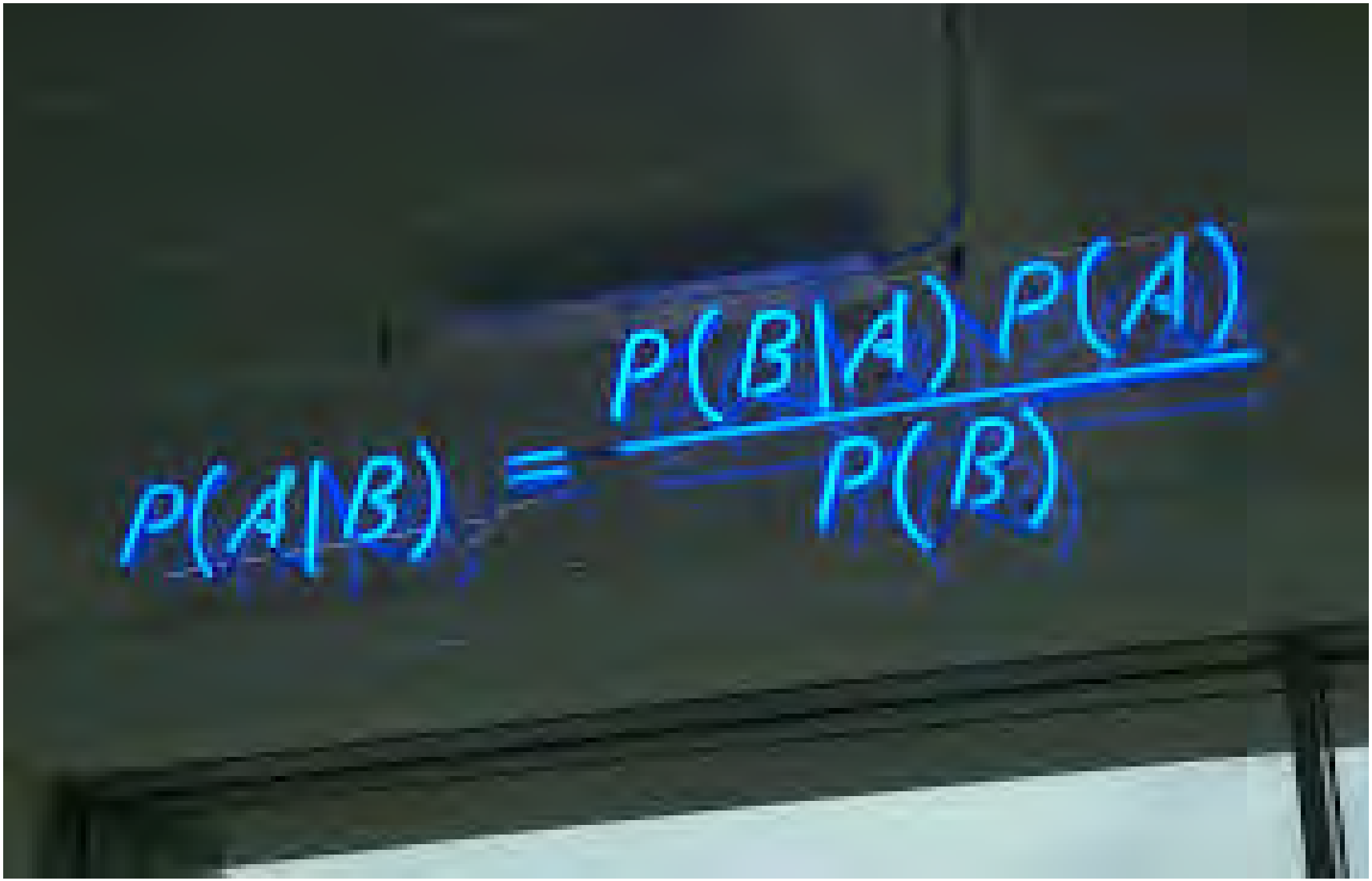
$$Pr[i = N] \;=\; (1-p)^{N-1} \cdot p \quad \text{for } N > 0$$

On termination

```
bool c := true;
int i := 0;
while (c) {
    i++;
    (c := false [p] c := true)
}
```

This program does not always terminate. It almost surely terminates.

Conditioning



Let's start simple

```
x := 0 [0.5] x := 1;
y := -1 [0.5] y := 0;
observe (x+y = 0)
```

This program blocks two runs as they violate `x+y = 0`. Outcome:

$$Pr[x=0, y=0] = Pr[x=1, y=-1] = 1/2$$

Observations thus normalize the probability of the "feasible" program runs

A loopy program

For $0 < p < 1$ an arbitrary probability:

```
bool c := true;
int i : = 0;
while (c) {
    i++;
    (c := false [p] c := true)
}
observe (odd(i))
```

The feasible program runs have a probability $\sum_{N \geq 0} (1-p)^{2N} \cdot p \;=\; \dfrac{1}{2-p}$

This program models the distribution:

$$Pr[i = 2N+1] \;=\; (1-p)^{2N} \cdot p \cdot (2-p) \quad \text{for } N \geq 0$$

$$Pr[i = 2N] = 0$$

Markov chains

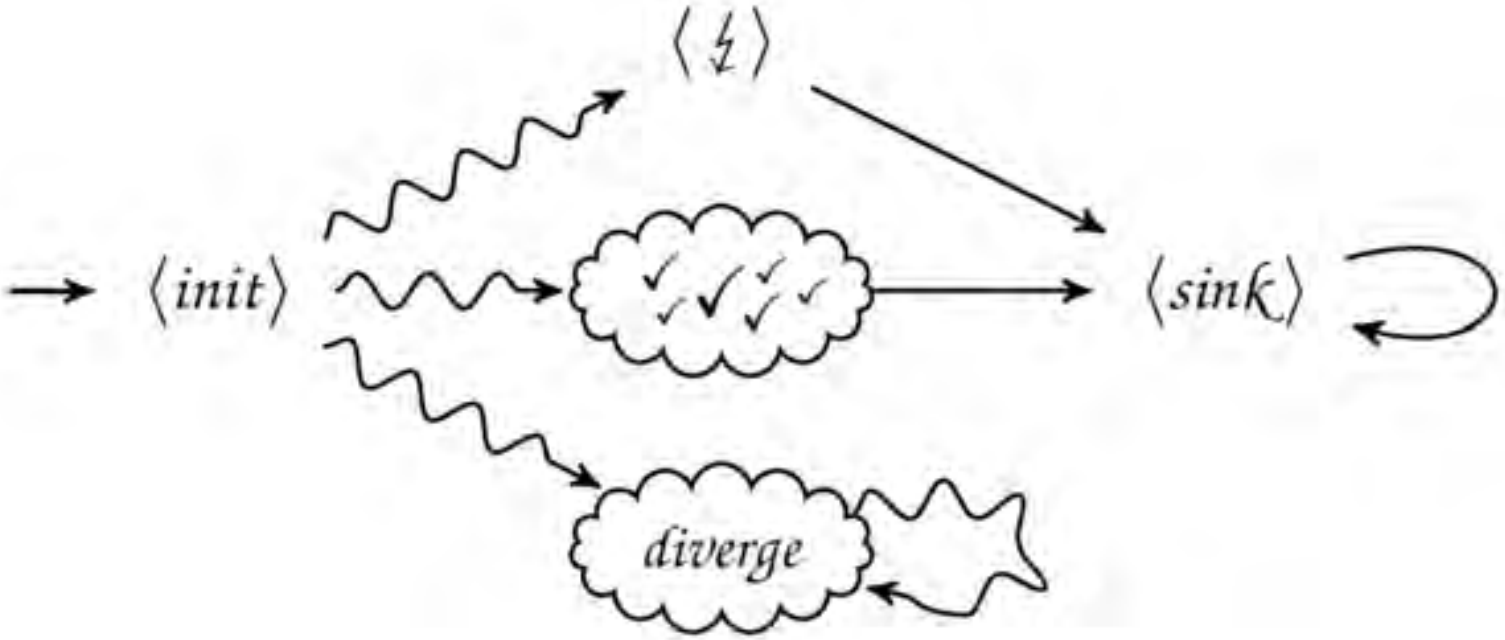
A Markov chain (MC) is a triple $(\Sigma, \sigma_I, \mathbf{P})$ with:

- Σ being a countable set of states
- $\sigma_I \in \Sigma$ the initial state, and
- $\mathbf{P} : \Sigma \to \mathit{Dist}(\Sigma)$ the transition probability function

where $\mathit{Dist}(\Sigma)$ is a discrete probability measure on Σ.

Operational semantics

Aim: Model the behaviour of a program P by the MC $[\![P]\!]$.



This can be defined using Plotkin's SOS-style semantics

Operational semantics

Aim: Model the behaviour of a program P by the MC $[\![P]\!]$.

Approach:

- Take states of the form
  - $\langle Q, s \rangle$ with program Q or $\downarrow$, and variable valuation $s : Var \to \mathbb{Q}$
  - $\langle \lightning \rangle$ models the violation of an observation, and
  - $\langle sink \rangle$ models program termination (successful or violated observation)
- Take initial state $\langle P, s \rangle$ where s fulfils the initial conditions
- Take transition relation $\to$ as smallest relation satisfying the SOS rules

Some SOS rules

$$\langle \texttt{skip}, s\rangle \to \langle \downarrow, s\rangle \qquad \langle \texttt{abort}, s\rangle \to \langle \texttt{abort}, s\rangle$$

$$\frac{s \vDash G}{\langle \texttt{observe}(G), s\rangle \to \langle \downarrow, s\rangle} \qquad \frac{s \nvDash G}{\langle \texttt{observe}(G), s\rangle \to \langle \lightning\rangle}$$

$$\langle \downarrow, s\rangle \to \langle sink\rangle \qquad \langle \lightning\rangle \to \langle sink\rangle \qquad \langle sink\rangle \to \langle sink\rangle$$

$$\langle x := E, s\rangle \to \langle \downarrow, s[x := s([\![E]\!])]\rangle$$

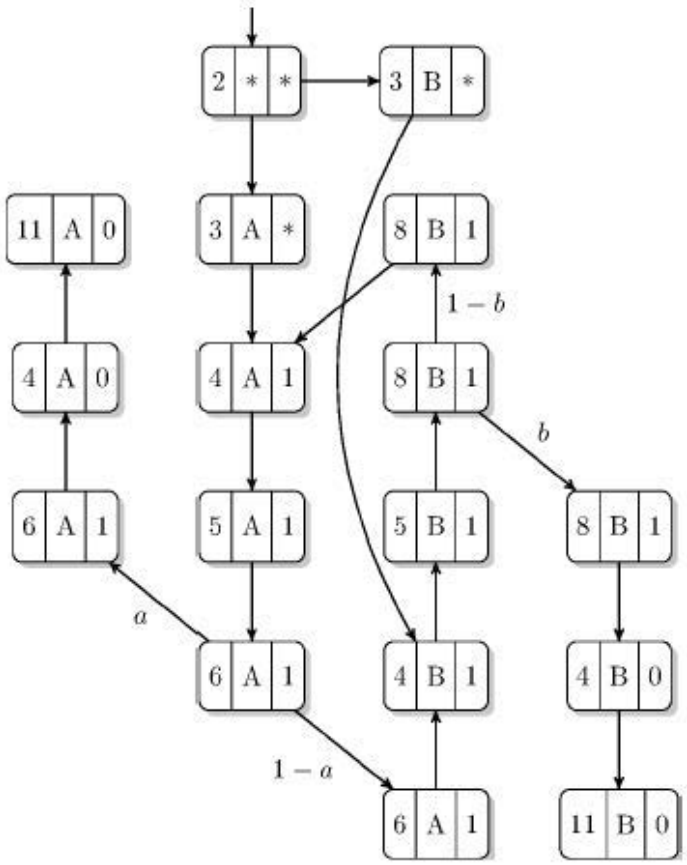
$$\langle P[\, p]\, Q, s\rangle \to \mu \text{ with } \mu(\langle P, s\rangle) = p \text{ and } \mu(\langle Q, s\rangle) = 1-p$$

$$\frac{\langle P, s\rangle \to \langle \lightning\rangle}{\langle P; Q, s\rangle \to \langle \lightning\rangle} \qquad \frac{\langle P, s\rangle \to \mu}{\langle P; Q, s\rangle \to \nu} \text{ with } \nu(\langle P'; Q', s'\rangle) = \mu(\langle P', s'\rangle) \text{ where } \downarrow; Q = Q$$

The good, the bad, and the ugly

Example operational semantics

```
int cowboyDuel(float a, b) {
  int t := A [0.5] t := B;
  bool c := true;
  while (c) {
    if (t = A) {
      (c := false [a] t := B);
    } else {
      (c := false [b] t := A);
    }
  }
  return t;
}
```



This (parametric) MC is finite. Once we count the number of shots before one of the cowboys dies, the MC becomes countably infinite.

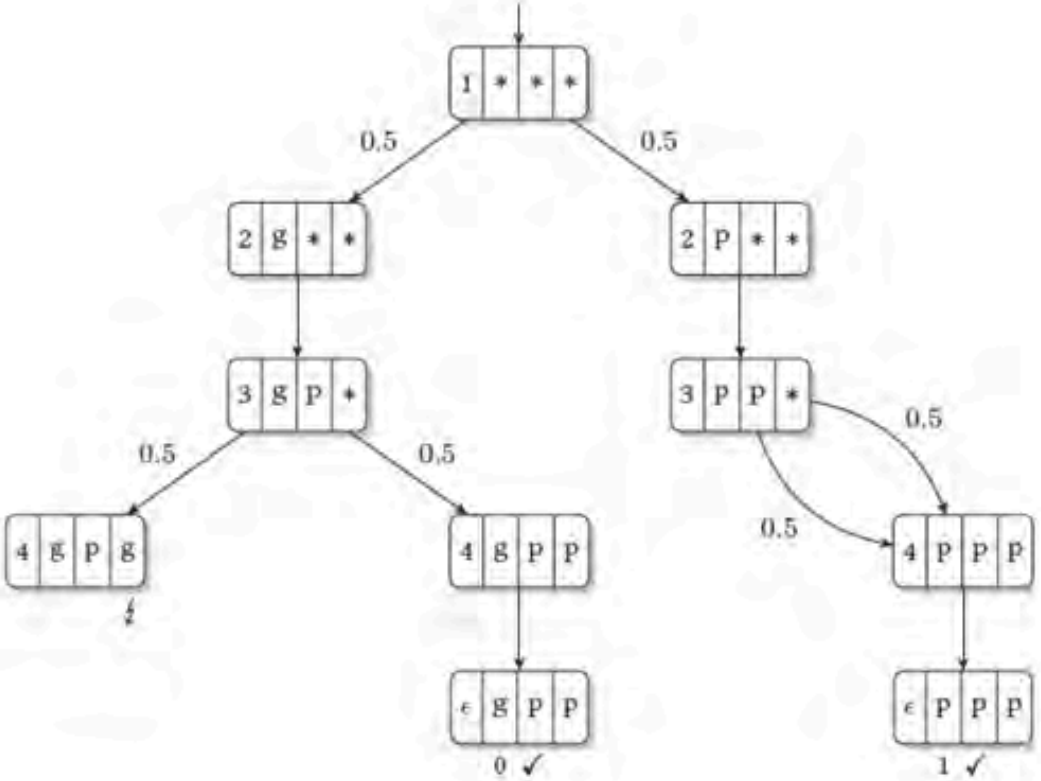
The piranha problem

[Tijms, 2004]

One fish is contained within the confines of an opaque fishbowl. The fish is equally likely to be a piranha or a goldfish. A sushi lover throws a piranha into the fish bowl alongside the other fish. Then, immediately, before either fish can devour the other, one of the fish is blindly removed from the fishbowl. The fish that has been removed from the bowl turns out to be a piranha. What is the probability that the fish that was originally in the bowl by itself was a piranha?

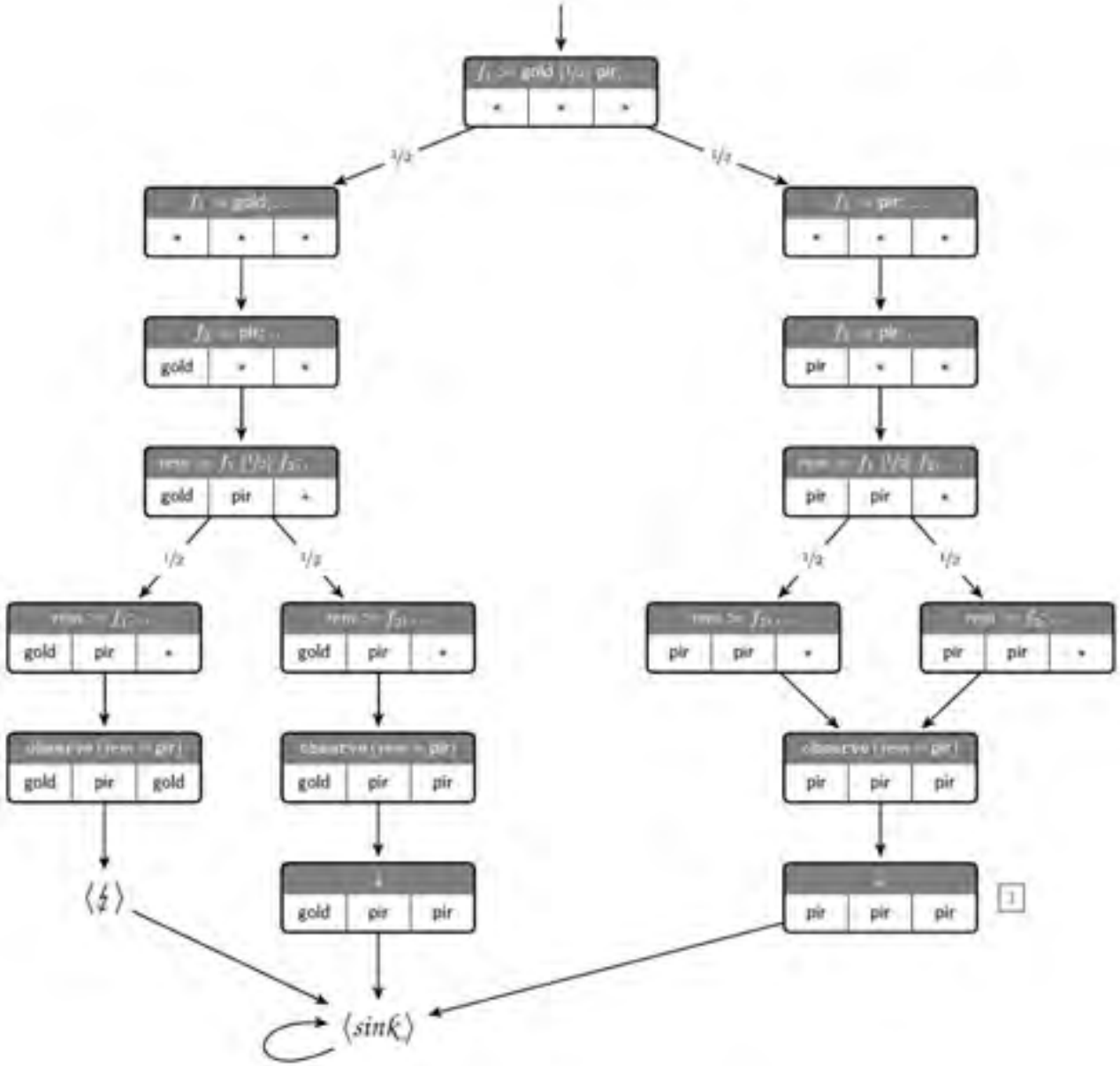
The piranha puzzle

```
f1 := gf [0.5] f1 := pir;
f2 := pir;
s := f1 [0.5] s := f2;
observe (s = pir)
```

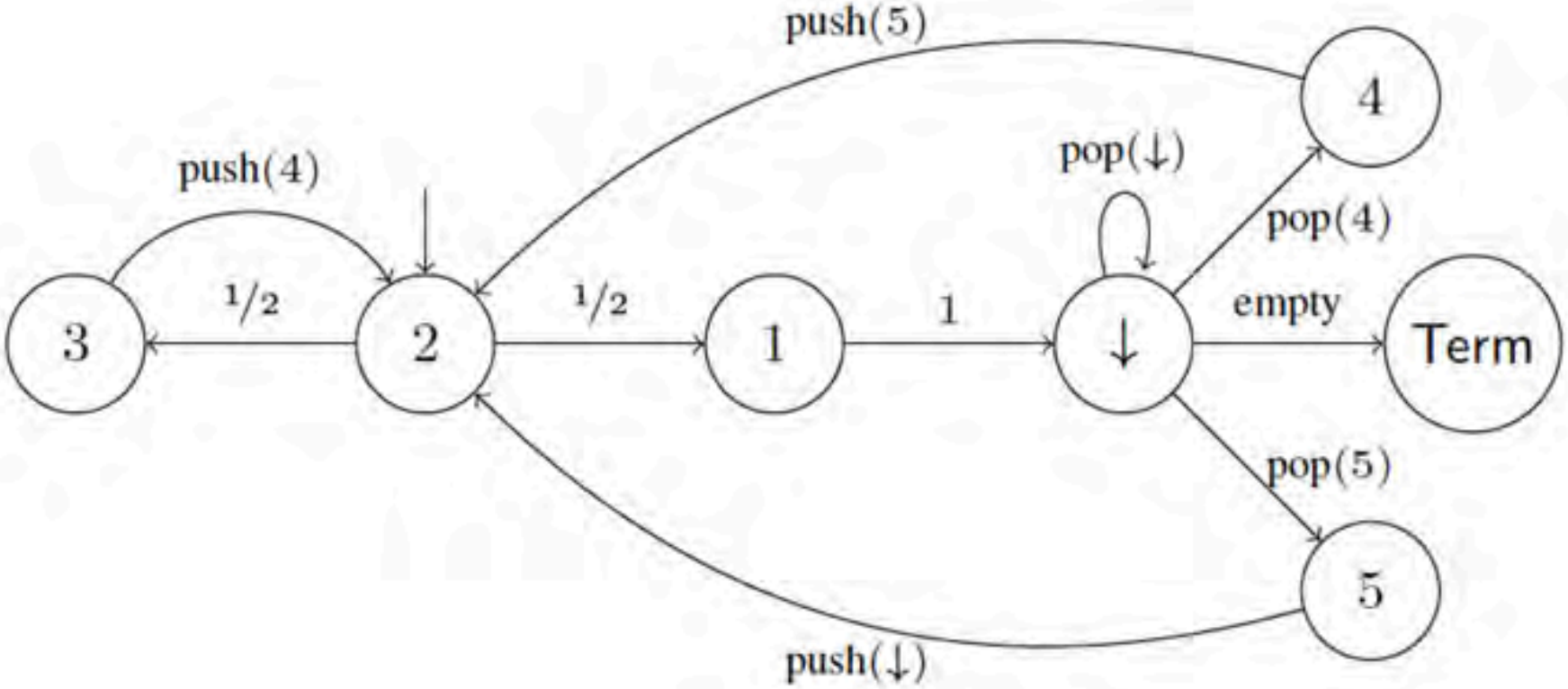


The full operational semantics

```
f1 := gf [0.5] f1 := pir
f2 := pir;
s := f1 [0.5] s := f2;
observe (s = pir)
```



Recursion: pushdown Markov chains



$$\{\mathsf{skip}^1\}\ [1/2]^2\ \{\mathsf{call}\ P^3;\ \mathsf{call}\ P^4;\ \mathsf{call}\ P^5\ \}$$

Take-home messages

Probabilistic programs:

- extend the expressive power of probabilistic graphical models
- are claimed to have many applications
- are usual programs with random assignment and conditioning
- have an operational semantics a (countably infinite) Markov chains
- recursive programs give rise to pushdown Markov chains

Next lecture: how to perform inference on probabilistic programs?

Today's pub quiz: which pairs are equivalent?

```
{ x := 0 [0.5] x := 1 };
observe(x = 1)
```

```
{ x := 0; observe(x = 1) }
[0.5]
{ x := 1; observe(x = 1) }
```

```
x := 2;
```

```
x := 2 [0.5] abort
```

```
int x := 1;
while (x = 1) {
    x := 1
}
```

```
int x := 1;
while (x = 1) {
    x := 1 [0.5] x := 0;
    observe (x = 1)
}
```

Overview

Dijkstra's guarded command language

- `skip` — empty statement
- `abort` — abortion
- `x := E` — assignment
- `prog1 ; prog2` — sequential composition
- `if (G) prog1 else prog2` — choice
- `prog1 [] prog2` — non-deterministic choice
- `while (G) prog` — iteration

Weakest preconditions

Weakest precondition **[Dijkstra 1975]**

An predicate maps program states onto Booleans.

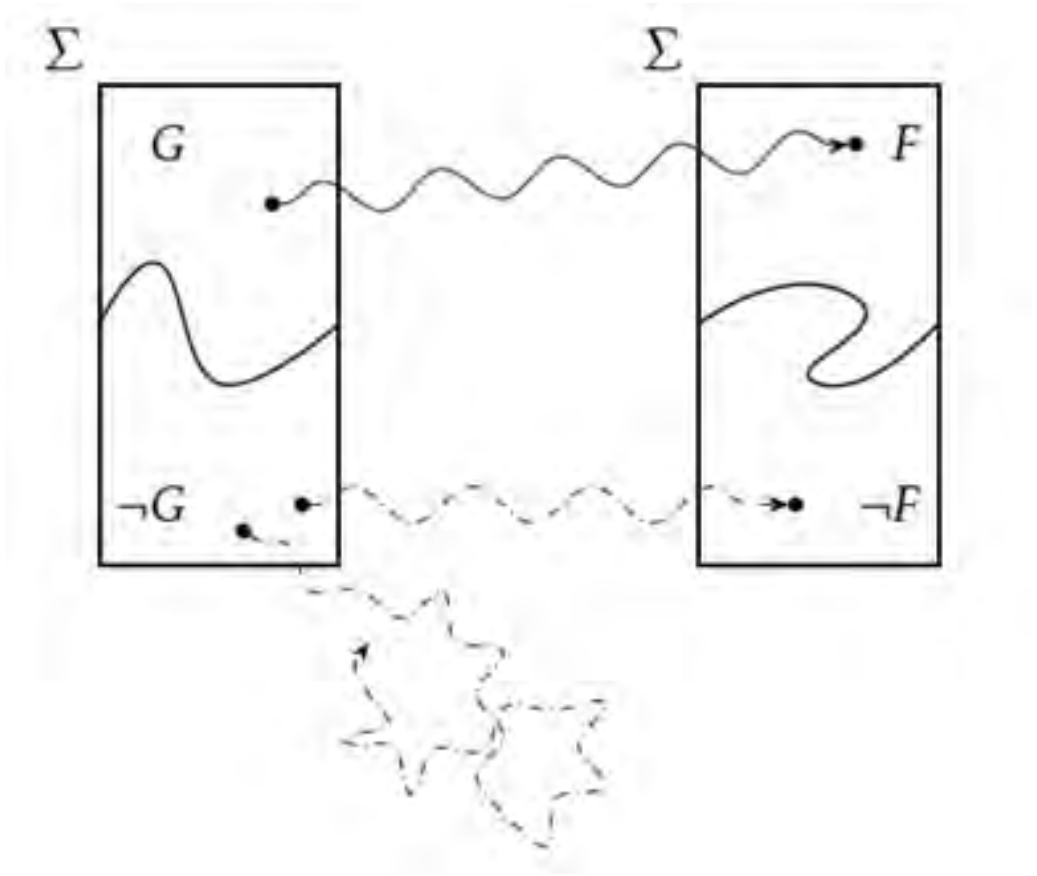
A predicate transformer is a total function between two predicates.

The predicate transformer $wp(P, F)$ for program P and postcondition F yields the "weakest" precondition E on the initial state of P ensuring that the execution of P terminates in a final state satisfying F.

Hoare triple $\{E\}\, P\, \{F\}$ holds for total correctness iff $E \Rightarrow wp(P, F)$.

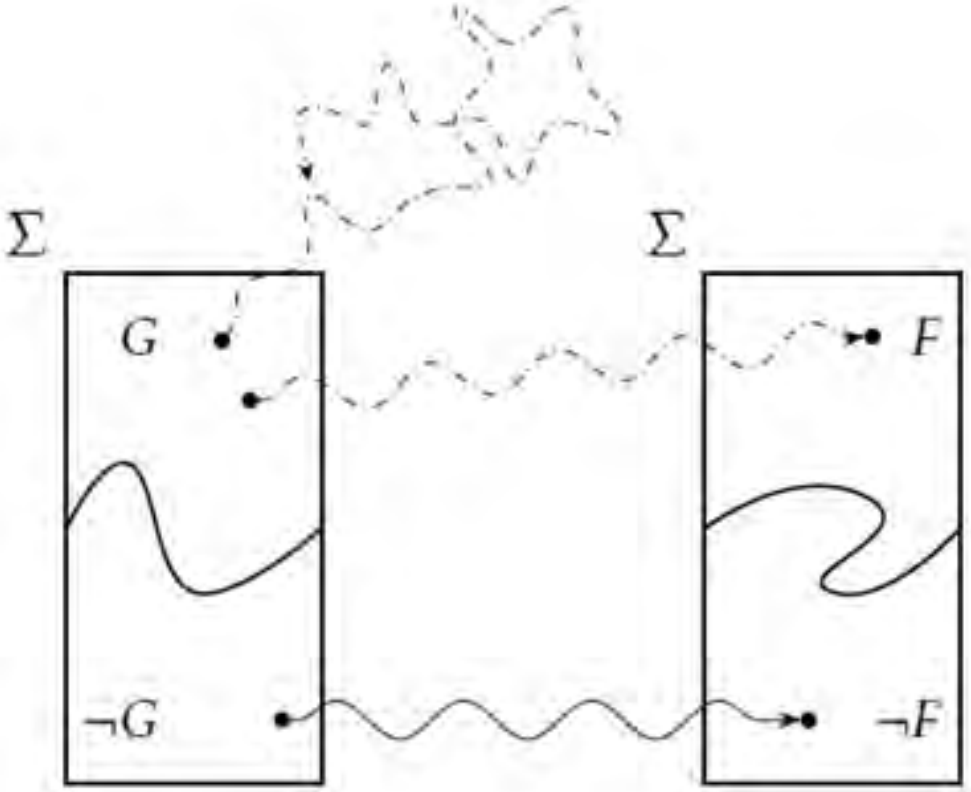
A weakest liberal precondition $wlp(P, F)$ yields the weakest precondition for which P either does not terminate or establishes F. It does not ensure termination and corresponds to Hoare logic for partial correctness.

Weakest precondition G w.r.t. F



This holds for deterministic programs. A nondeterministic program starting in $\neg G$ may also terminate in a state satisfying F.

Weakest liberal precondition G w.r.t. F



This holds for deterministic programs. A nondeterministic program starting in $\neg G$ may also terminate in a state satisfying F or diverge.

Predicate transformer semantics of Dijkstra's GCL

Syntax

- `skip`
- `abort`
- `x := E`
- `P1 ; P2`
- `if (G) P1 else P2`
- `P1 [] P2`
- `while (G)P`

Semantics $wp(P, F)$

- F
- $false$
- $F[x := E]$
- $wp(P_1, wp(P_2, F))$
- $(G \wedge wp(P_1, F)) \vee (\neg G \wedge wp(P_2, F))$
- $wp(P_1, F) \wedge wp(P_2, F)$
- $\mu X. ((G \wedge wp(P, X)) \vee (\neg G \wedge F))$

μ is the least fixed point operator wrt. the ordering $\Rightarrow$ on predicates.

wlp-semantics differs from wp-semantics only for `while` and `abort`.

Probabilistic GCL

Kozen

McIver

Morgan

- `skip` — empty statement
- `abort` — abortion
- `x := E` — assignment
- `observe (G)` — **conditioning**
- `prog1 ; prog2` — sequential composition
- `if (G) prog1 else prog2` — choice
- `prog1 [p] prog2` — **probabilistic choice**
- `while (G) prog` — iteration

Expectations

Weakest pre-expectation **[McIver & Morgan 2004]**

An expectation[2] maps program states onto non-negative reals (extended with ∞). It is the quantitative analogue of a predicate. Let $f \leq g$ iff for every state s it holds $f(s) \leq g(s)$.

An expectation transformer is a total function between two expectations.

The transformer $wp(P, f)$ for program P and post-expectation f yields the least expectation e on P's initial state ensuring that P's execution terminates with an expectation f.

Annotation $\{e\}\, P\, \{f\}$ holds for total correctness iff $e \;\leq\; wp(P, f)$.

Weakest liberal pre-expectation $wlp(P, f) \;=\;$ "$wp(P, f) + Pr[P \text{ diverges}]$".

[2] Not to be confused what expectations are in probability theory.

Expectation transformer semantics of pGCL

Syntax

- `skip`
- `abort`
- `x := E`
- `observe (G)`
- `P1 ; P2`
- `if (G) P1 else P2`
- `P1 [p] P2`
- `while (G)P`

Semantics $wp(P, f)$

- f
- 0
- $f[x := E]$
- $[G] \cdot f$
- $wp(P_1, wp(P_2, f))$
- $[G] \cdot wp(P_1, f) + [\neg G] \cdot wp(P_2, f)$
- $p \cdot wp(P_1, f) + (1-p) \cdot wp(P_2, f)$
- $\mu X.\ ([G] \cdot wp(P, X) + [\neg G] \cdot f)$

μ is the least fixed point operator wrt. the ordering $\leq$ on expectations.

wlp-semantics differs from wp-semantics only for `while` and `abort`.

```
x := 0 [1/2] x := 1; // command c1
y := 0 [1/3] y := 1; // command c2
```

$$
\begin{aligned}
& wp(c_1; c_2, [x = y]) \\
= \; & wp(c_1, wp(c_2, [x = y])) \\
= \; & wp(c_1, {}^1\!/\!_3 \cdot wp(y := 0, [x = y]) + {}^2\!/\!_3 \cdot wp(y := 1, [x = y])) \\
= \; & wp(c_1, {}^1\!/\!_3 \cdot [x = 0] + {}^2\!/\!_3 \cdot [x = 1]) \\
= \; & {}^1\!/\!_2 \cdot wp(x := 0, {}^1\!/\!_3 \cdot [x = 0] + {}^2\!/\!_3 \cdot [x = 1]) + {}^1\!/\!_2 \cdot wp(x := 1, {}^1\!/\!_3 \cdot [x = 0] + {}^2\!/\!_3 \cdot [x = 1]) \\
= \; & {}^1\!/\!_2 \cdot ({}^1\!/\!_3 \cdot [0 = 0] + {}^2\!/\!_3 \cdot [0 = 1]) + {}^1\!/\!_2 \cdot ({}^1\!/\!_3 \cdot [1 = 0] + {}^2\!/\!_3 \cdot [1 = 1]) \\
= \; & {}^1\!/\!_2 \cdot ({}^1\!/\!_3 \cdot \mathbf{1} + {}^2\!/\!_3 \cdot \mathbf{0}) + {}^1\!/\!_2 \cdot ({}^1\!/\!_3 \cdot \mathbf{0} + {}^2\!/\!_3 \cdot \mathbf{1}) \\
= \; & {}^1\!/\!_2 \cdot ({}^1\!/\!_3 + {}^2\!/\!_3) \\
= \; & {}^1\!/\!_2
\end{aligned}
$$

Elementary properties

- Monotonicity: $f \leq g$ implies $wp(P, f) \leq wp(P, g)$
- Linearity: $wp(P, \alpha \cdot f + g) = \alpha \cdot wp(P, f) + wp(P, g)$
- Duality: $wlp(P, f) = \mathbf{1} - wp(P, \mathbf{1} - f)$
- Zero: $wp(P, \mathbf{0}) = \mathbf{0}$

Quantitative loop invariants

Recall that for while-loops we have:

$$wp(\text{while}(G)\{P\}, f) \;=\; \mu X.\, ([G] \cdot wp(P, X) + [\neg G] \cdot f)$$

To determine this *wp*, we use an "invariant" I such that $[\neg G] \cdot I \;\leq\; f$.

Quantitative loop invariant

Expectation I is a quantitative loop invariant if — by consecution —

it is preserved by loop iterations: $[G] \cdot I \;\leq\; wlp(P, I)$.

To guarantee soundness, I has to fulfill either:

1. I is bounded from below and by above by some constants, or
2. on each iteration there is a probability $\epsilon > 0$ to exit the loop

Then: $\{I\}$ while$(G)\{P\}$ $\{f\}$ is a correct program annotation.

Invariant synthesis for linear programs

inspired by [Colón *et al.* 2002]

1. Speculatively annotate a program with linear expressions:

$$[\alpha_1 \cdot x_1 + \ldots + \alpha_n \cdot x_n + \alpha_{n+1} \ll 0] \cdot (\beta_1 \cdot x_1 + \ldots + \beta_n \cdot x_n + \beta_{n+1})$$

   with real parameters α_i, β_i, program variable x_i, and $\ll \in \{<, \leq\}$.
2. Transform these numerical constraints into Boolean predicates.
3. Transform these predicates into non-linear FO formulas.
4. Use constraint-solvers for quantifier elimination (e.g., REDLOG).
5. Simplify the resulting formulas (e.g., using SLFQ and SMT solving).
6. Exploit resulting assertions to infer program correctness.

Soundness and completeness

For any linear pGCL program annotated with propositionally linear expressions, this method will find all parameter solutions that make the annotation valid, and no others.

Duelling cowboys

```
int cowboyDuel(float a, b) { // 0 < a < 1, 0 < b < 1
   int t := A [0.5] t := B; // decide who shoots first
   bool c := true;
   while (c) {
     if (t = A) {
         (c := false [a] t := B); // A hits B with prob. a
     } else {
         (c := false [b] t := A); // B hits A with prob. b
     }
   }
return t; // the survivor
}
```

Using wp-reasoning, one can prove that cowboy A wins the duel with probability $\frac{(1-b)\cdot a}{a+b-a\cdot b}$. Syntactic reasoning only. Semi-automatable.

Duelling cowboys: when does A win?

```
int cbDuel(float a, b) {
  int t := A;
  int c := 1;
  while (c = 1) {
    if (t = A) {
      (c := 0 [a] t := B);
    } else {
      (c := 0 [b] t := A);
    }
  }
  return t;
}
```

Aim: find expectation T

Satisfying $T \leq [\mathrm{t} = A]$ upon termination.

Observation

On entering the loop, c = 1 and either t = A or t = B.

Template suggestion

$$
\begin{aligned}
T \quad = \quad & \underbrace{[\mathrm{t} = A \wedge \mathrm{c} = 0]}_{A \text{ wins duel}} \cdot 1 \\
& + \underbrace{[\mathrm{t} = A \wedge \mathrm{c} = 1]}_{A\text{'s turn}} \cdot \alpha \\
& + \underbrace{[\mathrm{t} = B \wedge \mathrm{c} = 1]}_{B\text{'s turn}} \cdot \beta
\end{aligned}
$$

Duelling cowboys: when does A win?

Invariant template

$T = [\mathrm{t} = A \wedge \mathrm{c} = 0] \cdot 1 + [\mathrm{t} = A \wedge \mathrm{c} = 1] \cdot \alpha + [\mathrm{t} = B \wedge \mathrm{c} = 1] \cdot \beta$

Initially, $\mathrm{t} = A \wedge \mathrm{c} = 1$ and thus $\alpha = Pr\{A \text{ wins duel}\}$.

Constraint solving yields

$a \cdot \beta - a + \alpha - \beta \leq 0 \quad \wedge \quad b \cdot \alpha - \alpha + \beta \leq 0$

Simplification yields

$\beta \leq (1 - b) \cdot \alpha \quad \text{and} \quad \alpha \leq \dfrac{a}{a + b - a \cdot b}$

As we want to maximise the probability to win

$\beta = (1 - b) \cdot \alpha \quad \text{and} \quad \alpha = \dfrac{a}{a + b - a \cdot b}$

It follows that cowboy A wins the duel with probability $\frac{a}{a+b-a \cdot b}$.

Quantitative loop invariant

Annotated program for post-expectation $[t = A]$

```
int cowboyDuel(a, b) {
  ⟨(1−b)a / (a+b−ab)⟩
  ⟨min{a/(a+b−ab), (1−b)a/(a+b−ab)}⟩
  (t := A [] t := B);
  ⟨[t = A] · a/(a+b−ab) + [t = B] · (1−b)a/(a+b−ab)⟩
  c := 1;
  ⟨[t = A ∧ c = 0] · 1 + [t = A ∧ c = 1] · a/(a+b−ab) + [t = B ∧ c = 1] · (1−b)a/(a+b−ab)⟩
  while (c = 1) {
    ⟨[t = A ∧ c = 1] · a/(a+b−ab) + [t = B ∧ c = 1] · (1−b)a/(a+b−ab)⟩
    ⟨[t = A ∧ c ≠ 1] · a + [t = A ∧ c = 1] · a/(a+b−ab)
        + [t = B ∧ c = 0] · (1 − b) + [t = B ∧ c = 1] · (1−b)a/(a+b−ab)⟩
    if (t = A) {
      (c := 0 [a] t := B);
    } else {
      (c := 0 [b] t := A);
    }
    ⟨[t = A ∧ c = 0] · 1 + [t = A ∧ c = 1] · a/(a+b−ab) + [t = B ∧ c = 1] · (1−b)a/(a+b−ab)⟩
  }
  ⟨[c ≠ 1] · ([t = A ∧ c = 0] · 1 + [t = A ∧ c = 1] · a/(a+b−ab)
      + [t = B ∧ c = 1] · (1−b)a/(a+b−ab))⟩
  ⟨[t = A]⟩
  return t; // the survivor
}
```

Program equivalence: $X - Y$

```
int XminY1(float p, q){
  int x, c := 0, 1;
  while (c) {
    (x +:= 1 [p] c := 0);
  }
  c := 1;
  while (c) {
    (x -:= 1 [q] c := 0);
  }
  return x;
}
```

```
int XminY2(float p, q){
  int x := 0;
  (c := 0 [0.5] c := 1);
  if (c) {
    while (c) {
      (x +:= 1 [p] c := 0);
    }
  } else {
    c := 1;
    while (c) {
      x -:= 1;
      (skip [q] c := 0);
    }
  }
return x;
}
```

Using wp-reasoning, one can prove that expected outcomes coincide iff $q = \frac{1}{2-p}$.

Program equivalence: $X - Y$

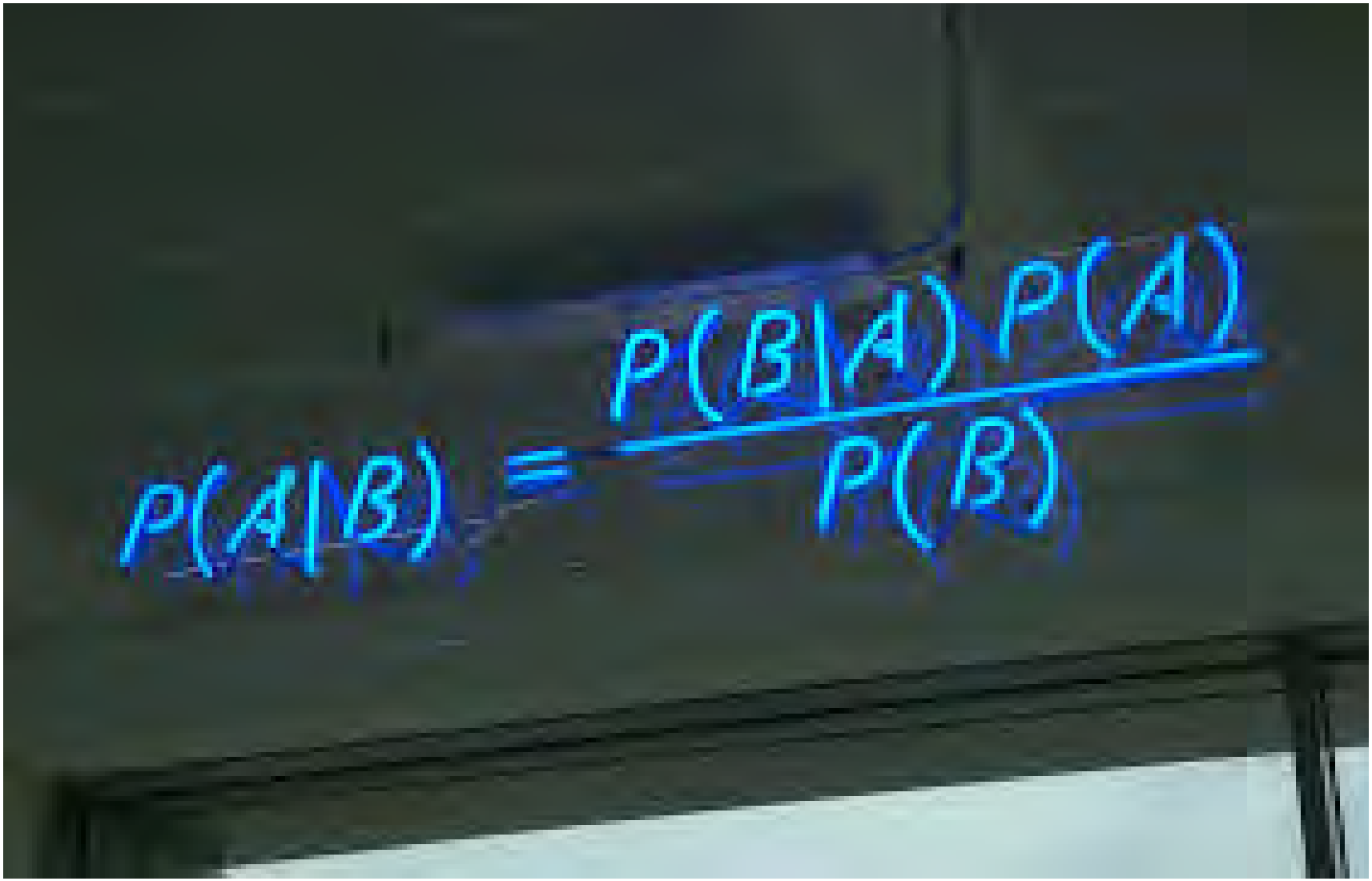
```
int XminY1(float p, q){
  int x, f := 0, 0;
  while (f = 0) {
    (x++ [p] f := 1);
  }
  f := 0;
  while (f = 0) {
    (x-- [q] f := 1);
  }
  return x;
}
```

```
int XminY2(float p, q){
  int x, f := 0, 0;
  (f := 0 [0.5] f := 1);
  if (f = 0) {
    while (f = 0) {
      (x++ [p] f := 1);
    }
  } else {
    f := 0;
    while (f = 0) {
      x--;
      (skip [q] f := 1);
    }
  }
  return x;
}
```

Using template $x + [\mathrm{f} = 0] \cdot \alpha$ we find the invariants :

$\alpha_{11} = \frac{p}{1-p}$, $\alpha_{12} = -\frac{q}{1-q}$, $\alpha_{21} = \alpha_{11}$ and $\alpha_{22} = -\frac{1}{1-q}$.

Conditioning



The piranha problem

[Tijms, 2004]

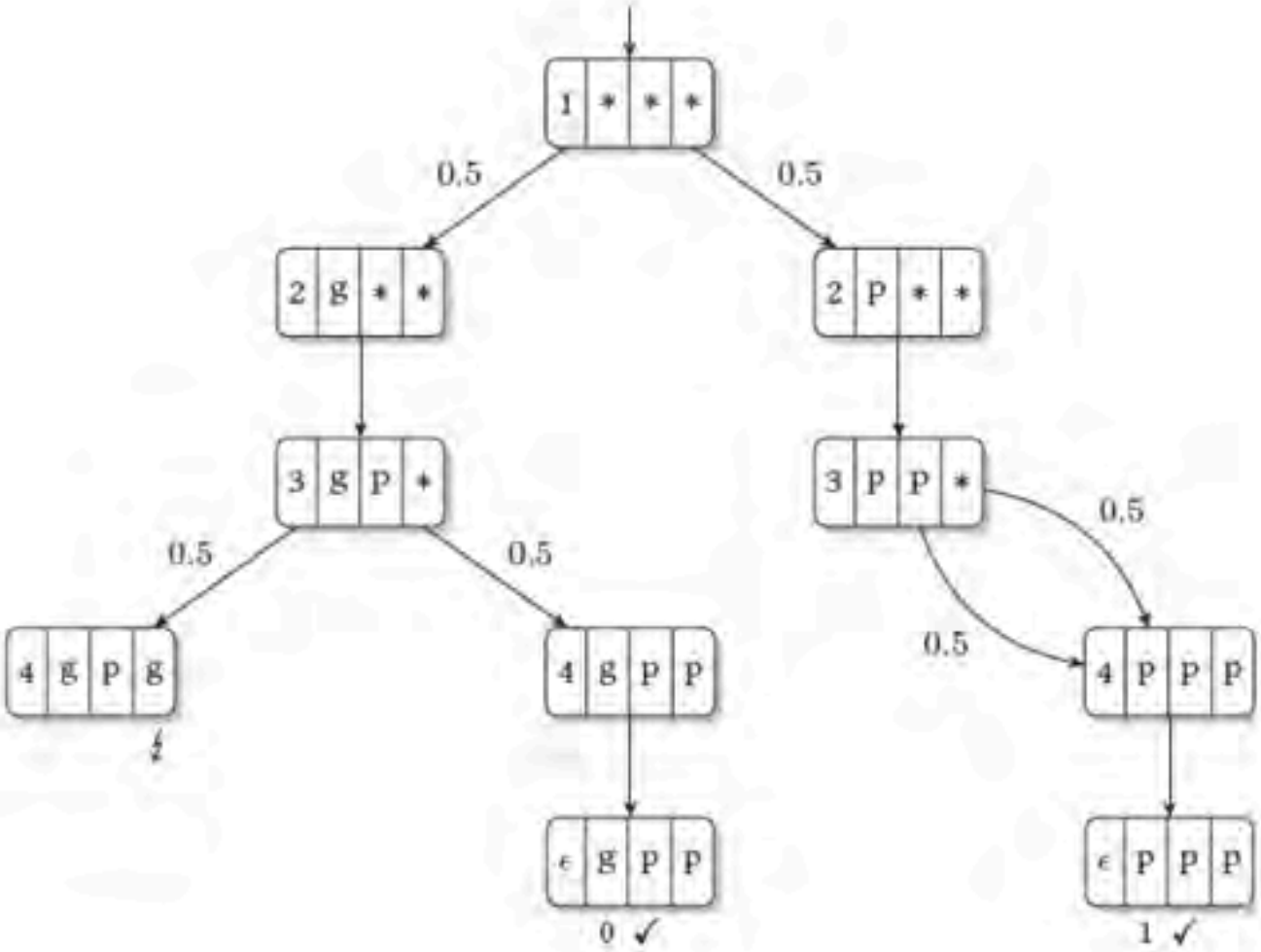
One fish is contained within the confines of an opaque fishbowl. The fish is equally likely to be a piranha or a goldfish. A sushi lover throws a piranha into the fish bowl alongside the other fish. Then, immediately, before either fish can devour the other, one of the fish is blindly removed from the fishbowl. The fish that has been removed from the bowl turns out to be a piranha. What is the probability that the fish that was originally in the bowl by itself was a piranha?

The piranha puzzle

```
f1 := gf [0.5] f1 := pir;
f2 := pir;
s := f1 [0.5] s := f2;
observe (s = pir)
```

The piranha puzzle

```
f1 := gf [0.5] f1 := p
f2 := pir;
s := f1 [0.5] s := f2;
observe (s = pir)
```



What is the probability that the original fish in the bowl was a piranha?

Equip the Markov chain with rewards. Consider expected rewards.

Rewards

Reasoning about expectation using the operational semantics: use rewards.

MC with rewards

An MC with rewards is a pair (M, r) with M an MC with state space Σ and $r : \Sigma \to \mathbb{R}$ a function assigning a real reward to each state.

The reward $r(\sigma)$ stands for the reward earned on leaving state σ.

Cumulative reward for reachability

Let $\pi = \sigma_0 \ldots \sigma_n$ be a finite path in (M, r) and $T \subseteq \Sigma$ a set of target states with $\pi \models \Diamond T$. The cumulative reward along π until reaching T is:

$$r_T(\pi) = r(\sigma_0) + \ldots + r(\sigma_{k-1}) \text{ where } \sigma_i \notin T \text{ for all } i < k \text{ and } \sigma_k \in T.$$

If $\pi \not\models \Diamond T$, then $r_T(\pi) = 0$.

Expected reward reachability

Expected reward for reachability

The expected reward until reaching $T \subseteq \Sigma$ from $\sigma \in \Sigma$ is:

$$\mathrm{ER}^M(\sigma, \Diamond T) \;=\; \sum_{\pi \models \Diamond T} Pr^M(\hat{\pi}) \cdot r_T(\hat{\pi})$$

where $\hat{\pi} = \sigma_0 \ldots \sigma_k$ is the shortest prefix of π such that $\sigma_k \in T$ and $\sigma_0 = \sigma$.

Conditional expected reward

Let $\mathrm{ER}^M(\sigma, \Diamond T \cap \neg \Diamond U)$ be the conditional expected reward until reaching T while avoiding states in $U \subseteq \Sigma$ in MC M.

On computing expected rewards

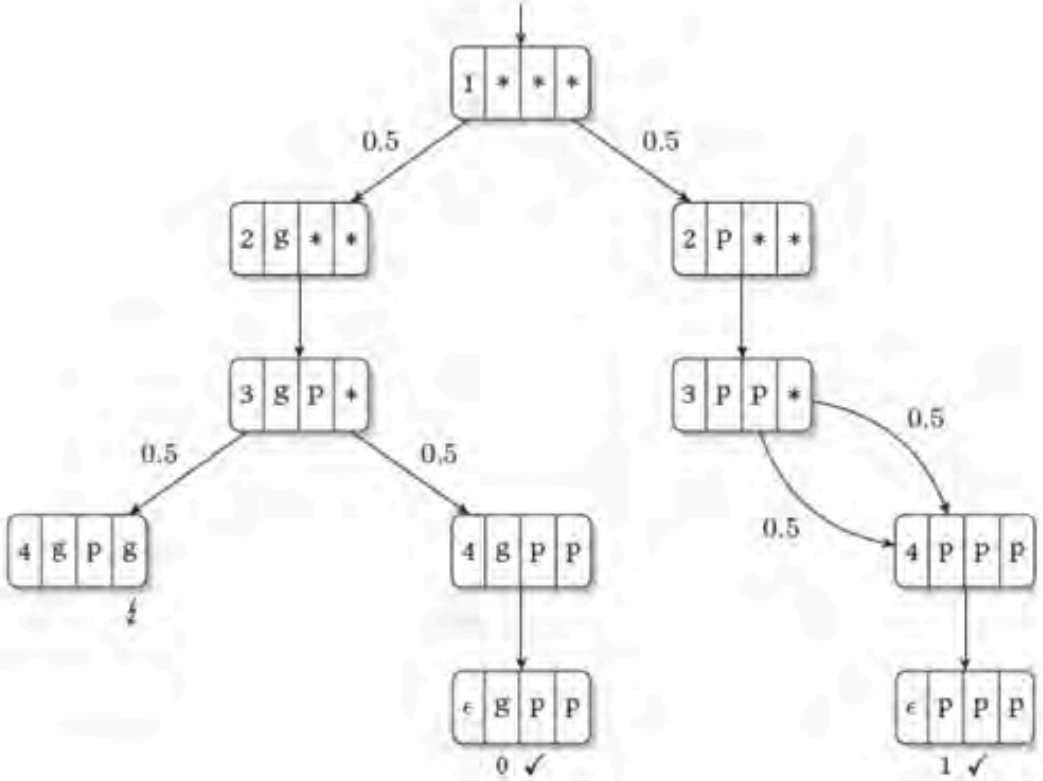
Expected rewards in finite Markov chains can be computed in polynomial time.

The same holds for conditional expected rewards.

[See the lectures by Christel Baier.]

The piranha puzzle

```
f1 := gf [0.5] f1 := pir;
f2 := pir;
s := f1 [0.5] s := f2;
observe (s = pir)
```



What is the probability that the original fish in the bowl was a piranha?

Consider the expected reward of successful termination without violating any observation

$$\mathrm{ER}^{[\![P]\!]}(\sigma_I, \Diamond\langle sink\rangle \cap \neg\Diamond\langle ↯\rangle) = \frac{1\cdot 1/2 + 0\cdot 1/4}{1 - 1/4} = \frac{1/2}{3/4} = 2/3.$$

The piranha program – a wp perspective

```
f1 := gf [0.5] f1 := pir;
f2 := pir;
s := f1 [0.5] s := f2;
observe (s = pir)
```

What is the probability that the original fish in the bowl was a piranha?

$$\mathbb{E}(\texttt{f1 = pir} \mid P \text{ is "feasible"}) = \frac{1 \cdot 1/2 + 0 \cdot 1/4}{1 - 1/4} = \frac{1/2}{3/4} = \frac{2}{3}.$$

Let $cwp(P, f) = \dfrac{wp(P, f)}{wlp(P, \mathbf{1})}$. In fact $cwp(P, f) = (wp(P, f), wlp(P, \mathbf{1}))$.

$wlp(P, \mathbf{1}) = 1 - Pr[P \text{ violates an observation}]$. This includes diverging runs.

Divergence matters

```
abort [0.5] {
   x := 0 [0.5] x := 1;
   y := 0 [0.5] y := 1;
   observe (x = 0 || y = 0)
}
```

Q: What is the probability that `y = 0` on termination?

We: $\dfrac{wp(P, f)}{wlp(P, \mathbf{1})} = \dfrac{2}{7}$

Microsoft's R2: $\dfrac{wp(P, f)}{wp(P, \mathbf{1})} = \dfrac{2}{3}$

In general:

`observe (G)` ≡ `while(!G) skip`

Warning: This is a silly example. Typically divergence comes from loops.

Leave divergence up to the programmer?

Almost-sure termination is "more undecidable" than ordinary termination!

Observations inside loops

These programs are mostly not distinguished as $wp(P_{left}, \mathbf{1}) = wp(P_{right}, \mathbf{1}) = \mathbf{0}$

```
int x := 1;
while (x = 1) {
    x := 1
}
```

- Certain divergence
- $(wp(P_{left}, f), wlp(P_{left}, \mathbf{1})) = (\mathbf{0}, \mathbf{1})$
- Conditional wp $=$ 0

```
int x := 1;
while (x = 1) {
    x := 1 [0.5] x := 0;
    observe (x = 1)
}
```

- Divergence with probability zero
- $\big(wp(P_{right}, f), wlp(P_{right}, \mathbf{1})\big) = (\mathbf{0}, \mathbf{0})$
- Conditional wp $=$ undefined

We do distinguish these programs.

Elementary properties

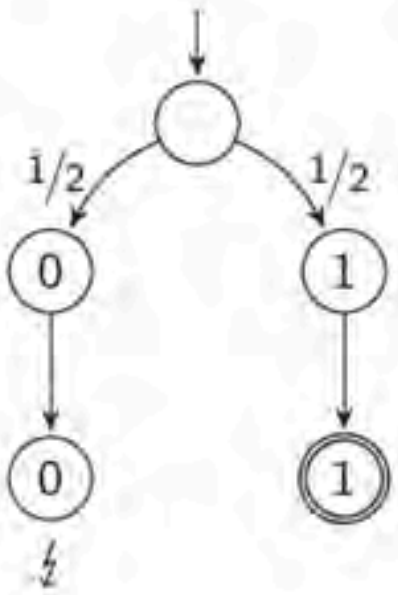
- Monotonicity: $f \leq g$ implies $cwp(P, f) \leq cwp(P, g)$
- Linearity: $cwp(P, \alpha \cdot f + \beta \cdot g) \;=\; \alpha \cdot cwp(P, f) + \beta \cdot cwp(P, g)$
- Duality: $cwlp(P, f) \;=\; \mathbf{1} - cwp(P, \mathbf{1} - f)$
- Zero: $cwp(P, \mathbf{0}) = \mathbf{0}$

Certified using the Isabelle/HOL theorem prover; see [Hölzl, PPS 2016].

Contextual equivalence?

$$P: \quad \{x := 0\}\ [1/2]\ \{x := 1\};\ observe(x = 1)$$

$$Q: \quad \{x := 0;\ observe(x = 1)\}\ [1/2]\ \{x := 1;\ observe(x = 1)\}$$



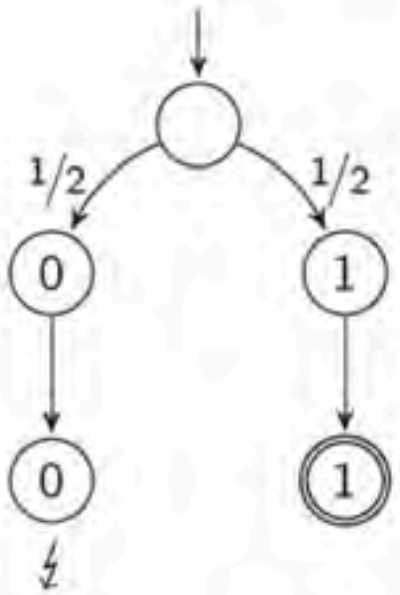
Of course

$$\frac{wp(P, [x = 1])}{wlp(P, 1)} = \frac{wp(Q, [x = 1])}{wlp(Q, 1)} = \frac{1/2}{1/2} = 1$$

Contextual equivalence?

$$P: \quad \{x := 0\}\ [1/2]\ \{x := 1\};\ observe(x = 1)$$

$$Q: \quad \underbrace{\{x := 0;\ observe(x = 1)\}}_{Q_1}\ [1/2]\ \underbrace{\{x := 1;\ observe(x = 1)\}}_{Q_2}$$



Of course

$$\frac{wp(P, [x = 1])}{wlp(P, 1)} = \frac{wp(Q, [x = 1])}{wlp(Q, 1)} = \frac{1/2}{1/2} = 1$$

but we cannot decompose

$$\frac{wp(Q, [x = 1])}{wlp(Q, 1)} \neq 0.5\frac{wp(Q_1, [x = 1])}{wlp(Q_1, 1)} + 0.5\frac{wp(Q_2, [x = 1])}{wlp(Q_2, 1)}$$

This all motivates the definition: $cwp(P, f) = (wp(P, f), wlp(P, \mathbf{1}))$.

Backward compatibility

McIver's wp-semantics is a conservative extension of Dijkstra's wp-semantics.

For any ordinary (aka: GCL) program P and expectation f:

$$\underbrace{wp(P, f)}_{\text{McIver}} \quad = \quad \underbrace{wp(P, [f])}_{\text{Dijkstra}}$$

Our cwp-semantics is a conservative extension of McIver's wp-semantics.

For any observe-free pGCL program P and expectation f:

$$cwp(P, f) \quad = \quad \big(wp(P, f), wlp(P, f)\big)$$

Wp = conditional rewards

For program P and expectation f with $cwp(P, f) = (wp(P, f), wlp(P, \mathbf{1}))$:

$$\frac{wp(P, f)(s)}{wlp(P, \mathbf{1})(s)} \quad = \quad \mathrm{ER}^{[\![P]\!]}\big(s, \Diamond\langle sink\rangle \cap \neg\Diamond\langle \lightning \rangle\big)$$

The ratio of $wp(P, f)$ over $wlp(P, \mathbf{1})$ for input s equals[3] the conditional expected reward to reach a successful terminal state $\langle sink\rangle$ while satisfying all observations in P's MC when starting with s.

Conditional expected rewards in finite MCs can be computed in polynomial time.

[3] Either both sides are equal or both sides are undefined.

Importance of these results

- Unambiguous meaning to (almost) all probabilistic programs
- Operational interpretation to weakest pre-expectations
- Basis for proving correctness
  - of programs
  - of program transformations
  - of program equivalence
  - of static analysis
  - of compilers
  - .....

Removal of conditioning

- Idea: restart an infeasible run until all `observe`-statements are passed
- For program variable `x` use auxiliary variable `sx`
  - store initial value of `x` into `sx`
  - on each new loop-iteration restore `x` to `sx`
- Use auxiliary variable `flag` to signal observation violation:

```
flag := true; while(flag) { flag := false; mprog }
```

- Change `prog` into `mprog` by:
  - `observe(G)` ⇝ `flag := !G || flag`
  - `abort` ⇝ `if(!flag) abort`
  - `while(G) prog` ⇝ `while(G && !flag) prog`

Resulting program

```
sx1,...,sxn := x1,...,xn; flag := true;
while(flag) {
   flag := false;
   x1,...,xn := sx1,...,sxn;
   modprog
}
```

In machine learning, this is known as rejection sampling.

A similar construction was used in Christel Baier's lecture.

Removal of conditioning

the transformation in action:

```
x := 0 [p] x := 1;
y := 0 [p] y := 1;
observe(x != y)
```

```
sx, sy := x, y; flag := true;
while(flag) {
  x, y := sx, sy; flag := false;
  x := 0 [p] x := 1;
  y := 0 [p] y := 1;
  flag := (x = y)
}
```

a simple data-flow analysis yields:

```
repeat {
  x := 0 [p] x := 1;
  y := 0 [p] y := 1
} until(x != y)
```

Removal of conditioning

Correctness of transformation

For program P, transformed program $\hat{P}$, and expectation f:

$$cwp(P, f) \;=\; wp(\hat{P}, f)$$

A dual program transformation

```
repeat
  a0 := 0 [0.5] a0 := 1;
  a1 := 0 [0.5] a1 := 1;
  a2 := 0 [0.5] a2 := 1;
  i := 4*a0 + 2*a1 + a0 + 1
until (1 <= i <= 6)
```

```
a0 := 0 [0.5] a0 := 1;
a1 := 0 [0.5] a1 := 1;
a2 := 0 [0.5] a2 := 1;
i := 4*a0 + 2*a1 + a0 + 1
observe (1 <= i <= 6)
```

Loop-by-observe replacement if there is "no data flow" between loop iterations

Independent and identically distributed loops

iid-Loop

Loop `while (G)P` is iid if and only if for any expectation f:

$$wp(P, [G] \cdot wp(P, f)) \quad = \quad wp(P, [G]) \cdot wp(P, f)$$

Event that G holds after P is independent of the expected value of f after P.

Correctness of transformation

For iid-loop `repeat P until (G)` and expectation f we have:

$$cwp(\texttt{repeat P until (G)}, f) \quad = \quad cwp(\texttt{P ; observe (G)}, f)$$

Loop-free programs are easier to reason about — no loop invariants.

Take-home messages

- Expectations are the quantitative analogue of predicates
- Conditioning involves considering program divergence
- Weakest preconditions correspond to conditional expected rewards
- Conditioning is equivalent to a loop
- wp-Reasoning is "Bayesian inference" on probabilistic programs

Next lecture: flavours of termination on probabilistic programs.

Today's pub quiz: how likely do these programs terminate?

```
x : = 10; while (x > 0) { (x-- [0.5] x++) }
```

```
while (c) { {c := false [0.5] c := true}; x := 2*x} ;
while (x > 0) { x := x-1 }
```

```
call P where
P = skip [0.5] { call P; call P; call P }
```

..... and how long does it take on average till termination?

Overview

On termination

```
bool c := true;
int i := 0;
while (c) {
    i++;
    (c := false [p] c := true)
}
```

This program does not always terminate. It almost surely terminates.

Termination

[Esparza *et al.* 2012]

"[Ordinary] termination is a purely topological property [...], but almost-sure termination is not. [...] Proving almost–sure termination requires arithmetic reasoning not offered by termination provers."

Nuances of termination

..... certain termination

..... termination with probability one

$\Longrightarrow$ almost-sure termination

..... in an expected finite number of steps

$\Longrightarrow$ positive almost-sure termination

..... in an expected infinite number of steps

$\Longrightarrow$ negative almost-sure termination

Certain termination

```
int i : = 100;
while (i > 0) {
    i--;
}
```

This program certainly terminates.

Positive almost-sure termination

For $0 < p < 1$ an arbitrary probability:

```
bool c := true;
int i : = 0;
while (c) {
    i++;
    (c := false [p] c := true)
}
```

This program almost surely terminates. In finite expected time.
Despite the possibility of divergence.

Negative almost-sure termination

Consider the one-dimensional (symmetric) random walk:

```
int x : = 10;
while (x > 0) {
    (x-- [0.5] x++)
}
```

This program almost surely terminates
but requires an infinite expected time to do so.

Compositionality

Consider the two probabilistic programs:

```
int x := 1;
bool c := true;
while (c) {
    c := false [0.5] c := true;
    x := 2*x
}
```

Finite expected termination time

```
while (x > 0) {
    x--
}
```

Finite termination time

Running the right after the left program
yields an infinite expected termination time

Hardness results

Determining expected outcomes is as hard as almost-sure termination.

Almost-sure termination is "more undecidable" than ordinary termination.

Positive almost-sure termination is "even more undecidable".

Universal almost-sure termination is as hard as almost-sure termination.

This does not hold for positive almost-sure termination.

Decision problems

(Positive) almost-sure termination

$(P, s) \in AST$ iff P terminates with probability one on input s

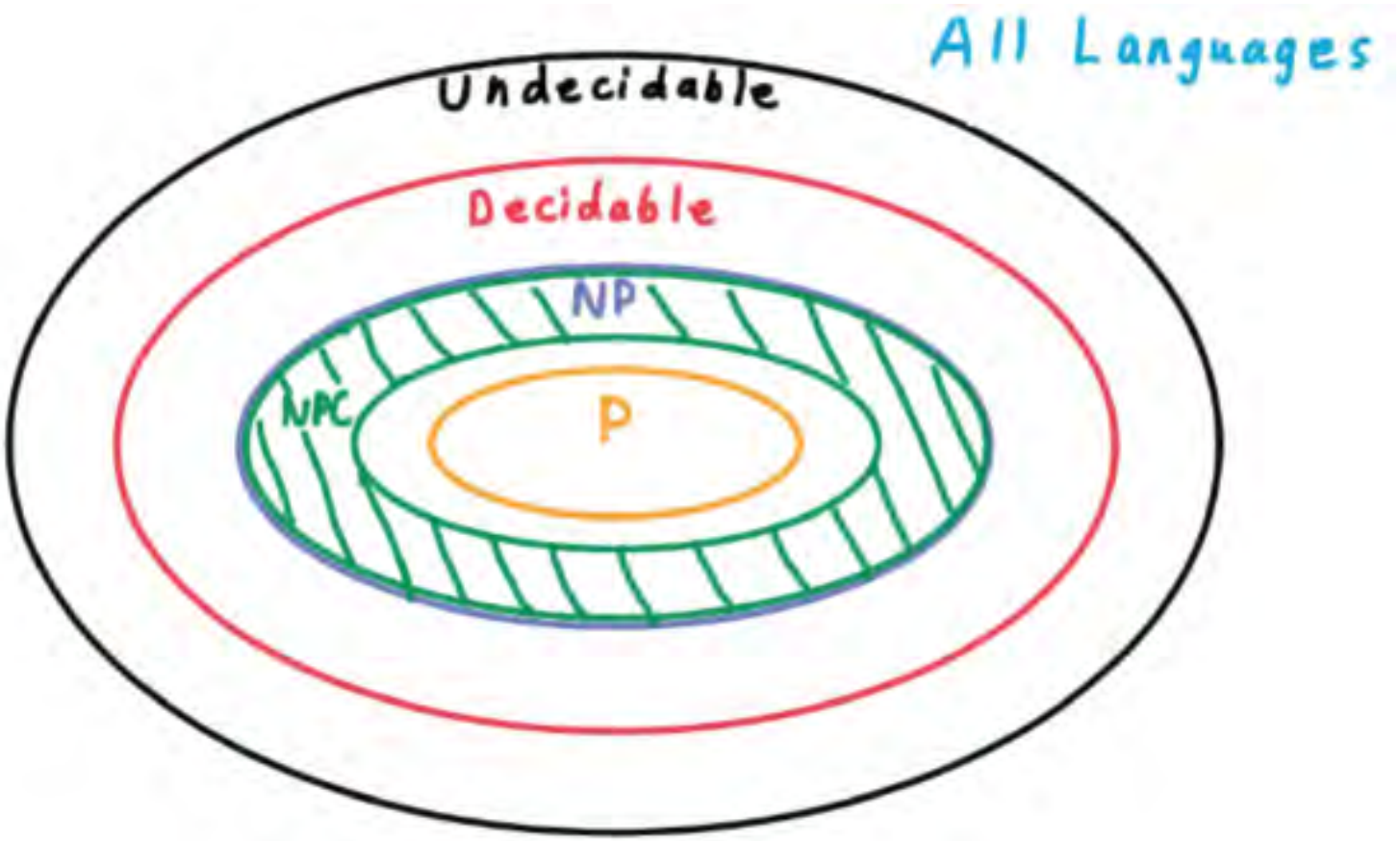
$(P, s) \in PAST$ iff P terminates in finite expected time on input s

The universal versions

$P \in UAST$ iff $\forall s : (P, s) \in AST$

$P \in UPAST$ iff $\forall s : (P, s) \in PAST$

Undecidable versus decidable problems



How can we categorize the undecidable problems?

The arithmetical hierarchy

- Class Σ_n^0 is defined as:

$$\Sigma_n^0 \;=\; \{A \mid A = \{\, x \mid \exists y_1 \forall y_2 \exists y_3 \ldots \forall/\exists y_n : (x, y_1, \ldots, y_n) \in R \,\}\}$$

  where R is a decidable relation.

  Example: the halting problem H is in Σ_1^0. It is semi-decidable.

- Class Π_n^0 is defined as:

$$\Pi_n^0 \;=\; \{A \mid A = \{\, x \mid \forall y_1 \exists y_2 \forall y_3 \ldots \forall/\exists y_n : (x, y_1, \ldots, y_n) \in R \,\}\}$$

  where R is a decidable relation.

  Example: the universal halting problem UH is in Π_2^0.

- Class Δ_n^0 is defined as $\Delta_n^0 = \Sigma_n^0 \cap \Pi_n^0$. Δ_1^0 is the class of decidable problems.

The arithmetical hierarchy is used to classify the degree of undecidability.

Reducibility and completeness

- $B \subseteq X$ is reducible to $A \subseteq X$ iff for some computable function $f : X \to X$ it holds:

$$\forall x \in X. \quad x \in B \quad \text{iff} \quad f(x) \in A$$

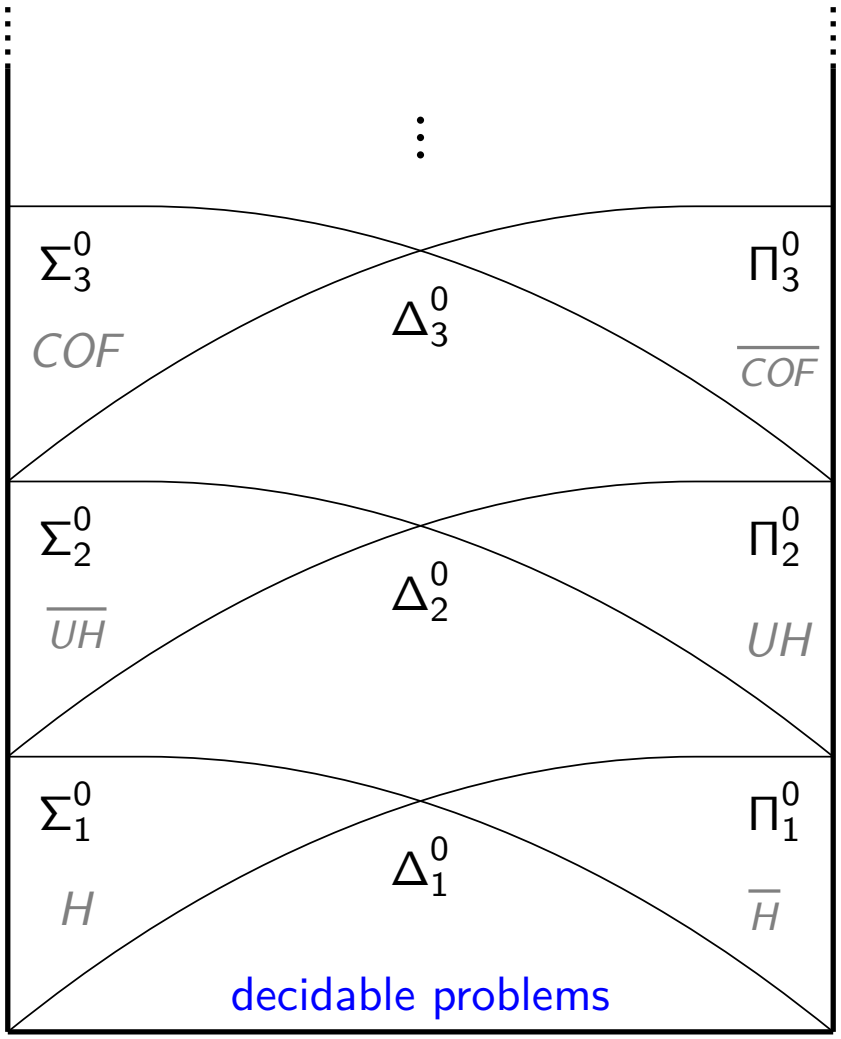
- Decision problem A is Γ^0_n-hard (for $\Gamma \in \{\Sigma, \Pi, \Delta\}$) iff every $B \in \Gamma^0_n$ can be reduced to A.

- Decision problem A is Γ^0_n-complete if $A \in \Gamma^0_n$ and A is Γ^0_n-hard.

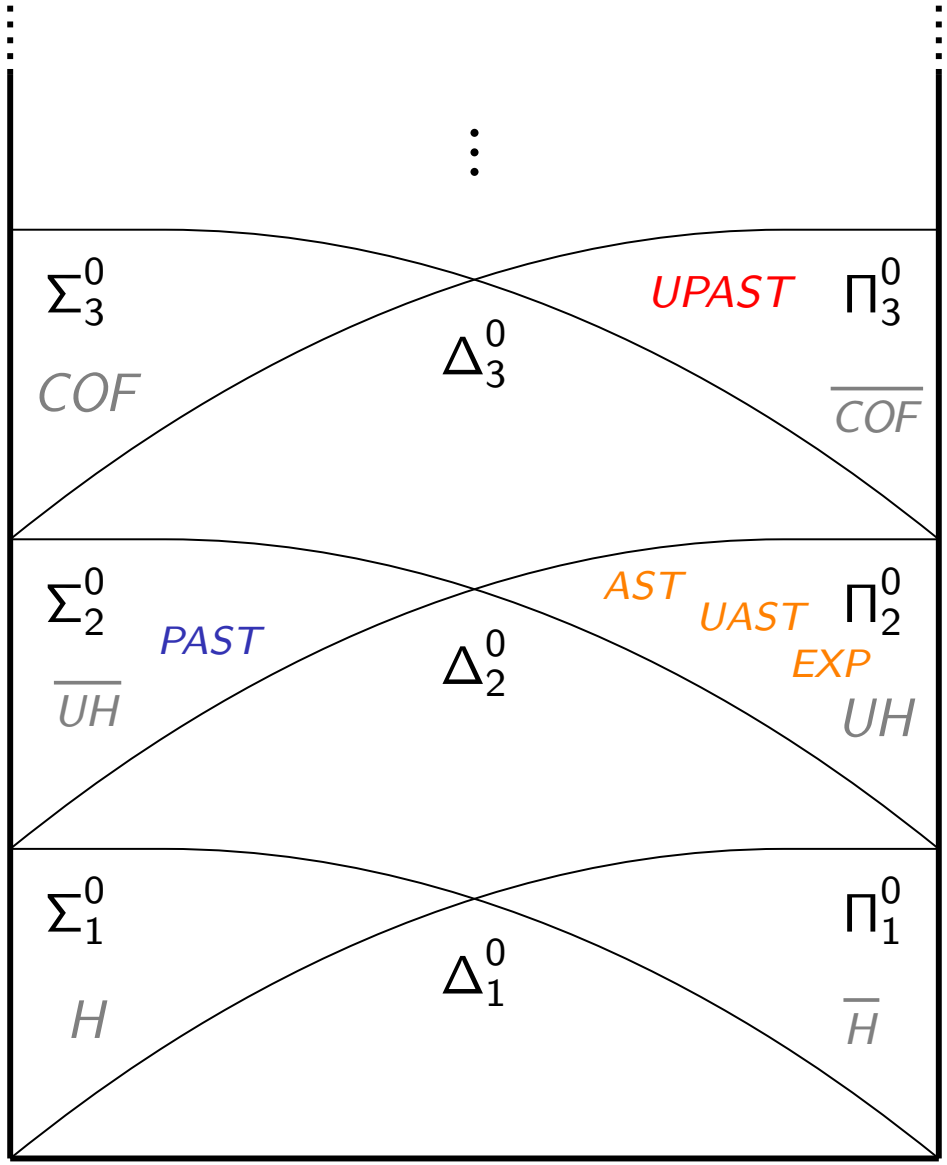
Example: the (universal) halting problem is Σ^0_1- (Π^0_2-) complete.

The bigger picture

The following inclusion diagram holds (all inclusions are strict):



Hardness of almost sure termination



Proof idea: hardness of positive as-termination

Reduction from the complement of the universal halting problem

For an ordinary program Q, provide a probabilistic program P (depending on Q) and an input s, such that

P terminates in a finite expected number of steps on inout s
if and only if
Q does not terminate on some input

Let's start simple

```
bool c := true;
int nrflips := 0;
while (c) {
    nrflips++;
    (c := false [0.5] c := true);
}
```

Expected runtime (integral over the bars):

The `nrflips`-th iteration takes place with probability $1/2^{\texttt{nrflips}}$.

Reducing an ordinary program to a probabilistic one

Assume an enumeration of all inputs for Q is given

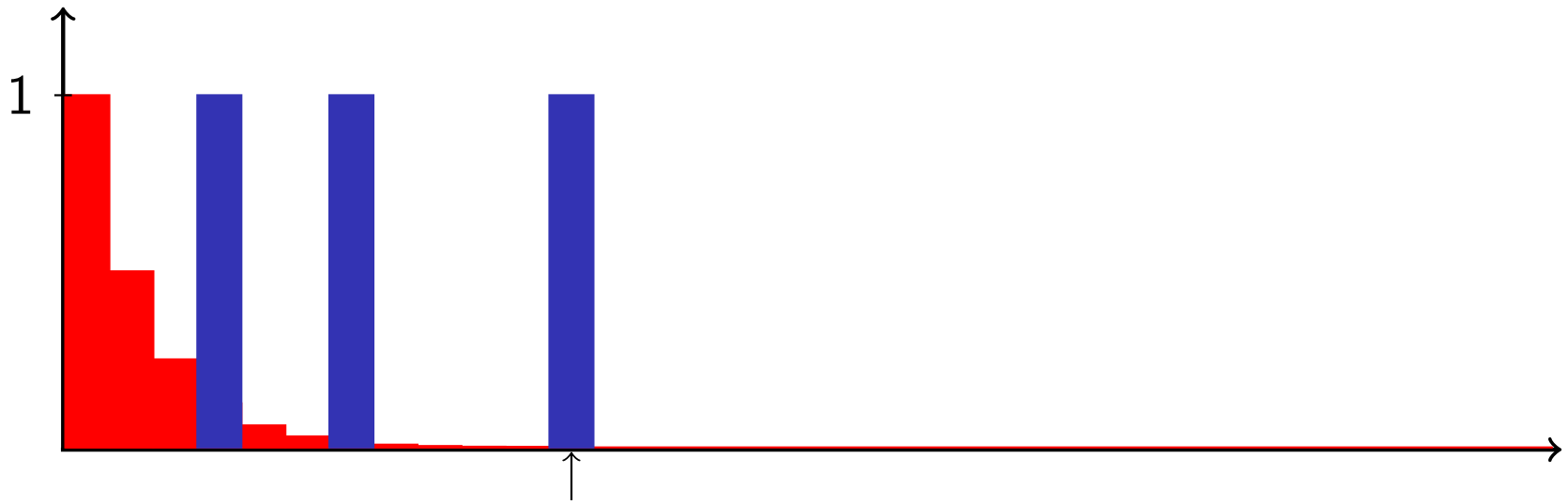
```
bool c := true;
int nrflips := 0;
int i := 0;
while (c) {
    // simulate Q for one (further) step on its i-th input
    if (Q terminates on its i-th input) {
        cheer; // take 2^nrflips effectless steps
        i++;
        // reset simulation of program Q
    }
    nrflips++;
    (c := false [0.5] c := true);
}
```

P looses interest in further simulating Q by a coin flip to decide for termination.

Q does not always halt

Let i be the first input for which Q does not terminate.

Expected runtime of P (integral over the bars):

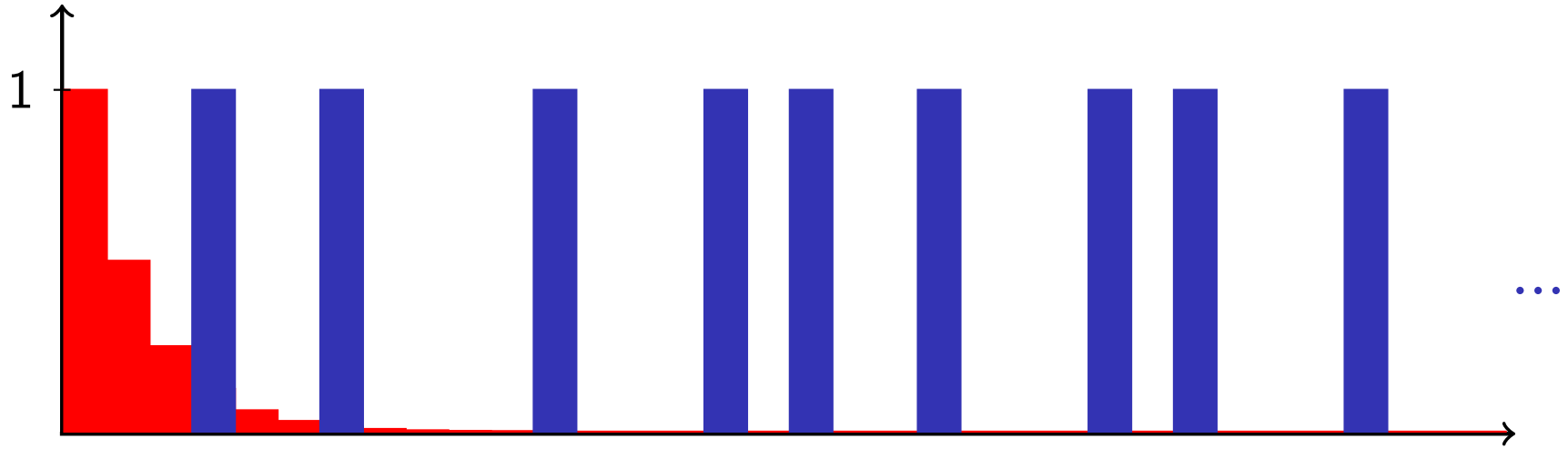


cheering on termination on $(i-1)$-th input

Finite cheering — finite expected runtime

Q terminates on all inputs

Expected runtime of P (integral over the bars):



Infinite cheering — infinite expected runtime

Recursion

Q: What is the probability that recursive program `call P` terminates?

```
P :: skip [0.5] { call P; call P; call P }
```

Recursion

The semantics of recursive procedures is the limit of their n-th inlining:

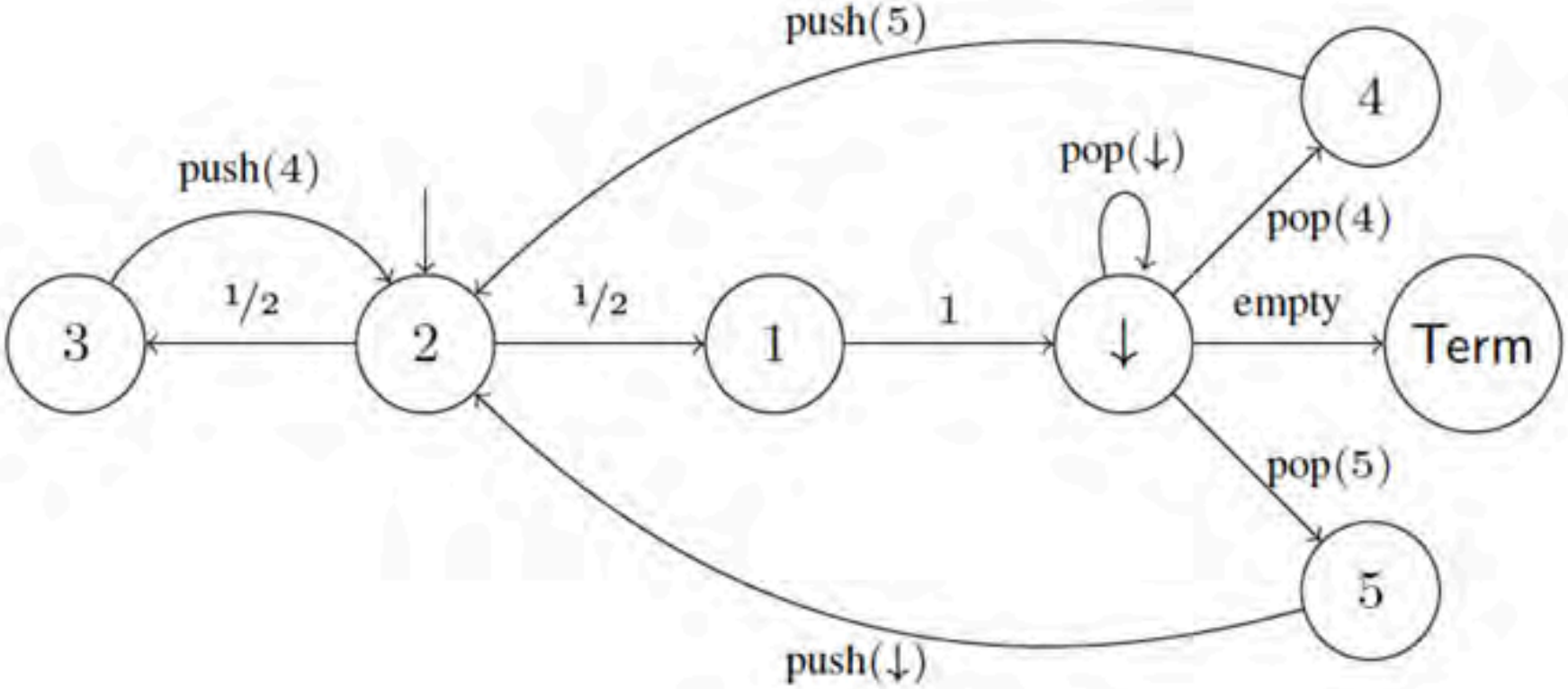
$$\begin{aligned} \texttt{call}_0^D\ P &= \texttt{halt} \\ \texttt{call}_{n+1}^D P &= D(P)[\texttt{call}\ P := \texttt{call}_n^D\ P] \end{aligned}$$

$$wp(\texttt{call}\ P, f)[D] = \sup_n wp(\texttt{call}_n^D\ P, f)$$

where D is the process declaration and $D(P)$ the body of P

This corresponds to the fixed point of a (higher order) environment transformer

Pushdown Markov chains



$$\{\mathsf{skip}^1\}\ [1/2]^2\ \{\mathsf{call}\ P^3;\ \mathsf{call}\ P^4;\ \mathsf{call}\ P^5\ \}$$

Wp = expected rewards in pushdown MCs

For recursive program P and post-expectation f:

$wp(P, f)$ for input s equals the expected reward (that depends on f) to reach a terminal state in the pushdown MC $[\![P]\!]$ when starting with s.

Checking expected rewards in finite-control pushdown MCs is decidable.[4]

[4] see [Brazdil, Esparza, Kiefer, Kucera, FMSD 2013].

Proof rules for recursion

Standard proof rule for recursion:

$$\frac{wp(\texttt{call}\ P, f) \leq g \text{ derives } wp(D(P), f) \leq g}{wp(\texttt{call}\ P, f)[D] \leq g}$$

`call` P satisfies f, g if P's body satisfies it,
assuming the recursive calls in P's body do so too.

Proof rule for obtaining two-sided bounds given $\ell_0 = \mathbf{0}$ and $u_0 = \mathbf{0}$:

$$\frac{\ell_n \leq wp(\texttt{call}\ P, f) \leq u_n \text{ derives } \ell_{n+1} \leq wp(D(P), f) \leq u_{n+1}}{\sup_n \ell_n \ \leq \ wp(\texttt{call}\ P, f)[D] \ \leq \ \sup_n u_n}$$

The golden ratio

Extension with proof rules allows to show e.g.,

```
P :: skip [0.5] { call P; call P; call P }
```

terminates with probability $\frac{\sqrt{5}-1}{2} = \frac{1}{\phi} = \varphi$

Or: apply to reason about Sherwood variants of binary search, quick sort etc.

$$\mathsf{wp}[\mathsf{call}\ P](\mathbf{1}) \preceq \varphi \ \Vdash\ \mathsf{wp}[\mathcal{D}(P_{\mathsf{rec}_3})](\mathbf{1}) \preceq \varphi$$

$$\begin{array}{ll}
 & \mathsf{wp}[\mathcal{D}(P_{\mathsf{rec}_3})](\mathbf{1}) \\
= & \quad \{\text{def. of wp}\} \\
 & \frac{1}{2} \cdot \mathsf{wp}[\mathsf{skip}](\mathbf{1}) + \frac{1}{2} \cdot \mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3};\ \mathsf{call}\ P_{\mathsf{rec}_3};\ \mathsf{call}\ P_{\mathsf{rec}_3}](\mathbf{1}) \\
= & \quad \{\text{def. of wp}\} \\
 & \frac{1}{2} + \frac{1}{2} \cdot \mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3};\ \mathsf{call}\ P_{\mathsf{rec}_3}]\big(\mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3}](\mathbf{1})\big) \\
\preceq & \quad \{\text{assumption, monot. of wp}\} \\
 & \frac{1}{2} + \frac{1}{2} \cdot \mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3}; \mathsf{call}\ P_{\mathsf{rec}_3}](\varphi) \\
= & \quad \{\text{def. of wp, scalab. of wp twice}\} \\
 & \frac{1}{2} + \frac{1}{2}\,\varphi \cdot \mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3}]\big(\mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3}](\mathbf{1})\big) \\
\preceq & \quad \{\text{assumption, monot. of wp}\} \\
 & \frac{1}{2} + \frac{1}{2}\,\varphi \cdot \mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3}](\varphi) \\
= & \quad \{\text{scalab. of wp}\} \\
 & \frac{1}{2} + \frac{1}{2}\,\varphi^2 \cdot \mathsf{wp}[\mathsf{call}\ P_{\mathsf{rec}_3}](\mathbf{1}) \\
\preceq & \quad \{\text{assumption, monot. of wp}\} \\
 & \frac{1}{2} + \frac{1}{2}\,\varphi^3 \\
= & \quad \{\text{algebra}\} \\
 & \varphi
\end{array}$$

$\triangle$

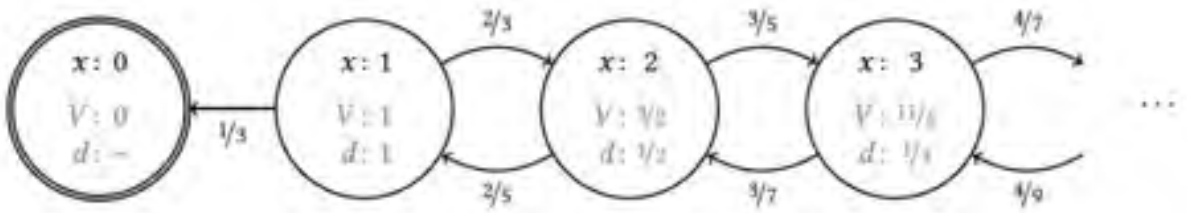
Take-home messages

- Termination for probabilistic programs has different flavours
  - certain, almost-sure termination, and finite expected termination time
- Almost-sure termination for one input is as hard as termination of an ordinary program for all inputs
- Deciding finite expected termination time is "one degree" harder

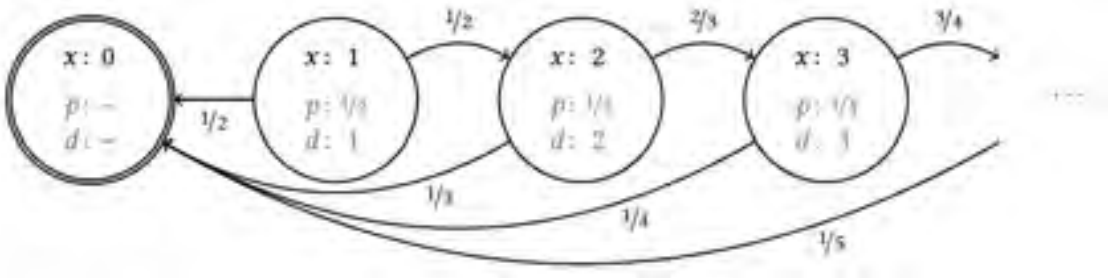
Next lecture: wp-reasoning about (in/finite) expected runtimes.

Today's pub quiz: which programs almost surely terminate?

```
while (x > 0) {
  q := x/(2*x+1);
  x-- [q] x++}
```



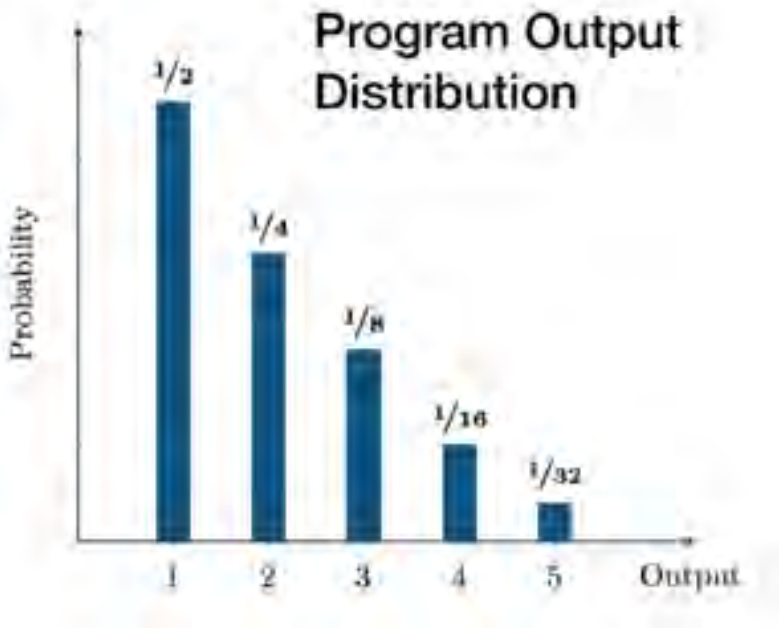

```
while (x > 0) {
  q := 1/x + 1;
  x := 0 [q] x++}
```

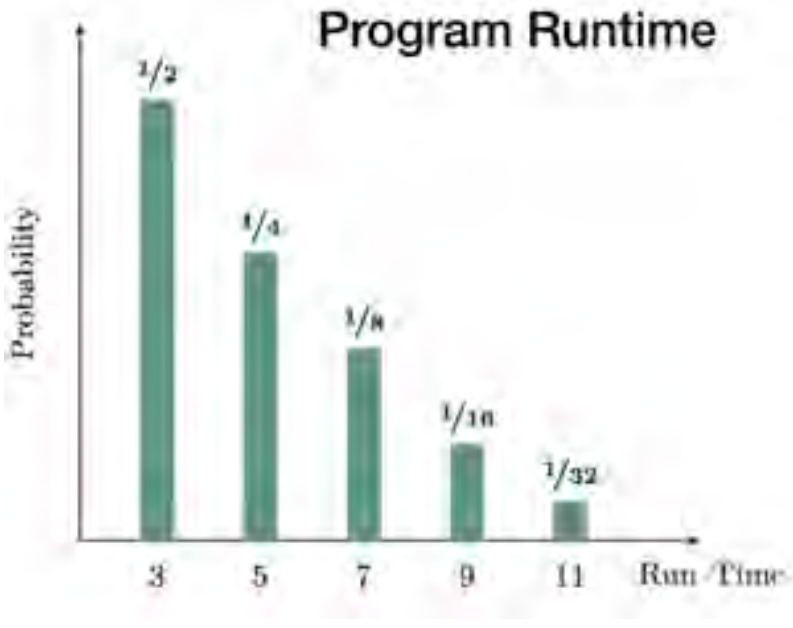


Overview

The runtime of a probabilistic program is random

```
int i := 0;
repeat {i++; (c := false [0.5] c := true)}
until (c)
```





The expected runtime is $3 \cdot 1/2 + 5 \cdot 1/4 + \ldots (2n+1) \cdot 1/2n \;=\; 5$.

Expected runtimes

Aim

Provide a wp-calculus to determine expected runtimes. Why?

1. Prove universal positive almost-sure termination $\Rightarrow \Pi^0_3$-complete
2. Reason about the efficiency of randomised algorithms[5]
3. Reason about the sampling time of a Bayesian network
4. .....

$ert(P, t)$ bounds P's expected runtime if P's continuation takes t time.

[5] Typically by classical probability theory using martingales and expected values.

A naive, unsound approach

```
c := false [0.5] c := true;
for (i := 1; i < 2k; i++) { skip; } // 2k steps
while (c) { skip }
```

Q: what is the expected runtime of this program? A: infinity

Equip the program with a counter `rc` and use standard wp-reasoning.

A naive, unsound approach

```
rc : = 0;
{ c := false [0.5] c := true; } rc++;
for (i := 1; i < 2k; i++) { skip; rc++ } // 2k steps
while (c) { skip; rc++ }
```

wp-reasoning reveals that the expected value of rc is $1/2 \cdot 2\lceil k \rceil + 1/2 \cdot 0 = \lceil k \rceil$.

But the expected runtime is ∞

Expected runtimes

Syntax

- `skip`
- `abort`
- `x := mu`
- `observe (G)`
- `P1 ; P2`
- `if (G) P1 else P2`
- `while(G)P`

Semantics $ert(P, t)$

- $\mathbf{1}+t$
- ∞
- $\mathbf{1} + \lambda\sigma.E_{[\![\mu]\!](\sigma)}(\lambda v.t[x := v](\sigma))$
- $[G]\cdot(\mathbf{1}+t)$
- $ert(P_1, ert(P_2, t))$
- $\mathbf{1} + [G]\cdot ert(P_1, t) + [\neg G]\cdot ert(P_2, t)$
- $\mu X.\mathbf{1} + ([G]\cdot ert(P, X) + [\neg G]\cdot t)$

μ is the least fixed point operator wrt. the ordering $\leq$ on runtimes

and a set of proof rules [6] to get two-sided bounds on runtimes of loops

[6] Certified using the Isabelle/HOL theorem prover; see [Hölzl, ITP 2016].

Backward compatibility

The ert-semantics is a conservative extension of Nielson's rt-semantics.

Soundness

For any ordinary (aka: GCL) program P and predicates E and F:

$$\underbrace{\vdash \{\, E \,\}\ P\ \{\, \Downarrow\ F \,\}}_{\text{total correctness of } P \text{ w.r.t. } F} \qquad \text{implies} \qquad \underbrace{\vdash \{\, E \,\}\ P\ \{\, ert(P, \mathbf{0}) \Downarrow\ F \,\}}_{\text{provable in Nielson's proof system}}$$

A proof in Hoare calculus plus the $ert(P, \mathbf{0})$ yields a proof in Nielson's calculus.

Completeness

For any ordinary (aka: GCL) program P and predicates E and F:

$$\underbrace{\vdash \{\, E \,\}\ P\ \{\, \mathrm{Nrt} \Downarrow\ F \,\}}_{\text{proof in Nielson's proof system}} \qquad \text{implies} \qquad ert(P, \mathbf{0})(s) \ \leq\ k \cdot [\![\, \mathrm{Nrt} \,]\!](s)$$

for some $k \in \mathbb{N}$ and all program states s satisfying E.

Proof rules for loops

Let n be a natural and let $\texttt{while}(G)\,P$ be our loop.

Runtime transformer I_n is a lower ω-invariant w.r.t. t iff

$$I_0 \preceq F_t(\mathbf{0}) \quad \text{and} \quad I_{n+1} \preceq F_t(I_n) \text{ for all } n$$

where $F_t(X) = \mathbf{1} + ([G] \cdot ert(P, X) + [\neg G] \cdot t)$.

In a similar way, upper ω-invariants w.r.t. t are defined.

If I_n is a lower ω-invariant w.r.t. t and $\lim_{n\to\infty} I_n$ exists, then:

$$\lim_{n\to\infty} I_n \preceq ert(\texttt{while}(G)\,P, t)$$

Upper ω-invariants provide an upper bound on the loop's run time.

Completeness: such lower- and upper ω-invariants always exist.

Invariant synthesis

```
while (x > 0) { x := x-1 }
```

It is easy to check that a lower ω-invariant is:

$$J_n \;=\; \mathbf{1} \;+\; \underbrace{[0 < x < n]\cdot 2x}_{\text{on iteration}} \;+\; \underbrace{[x \geq n]\cdot(2n-1)}_{\text{on termination}}$$

Using our result we obtain that

$$\lim_{n\to\infty}\big(\mathbf{1} + [0 < x < n]\cdot 2x + [x \geq n]\cdot(2n-1)\big) \;=\; \mathbf{1} + [x > 0]\cdot 2x$$

is a lower bound on the runtime of the above program.

Invariant synthesis

```
while (c) { {c := false [0.5] c := true}; x := 2*x} ;
while (x > 0) { x := x-1 }
```

Template for a lower ω-invariant:

$$I_n \;=\; \mathbf{1} \;+\; \underbrace{[c \neq 1]\cdot(\mathbf{1} + [x > 0]\cdot 2x)}_{\text{on termination}} \;+\; \underbrace{[c = 1]\cdot(a_n + b_n \cdot [x > 0]\cdot 2x)}_{\text{on iteration}}$$

The constraints on being a lower ω-invariant yield:

$$a_0 \;\leq\; 2 \quad \text{and} \quad a_{n+1} \;\leq\; {}^{7}\!/_{2} + {}^{1}\!/_{2}\cdot a_n \quad \text{and} \quad b_0 \;\leq\; 0 \quad \text{and} \quad b_{n+1} \;\leq\; 1 + b_n$$

This admits the solution $a_n = 7 - {}^{5}\!/_{2^n}$ and $b_n = n$. Then: $\lim_{n\to\infty} I_n \;=\; \infty$.

Coupon collector's problem

ON A CLASSICAL PROBLEM OF PROBABILITY THEORY

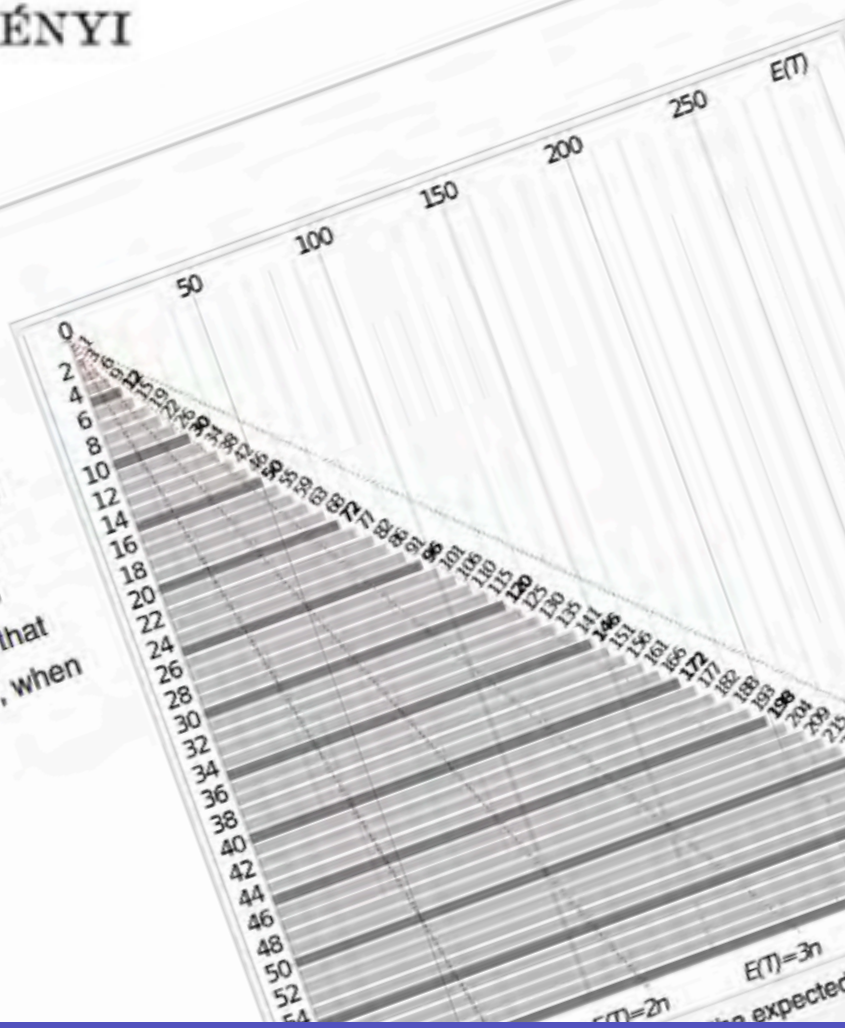
by

P. ERDŐS and A. RÉNYI

Coupon collector's problem

From Wikipedia, the free encyclopedia

In probability theory, the **coupon collector's problem** describes the "collect all coupons and win" contests. It asks the following question: Suppose that there is an urn of n different coupons, from which coupons are being collected, equally likely, with replacement. What is the probability that more than t sample trials are needed to collect all n coupons? An alternative statement is: Given n coupons, how many coupons do you expect you need to draw with replacement before having drawn each coupon at least once? The mathematical analysis of the problem reveals that the expected number of trials needed grows as $\Theta(n \log(n))$.[1] For example, when about 225[2] trials to collect all 50 coupons.



Coupon collector's problem

A more modern phrasing:

Each box of cereal contains one (equally likely) out of N coupons.
You win a price if all N coupons are collected.
How many boxes of cereal need to be bought on average to win?

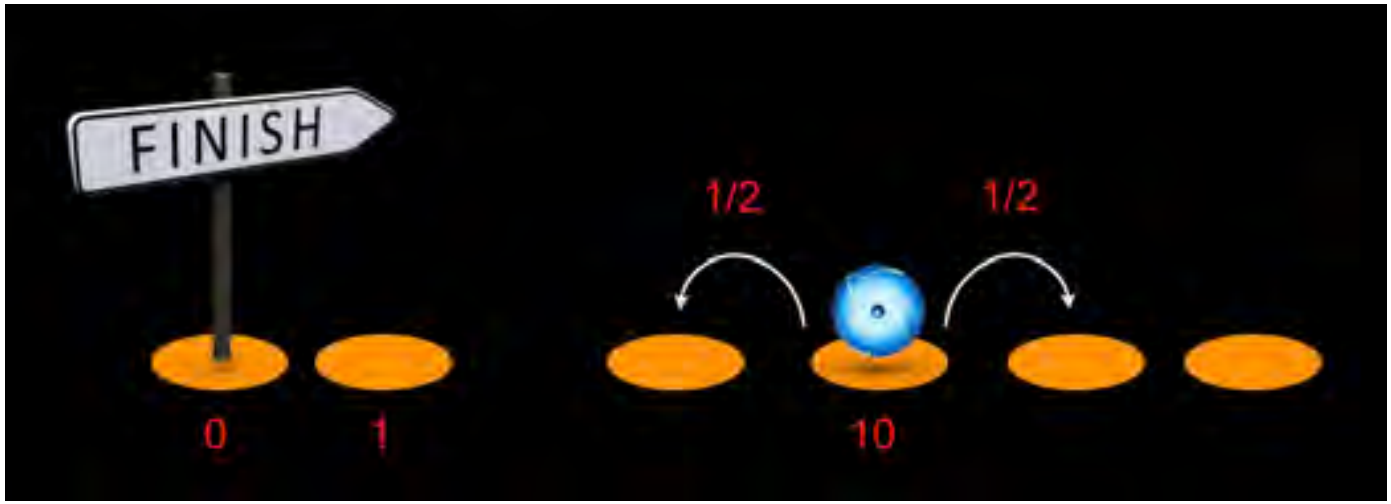
Coupon collector's problem

```
cp := [0,...,0]; // no coupons yet
i := 1; // coupon to be collected next
x := 0: // number of coupons collected
while (x < N) {
   while (cp[i] != 0) {
       i := uniform(1..N) // next coupon
   }
   cp[i] := 1;  // coupon i obtained
   x++;  // one coupon less to go
}
```

Using our ert-calculus one can prove that expected runtime is $\Theta(N \cdot \log N)$.

By systematic formal verification à la Floyd-Hoare. Machine checkable.

Random walk

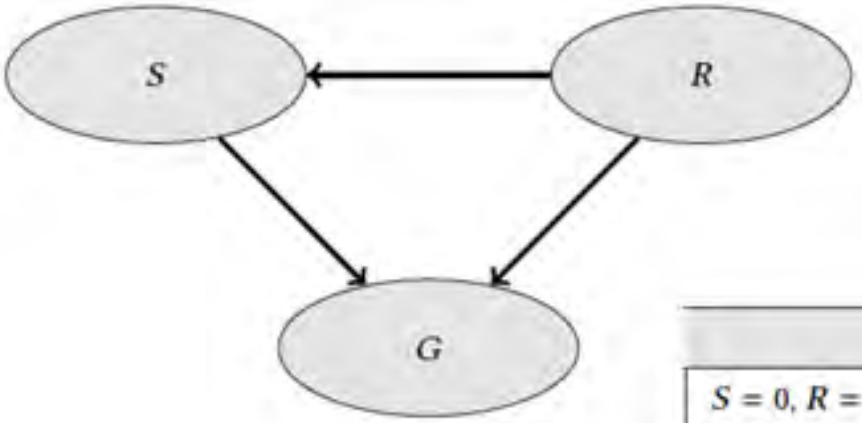


Using our ert-calculus one can prove that its expected runtime is ∞.
By systematic formal verification à la Floyd-Hoare. Machine checkable.

Overview

A toy Bayesian network

	$S=0$	$S=1$
$R=0$	a	$1-a$
$R=1$	0.2	0.8



$R=0$	$R=1$
a	$1-a$

	$G=0$	$G=1$
$S=0, R=0$	0.01	0.99
$S=0, R=1$	0.25	0.75
$S=1, R=0$	0.9	0.1
$S=1, R=1$	0.2	0.8

Note that this BS is parametric (in a)

How often do we need to sample this BN under the evidence that $G = 0$?

This is inference. Exact inference is NP-hard. Mostly sampling is used.

Recall: rejection sampling

For a given Bayesian network and some evidence:

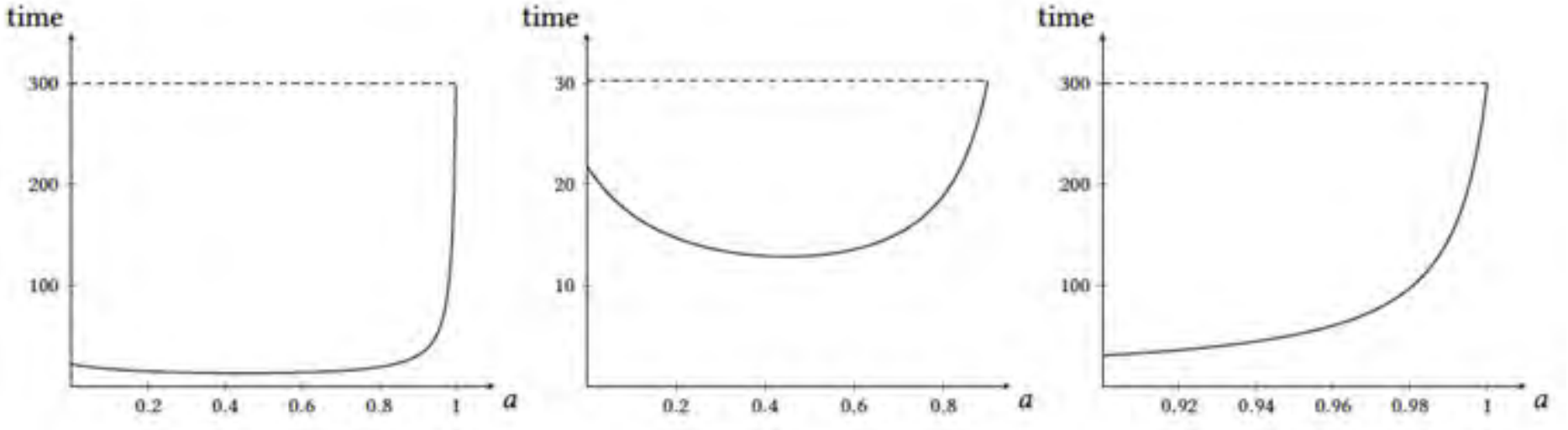
- Sample from the joint distribution described by the Bayesian network
- If the sample complies with the evidence, accept the sample; if not, repeat sampling.

If this procedure is applied N times, this results in N iid-samples.

Q: How many samples do we need on average for a single iid-sample?

Sampling time for example BN

Rejection sampling requires $\dfrac{200a^2 - 40a - 460}{89a^2 - 69a - 21}$ samples:



Decomposition theorem

For every pGCL program P and expectation f:

$$ert(P, f) \;=\; ert(P, \mathbf{0}) + wp(P, f)$$

Independent and identically distributed loops

iid-Loop

Loop `while(G)P` is iid if and only if for any expectation f:

$$wp(P, [G] \cdot wp(P, f)) \quad = \quad wp(P, [G]) \cdot wp(P, f)$$

Event that G holds after P is independent of the expected value of f after P.

Weakest precondition of iid-loop

wp-semantics of iid-loops

For iid-loop `while(G)P` we have for every expectation f and state s:

1. If $wp(P,[G])(s) < 1$, then:

$$wp(\texttt{while(G)P}, f)(s) \;=\; [G](s) \cdot \frac{wp(P,[\neg G]\cdot f)(s)}{1 - wp(P,[G])(s)} + [\neg G](s) \cdot f(s)$$

2. If $wp(P,[G])(s) = 1$, then

$$wp(\texttt{while(G)P}, f)(s) \;=\; [\neg G](s) \cdot f(s)$$

Proof: use $wp(while_n(G)P, f) = [G]\cdot wp(P,[\neg G]\cdot f) \cdot \sum_{i=0}^{n-2}\left(wp(P,[G])^i\right) + [\neg G]\cdot f$

No loop invariant, martingale, or metering function needed. Fully automatable.

Expected runtime of iid-loop

ert of iid-loops

For almost-surely terminating iid-loop `while(G)P` for which every iteration runs in the same expected time, we have for every expectation f:

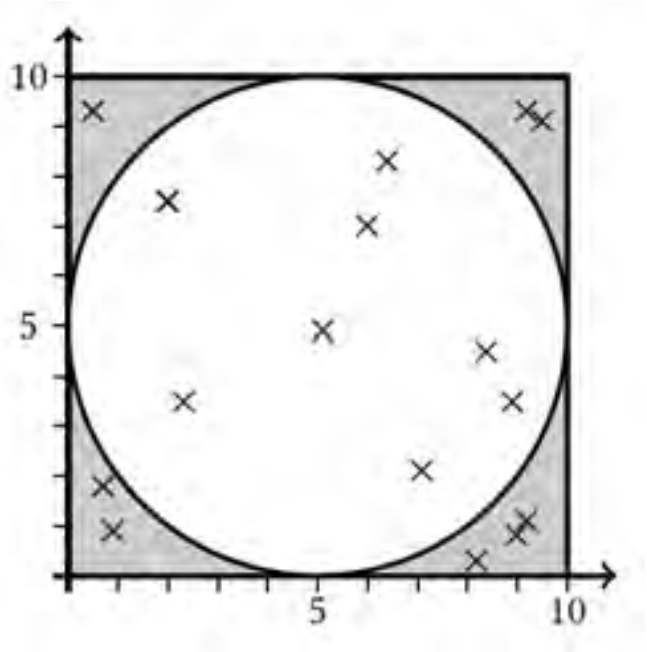
$$ert(\texttt{while(G)P}, f) \;=\; \mathbf{1} + [G] \cdot \frac{\mathbf{1} + ert(P, [\neg G] \cdot f)}{1 - wp(P, [G])} + [\neg G](s) \cdot f$$

where we define $0/0 := 0$ and $a/0 := \infty$ for $a \neq 0$.

No loop invariant, martingale, or metering function needed. Fully automatable.

Example: sampling within a circle

```
while ((x-5)**2 + (y-5)**2 >= 25){
    x := uniform(0..10);
    y := uniform(0..10)
}
```



It is easy to check that this loop is iid, almost surely terminating, and every iteration runs in the same expected time. Our theorem then yields:

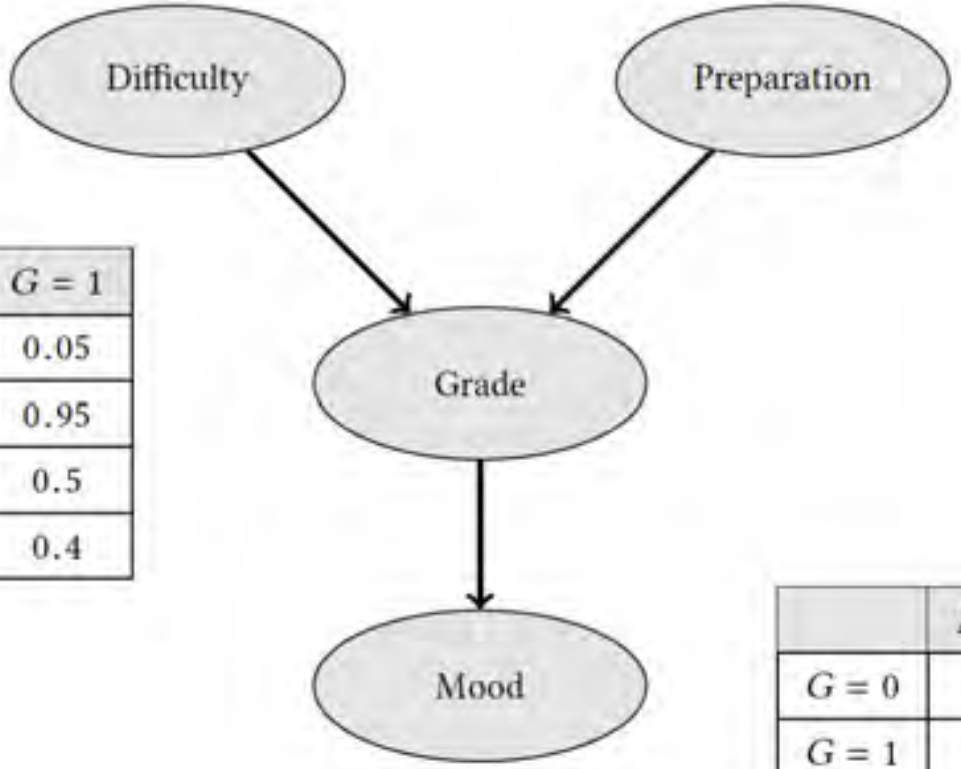
$$ert(P_{circle}, \mathbf{0}) \;=\; \mathbf{1} + [(x-5)^2 + (y-5)^2 \geq 25] \cdot \frac{363}{73}$$

So: $1 + {}^{363}/_{73} \approx 5.97$ operations are required on average using rejection sampling

Student's mood after an exam

$D = 0$	$D = 1$
0.6	0.4

$P = 0$	$P = 1$
0.7	0.3



	$G = 0$	$G = 1$
$D = 0, P = 0$	0.95	0.05
$D = 1, P = 1$	0.05	0.95
$D = 0, P = 1$	0.5	0.5
$D = 1, P = 0$	0.6	0.4

	$M = 0$	$M = 1$
$G = 0$	0.9	0.1
$G = 1$	0.3	0.7

How likely does a well-prepared student end up with a bad mood?

Programs for Bayesian networks

- Take a topological sort of the BN's vertices, e.g., $D; P; G; M$

- Map each conditional probability table (aka: node) to a program, e.g.:

```
if (xD = 0 && xP = 0) {
   xG := 0 [0.95] xG := 1
   } else if (xD = 1 && xP = 1) {
   xG := 0 [0.05] xG := 1
   } else if (xD = 0 && xP = 1) {
   xG := 0 [0.5] xG := 1
   } else if (xD = 1 && xP = 0) {
   xG := 0 [0.6] xG := 1
}
```

	$G = 0$	$G = 1$
$D = 0, P = 0$	0.95	0.05
$D = 1, P = 1$	0.05	0.95
$D = 0, P = 1$	0.5	0.5
$D = 1, P = 0$	0.6	0.4

- Condition on the evidence, e.g., for $P = 1$ we get:

```
repeat { progD ; progP; progG ; progM } until (P=1)
```

Soundness

Correctness of BN programs

For BN B over variables V with evidence obs over $O \subseteq V$ and value $\underline{v}$ for node (and input) v:

$$wp(\texttt{prog}(B, obs), \left(\bigwedge_{v \in V \setminus O} x_v = \underline{v}\right)) = \frac{Pr\left(\bigwedge_{v \in V \setminus O} v = \underline{v} \wedge \bigwedge_{o \in O} o = \underline{o}\right)}{Pr\left(\bigwedge_{o \in O} o = \underline{o}\right)}$$

where $\texttt{prog}(B, obs)$ equals **repeat** $progB$ **until** $\left(\bigwedge_{o \in O} x_o = obs(o)\right)$.

Ergo: exact Bayesian inference can be done by wp-reasoning, e.g.,

$$wp(P_{mood}, [x_D = 0 \wedge x_G = 0 \wedge x_M = 0]) = \frac{Pr(D = 0, G = 0, M = 0, P = 1)}{Pr(P = 1)} \approx 0.27$$

The expected runtime of a Bayesian network

Every BN program is an iid-loop[7], terminates almost surely and each iteration takes the same expected time

Expected runtime of BN programs

For every runtime t we have:

$$ert(\underbrace{\texttt{repeat Seq until (G)}}_{\text{program of the BN}}, t) \quad = \quad \frac{\mathbf{1} + ert(\mathrm{Seq}, [G]\cdot t)}{wp(\mathrm{Seq}, [G])}$$

Seq is a sequence of blocks, where a block corresponds to a single BN vertex.

For any BN, a closed-form for its expected runtime can be obtained compositionally.

$$ert(\underbrace{\texttt{repeat D; P; G; M until (P=1)}}_{\text{program of student mood's BN}}, \mathbf{0}) \; = \; \frac{\mathbf{1} + ert(D; P; G; M, \mathbf{0})}{wp(D; P; G; M, [P = 1])} \quad \approx \quad 23.46$$

[7] A mild variant of iid for a single runtime, which can be checked syntactically.

Overview

nature Perspective

"There are several reasons why probabilistic programming could prove to be revolutionary for machine intelligence and scientific modelling."

REVIEW

doi:10.1038/nature14541

Probabilistic machine learning and artificial intelligence

Zoubin Ghahramani[1]

Epilogue

Take-home message

- Connecting wp and operational semantics
- Semantic intricacies of conditioning
- Almost-sure termination is harder than termination
- Expected runtime analysis (e.g., Bayesian networks)

Extensions

- Non-determinism
- Mixed-sign random variables
- Link to Bayesian networks
- Invariant synthesis

Further reading

- JPK, A. McIver, L. Meinicke, and C. Morgan.
  Linear-invariant generation for probabilistic programs. SAS 2010.
- F. Gretz, JPK, and A. McIver.
  Operational versus wp-semantics for pGCL. J. on Performance Evaluation, 2014.
- F. Gretz *et al.*.
  Conditioning in probabilistic programming. MFPS 2015.
- B. Kaminski, JPK.
  On the hardness of almost-sure termination. MFCS 2015.
- B. Kaminski, JPK.
  A wp-semantics for mixed-sign expectations. LICS 2017.
- B. Kaminski, JPK, C. Matheja, and F. Olmedo.
  Expected runtime analysis of probabilistic programs [8]. ESOP 2016.
- F. Olmedo, B. Kaminski, JPK, C. Matheja.
  Reasoning about recursive probabilistic programs. LICS 2016.

[8] Recipient EATCS best paper award of ETAPS 2016.