Discrete Probabilistic Programming from First Principles

Guy Van den Broeck

The Fourth International Workshop on
Declarative Learning Based Programming (DeLBP)

Aug 11, 2019

What are probabilistic programs?

What is the formal semantics?

How to do exact inference?

What about approximate inference?

References

- Steven Holtzen, Todd Millstein and Guy Van den Broeck. Symbolic Exact Inference for Discrete Probabilistic Programs, *In Proceedings of the ICML Workshop on Tractable Probabilistic Modeling (TPM)*, 2019.
- Tal Friedman and Guy Van den Broeck. Approximate Knowledge Compilation by Online Collapsed Importance Sampling, *In Advances in Neural Information Processing Systems 31 (NeurIPS)*, 2018.
- Steven Holtzen, Guy Van den Broeck and Todd Millstein. Sound Abstraction and Decomposition of Probabilistic Programs, *In Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.
- Steven Holtzen, Todd Millstein and Guy Van den Broeck. Probabilistic Program Abstractions, *In Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI)*, 2017.

…with slides stolen from Steven Holtzen and Tal Friedman.

What are probabilistic programs?

What are probabilistic programs?

```
x ~ flip(0.5);
y ~ flip(0.7);
z := x || y;
if(z) {
    …
}
observe(z);
```

means “flip a coin, and output true with probability ½”

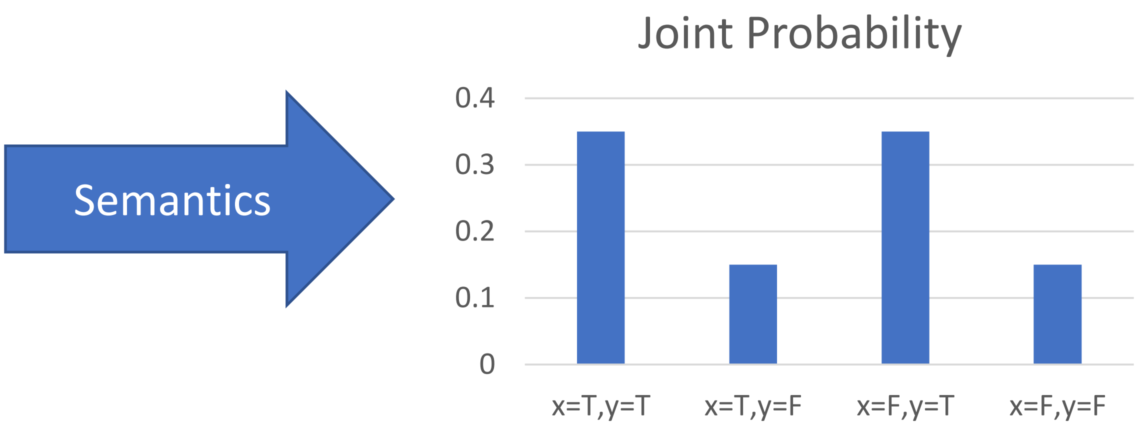
Standard programming language constructs

means “reject this execution if z is not true”

Semantics of a Probabilistic Program

A *probability distribution* on its states

```
x ~ flip(0.5);
y ~ flip(0.7);
```



Goal: To perform *probabilistic inference*

- Compute the probability of some event
- Can be used for *Bayesian machine learning*: compute posterior (learned) parameters/structure given data

Why Probabilistic Programming?

- PPLs have grown in popularity: there are dozens

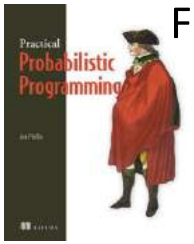
Edward

Pyro

Stan

Venture, Church

Figaro



ProbLog, PRISM, LPADs, CPLogic, ICL, PHA, etc.

- They are popular with practitioners
  - Specify a probability model in a familiar language
  - Expressive and concise
  - Cleanly separates model from inference

The Challenge of PPL Inference

Most popular inference algorithms are **black box**

– Treat program as a map from inputs to outputs

Pyro Stan Edward

(black-box variational, Hamiltonian MC)

– Simplifying assumptions: differentiability, continuity

– Little to no effort to exploit program structure (automatic differentiation aside)

– Approximate inference ☹

Why Discrete Models?

1. Real programs have inherent discrete structure (e.g. if-statements)

2. Discrete structure is important in modeling (graphs, topic models, etc.)

3. Many existing systems assume smooth and differentiable densities:

   Discrete probabilistic programming is the important unsolved open problem!

What is the formal semantics?

Simple Discrete PPL Syntax

(statements and expressions)

```
s ::=
  | s; s
  | x := e
  | x ~ flip(θ)
  | if e { s } else { s }
  | observe(e)
  | skip
e :: =
  | x
  | T | F
  | e ∨ e
  | e ∧ e
  | ¬ e
```

Semantics

- The *program state* is a map from *variables* to *values*, denoted σ
- The goal of our semantics is to associate
  - statements in the syntax with
  - a probability distribution on states
- Notation: semantic brackets [[s]]

Sampling Semantics

- The simplest way to give a semantics to our language is to *run the program infinite times*

```
x ~ flip(0.5);
```

Draw samples

σ
x=true
x=false
x=true
x=false

- The probability distribution of the program is defined as the *long run average* of how often it ends in a particular state

Semantics of

```
x ~ flip(0.5);
y ~ flip(0.7);
```

x = true y = true	x = false y = true
ω_1 `0.5*0.7 = 0.35`	ω_2 `0.5*0.7 = 0.35`
x = false y = false	x = true y = false
ω_3 `0.5*0.3 = 0.15`	ω_4 `0.5*0.3 = 0.15`

Semantics of

```
x ~ flip(0.5);
y ~ flip(0.7);
observe(x || y);
```

x = true
y = true

ω_1

0.5*0.7 = 0.

x = false
= true

ω_2

0.7 = 0.35

= fals
y alse

ω_3

0.3 =

= true
y = false

ω_4

0.5*0.3 = 0.15

Semantics: Throw away all executions that do not satisfy the condition x || y.

Rejection Sampling Semantics

- Observes give a *posterior distribution* on the program states
- Semantics of a program: draw (infinite) samples, take the long run average over *accepted samples*

```
x ~ flip(0.5);
y ~ flip(0.7);
observe(x || y);
```

Draw samples

σ	
x=true	y=true
~~x=false~~	~~y=false~~
x=true	y=false
x=false	y=true

Rejection Sampling Semantics

☺

- Extremely general: you only need to be able to run the program to implement a rejection-sampling semantics
- This how most AI researchers think about the meaning of their programs (?)

☹

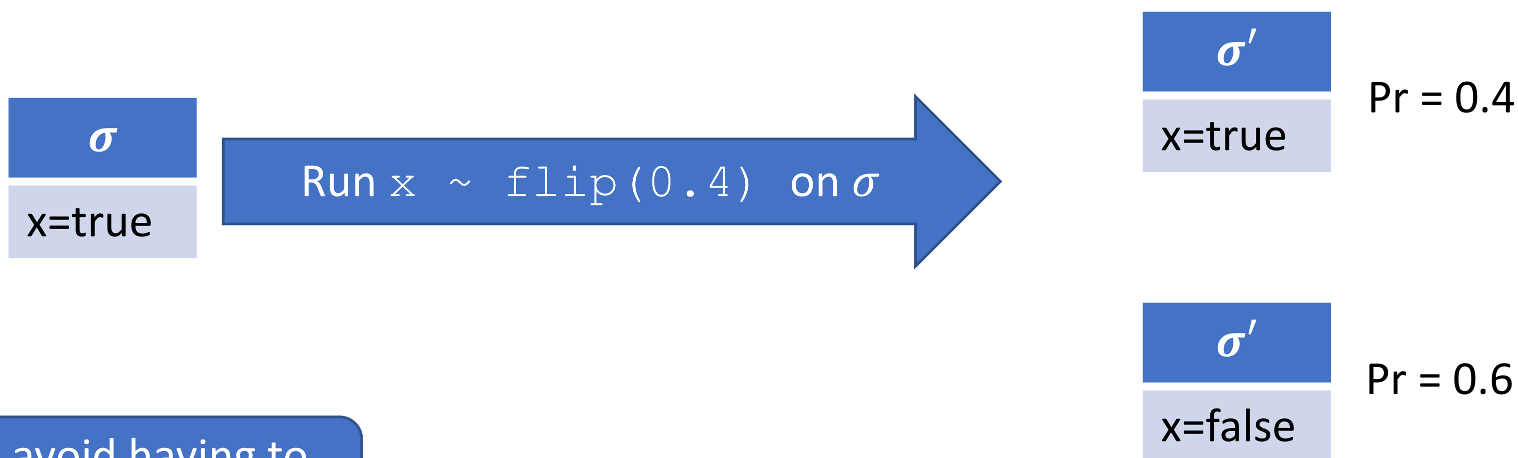
- “Procedural”: the meaning of the program is whatever it executes to …not entirely satisfying…
- A sample is a full execution: a global property that makes it harder to think modularly about local meaning of code

Next: the gold standard in programming languages

denotational semantics

Denotational Semantics

- Idea: We don't have to *run* a flip statement to know what its distribution is
- For some input state σ and output state σ', we can directly compute the *probability of transitioning* from σ to σ' upon executing a flip statement:



We can avoid having to think about sampling!

Denotational Semantics of Flip

Idea: Directly define the probability of transitioning upon executing each statement

Call this its *denotation,* written $[\![\mathtt{s}]\!]$

$$[\![x \sim \mathtt{flip}(\theta)]\!](\sigma' \mid \sigma) \triangleq \begin{cases} \theta & \text{if } \sigma' = \sigma[x \mapsto T] \\ 1 - \theta & \text{if } \sigma' = \sigma[x \mapsto F] \\ 0 & \text{otherwise} \end{cases}$$

Semantic bracket: associate semantics with syntax

Output state

Input State

Assign x to false in the state σ

Semantics of Expressions

- What about x := e?
- Need semantics for expressions: simple
- Just evaluate the expression *e* on state σ

$$[\![\underbrace{\mathtt{b} \vee \mathtt{c}}_{e}]\!](\underbrace{\{b \mapsto T, c \mapsto F\}}_{\sigma}) = T$$

Semantics of Assignments

What about x := e?

$$[\![\mathtt{x} := \mathsf{e}]\!](\sigma' \mid \sigma) \triangleq \begin{cases} 1 & \sigma' = \sigma[x \mapsto [\![\mathsf{e}]\!](\sigma)] \\ 0 & \text{otherwise} \end{cases}$$

(semantics of if-then-else
also based on if-test expression)

Semantics of Sequencing

- Assume the program has no observe statements
- We can compute the denotation of sequencing by *marginalizing out the intermediate state*

$$[\![s_1; s_2]\!](\sigma' \mid \sigma) = \sum_{\tau} [\![s_1]\!](\sigma \mid \tau) \times [\![s_2]\!](\sigma' \mid \tau)$$

Example: $[\![x \sim \texttt{flip}(0.4); y \sim \texttt{flip}(0.1)]\!](\{x \mapsto T, y \mapsto F\} \mid \emptyset)$

$$= \sum_{\tau \in \{\{x \mapsto T\}, \{x \mapsto F\}\}} [\![x \sim \texttt{flip}(0.4)]\!](\tau \mid \emptyset) \times [\![y \sim \texttt{flip}(0.1)]\!](\{x \mapsto T, y \mapsto F\} \mid \tau)$$

$$= 0.4 \cdot 0.9 + 0.6 \cdot 0$$

Semantics of Observations

- What if we introduce observations *only at the end* of the program?

$$[\![s; \mathtt{observe}(\mathtt{e})]\!](\sigma' \mid \sigma)$$

$$\triangleq \begin{cases} \dfrac{[\![\mathtt{s}]\!](\sigma' | \sigma)}{\sum_{\tau \models [\![\mathtt{e}]\!]} [\![\mathtt{s}]\!](\tau | \sigma)} & \sigma' \models [\![\mathtt{e}]\!] \\ 0 & \text{otherwise} \end{cases}$$

- Bayes rule “given that the observe succeeds”
- Look ma! No rejected samples!

What is the meaning of?

$$bar_1 = \left\{ \begin{array}{l} \texttt{if}(x)\ \{\ y \sim \texttt{flip}(1/4)\ \} \\ \texttt{else}\ \{\ y \sim \texttt{flip}(1/2)\ \} \end{array} \right\}$$

$$[\![bar_1]\!]_T(\sigma' \mid \sigma) = \begin{cases} 1/2 & \text{if } x[\sigma] = x[\sigma'] = \mathsf{F}, \\ 1/4 & \text{if } x[\sigma] = x[\sigma'] = \mathsf{T} \text{ and } y[\sigma'] = \mathsf{T}, \\ 3/4 & \text{if } x[\sigma] = x[\sigma'] = \mathsf{T} \text{ and } y[\sigma'] = \mathsf{F}, \\ 0 & \text{otherwise.} \end{cases}$$

What is the meaning of?

$$bar_2 = \left\{\begin{array}{l} y \sim \texttt{flip}(1/2); \\ \texttt{observe}(x \vee y); \\ \texttt{if}(y)\ \{\ y \sim \texttt{flip}(1/2)\ \} \\ \texttt{else}\ \{\ y := \mathsf{F}\ \} \end{array}\right\}$$

$$\begin{array}{l}[\![bar_1]\!]_T(\sigma' \mid \sigma) = \\ [\![bar_2]\!]_T(\sigma' \mid \sigma) = \end{array} \begin{cases} 1/2 & \text{if } x[\sigma] = x[\sigma'] = \mathsf{F}, \\ 1/4 & \text{if } x[\sigma] = x[\sigma'] = \mathsf{T} \text{ and } y[\sigma'] = \mathsf{T}, \\ 3/4 & \text{if } x[\sigma] = x[\sigma'] = \mathsf{T} \text{ and } y[\sigma'] = \mathsf{F}, \\ 0 & \text{otherwise.} \end{cases}$$

Are these programs equivalent?

$$bar_1 = \left\{ \begin{array}{l} \texttt{if}(x)\ \{\ y \sim \texttt{flip}(1/4)\ \} \\ \texttt{else}\ \{\ y \sim \texttt{flip}(1/2)\ \} \end{array} \right\}$$

$$bar_2 = \left\{ \begin{array}{l} y \sim \texttt{flip}(1/2); \\ \texttt{observe}(x \vee y); \\ \texttt{if}(y)\ \{\ y \sim \texttt{flip}(1/2)\ \} \\ \texttt{else}\ \{\ y\ := \texttt{F}\ \} \end{array} \right\}$$

Are these programs equivalent?

$$bar_1 = \left\{ \begin{array}{l} \texttt{if}(x)\ \{\ y \sim \texttt{flip(1/4)}\ \} \\ \texttt{else}\ \{\ y \sim \texttt{flip(1/2)}\ \} \end{array} \right\}$$

$$foo = \{\ \ x \sim \texttt{flip(1/3)}\ \ \}$$

$$bar_2 = \left\{ \begin{array}{l} y \sim \texttt{flip(1/2);} \\ \texttt{observe}(x \vee y)\texttt{;} \\ \texttt{if}(y)\ \{\ y \sim \texttt{flip(1/2)}\ \} \\ \texttt{else}\ \{\ y := \texttt{F}\ \} \end{array} \right\}$$

In $[\![foo; bar_1]\!]$ the probability of x = F in the output state is:

$$2/3$$

In $[\![foo; bar_2]\!]$ the probability of x = F in the output state is:

$$\frac{2/3 \cdot 1/2}{1/3 + 2/3 \cdot 1/2} = \frac{1}{2}$$

Accepting and Transition Semantics

$$[\![\mathtt{skip(e)}]\!]_A(\sigma) \triangleq 1$$
$$[\![x \sim \mathtt{flip}(\theta)]\!]_A(\sigma) \triangleq 1$$
$$[\![x := \mathtt{e}]\!]_A(\sigma) \triangleq 1$$
$$[\![\mathtt{observe(e)}]\!]_A(\sigma) \triangleq \begin{cases} 1 & \text{if } [\![\mathtt{e}]\!](\sigma) = \mathsf{T} \\ 0 & \text{otherwise} \end{cases}$$
$$[\![\mathtt{s}_1; \mathtt{s}_2]\!]_A(\sigma) \triangleq [\![\mathtt{s}_1]\!]_A(\sigma) \times \sum_{\tau \in \Sigma} \left([\![\mathtt{s}_1]\!]_T(\tau \mid \sigma) \times [\![\mathtt{s}_2]\!]_A(\tau)\right)$$
$$[\![\mathtt{if\ e\ \{s_1\}\ else\ \{s_2\}}]\!]_A(\sigma) \triangleq \begin{cases} [\![\mathtt{s}_1]\!]_A(\sigma) & \text{if } [\![\mathtt{e}]\!](\sigma) = \mathsf{T} \\ [\![\mathtt{s}_2]\!]_A(\sigma) & \text{otherwise} \end{cases}$$

$$[\![\mathtt{skip}]\!]_T(\sigma' \mid \sigma) \triangleq \begin{cases} 1 & \text{if } \sigma' = \sigma \\ 0 & \text{otherwise} \end{cases}$$
$$[\![x \sim \mathtt{flip}(\theta)]\!]_T(\sigma' \mid \sigma) \triangleq \begin{cases} \theta & \text{if } \sigma' = \sigma[x \mapsto \mathsf{T}] \\ 1 - \theta & \text{if } \sigma' = \sigma[x \mapsto \mathsf{F}] \\ 0 & \text{otherwise} \end{cases}$$
$$[\![x := \mathtt{e}]\!]_T(\sigma' \mid \sigma) \triangleq \begin{cases} 1 & \text{if } \sigma' = \sigma[x \mapsto [\![\mathtt{e}]\!](\sigma)] \\ 0 & \text{otherwise} \end{cases}$$
$$[\![\mathtt{observe(e)}]\!]_T(\sigma' \mid \sigma) \triangleq \begin{cases} 1 & \text{if } \sigma' = \sigma \text{ and } [\![\mathtt{e}]\!](\sigma) = \mathsf{T} \\ 0 & \text{otherwise} \end{cases}$$
$$[\![\mathtt{s}_1; \mathtt{s}_2]\!]_T(\sigma' \mid \sigma) \triangleq \frac{\sum_{\tau \in \Sigma} [\![\mathtt{s}_1]\!]_T(\tau \mid \sigma) \times [\![\mathtt{s}_2]\!]_T(\sigma' \mid \tau) \times [\![\mathtt{s}_2]\!]_A(\tau)}{\sum_{\tau \in \Sigma} [\![\mathtt{s}_1]\!]_T(\tau \mid \sigma) \times [\![\mathtt{s}_2]\!]_A(\tau)}$$
$$[\![\mathtt{if\ e\ \{s_1\}\ else\ \{s_2\}}]\!]_T(\sigma' \mid \sigma) \triangleq \begin{cases} [\![\mathtt{s}_1]\!]_T(\sigma' \mid \sigma) & \text{if } [\![\mathtt{e}]\!](\sigma) = \mathsf{T} \\ [\![\mathtt{s}_2]\!]_T(\sigma' \mid \sigma) & \text{if } [\![\mathtt{e}]\!](\sigma) = \mathsf{F} \end{cases}$$

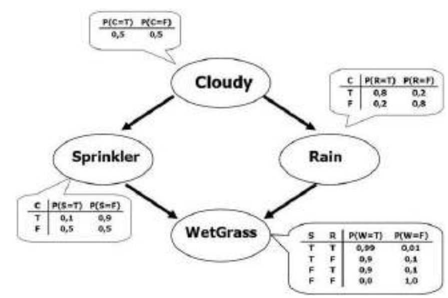
Pitfalls of Denotational Semantics

- Intermediate observes:
  - Need accepting semantic
  - Key difference from probabilistic graphical models
  - Sometimes encoded using unnormalized probabilities
- While loops
  - Bounded? *"while(i<10)"*
  - Almost surely terminating? *"while(flip(0.5))"*
  - Not almost surely terminating? "*while(true)*"
- Adding continuous variables:
  - Indian GPA problem [Wu et al. ICML 2018]
  - What is the meaning of "if(Normal(0,1) == 0.34) then …"
  - Etc.

How to do exact inference for probabilistic programs?

The Challenge of PPL Inference

- Probabilistic inference is #P-*hard*
  - – Implies there is likely no universal solution
- In practice inference is often feasible
  - – Often relies on conditional independence
  - – Manifests as *graph properties*



- *Why exact?*
  1. No error propagation
  2. Approximations are intractable in theory as well
  3. Approximates are known to mislead learners
  4. Core of effective approximation techniques
  5. Unaffected by low-probability observations

Techniques for exact inference

Exploits independence to decompose inference?	Yes	**Graphical Model Compilation**	**Symbolic compilation (This work)**
	No		**Enumeration**
		No	Yes

Keeps program structure?

PL Background: Symbolic Execution

- Non-probabilistic programs can be interpreted as *logical formulae* which relate input and output states


```
x := y;
```

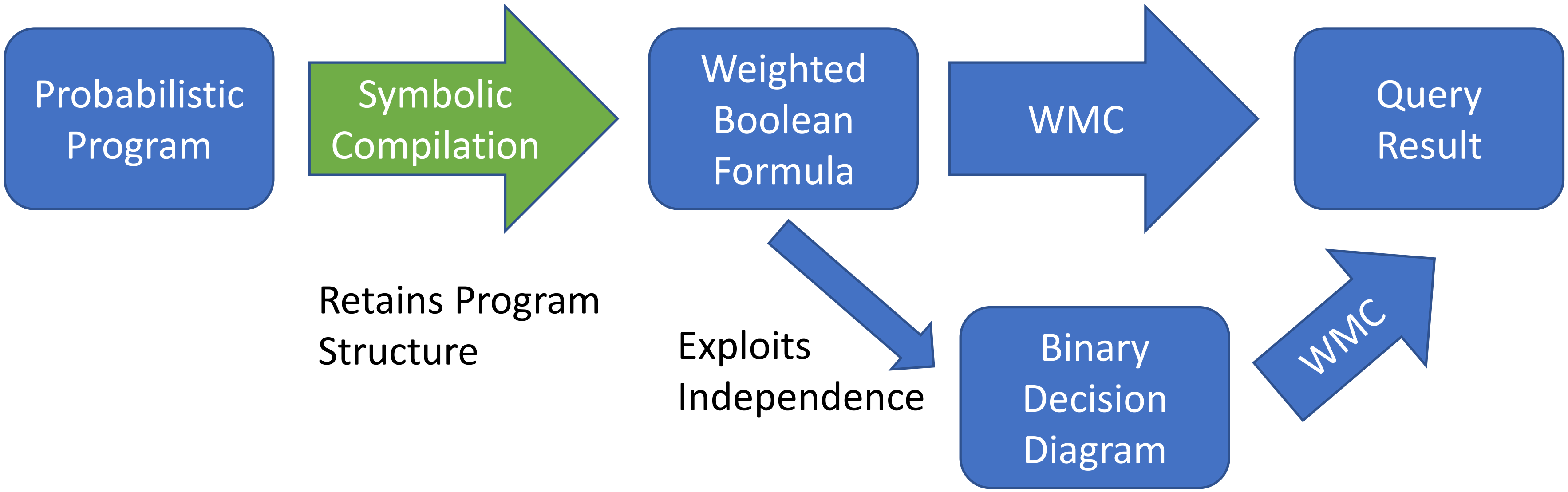
$$\varphi = (x' \Leftrightarrow y) \land (y' \Leftrightarrow y)$$

$$SAT(\varphi \land x' \land y) = T$$

$$SAT(\varphi \land x' \land \bar{y}) = \mathrm{F}$$

Output state: primed
Input state: unprimed

Our Approach: Inference via Weighted Model Counting



Inference via Weighted Model Counting

Probabilistic Program → Symbolic Compilation → Weighted Boolean Formula → WMC → Query Result

```
x := flip(0.4);
```

$\boldsymbol{l}$	$\boldsymbol{w(l)}$
f_1	0.4
$\overline{f_1}$	0.6

$(x' \Leftrightarrow f_1)$

$$\mathrm{WMC}(\varphi, w) = \sum_{m \models \varphi} \prod_{l \in m} w(l).$$

$\mathrm{WMC}\big((x' \Leftrightarrow f_1) \wedge x \wedge x', w\big)$?

- A single model: $\mathrm{m} = x' \wedge x \wedge f_1$
- $w(x') * w(x) * w(f_1) = 0.4$

Symbolic compilation: Flip

- Compositional process $\mathtt{s} \rightsquigarrow (\varphi, w)$

$$\frac{\text{fresh } f}{x \sim \mathtt{flip}(\theta) \rightsquigarrow \Big((x' \Leftrightarrow f) \wedge (\text{rest unchanged}), w\Big)}$$

All variables in the program except
for x are not changed by this statement

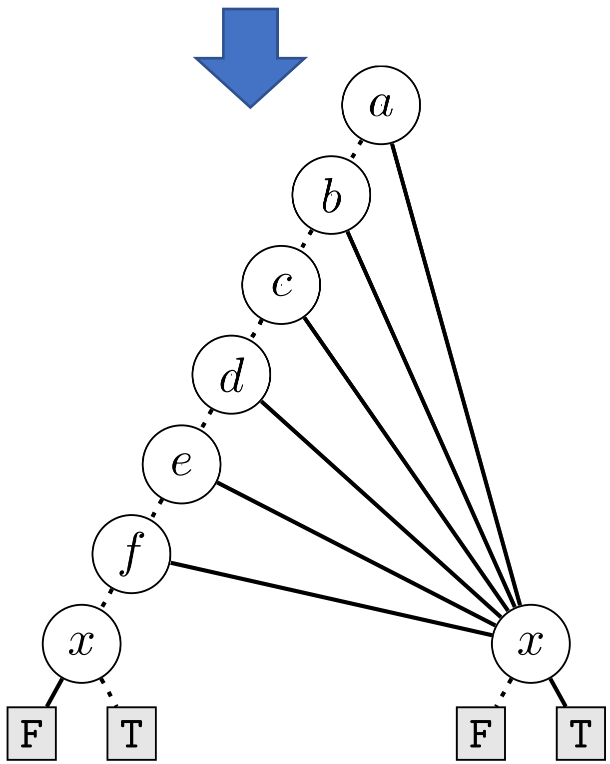
Symbolic compilation: Assignment

- Compositional process $\mathtt{s} \rightsquigarrow (\varphi, w)$

$$\frac{}{x := \mathtt{e} \rightsquigarrow \Big((x' \Leftrightarrow \mathtt{e}) \wedge (\text{rest unchanged}), w\Big)}$$

Compiling to BDDs

- BDDs compactly capture complex program structure `x = a || b || c || d || e || f;`



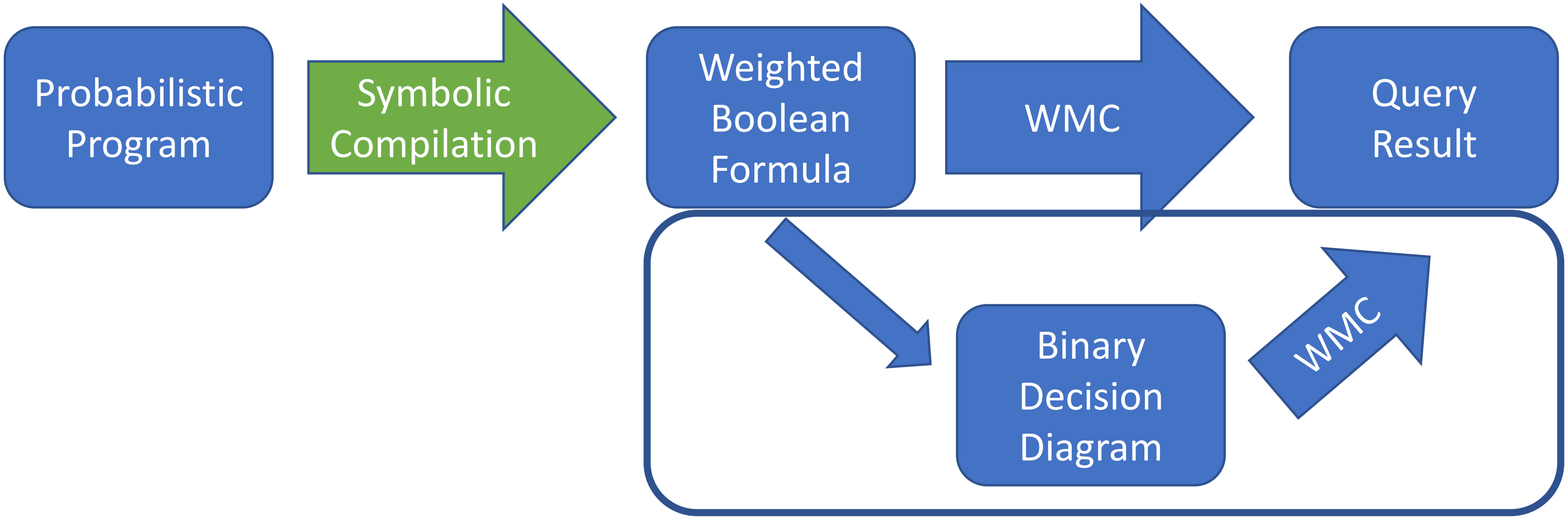
Symbolic compilation: Sequencing

- Compositional process $\mathbf{s} \rightsquigarrow (\varphi, w)$

$$\frac{s_1 \rightsquigarrow (\varphi_1, w_1) \qquad s_2 \rightsquigarrow (\varphi_2, w_2) \qquad \varphi_2' = \varphi_2[x_i \mapsto x_i', x_i' \mapsto x_i'']}{\mathbf{s}_1; \mathbf{s}_2 \rightsquigarrow ((\exists x_i'.\varphi_1 \wedge \varphi_2')[x_i'' \mapsto x_i'], w_1 \uplus w_2)}$$

- Compile two sub-statements, do some relabeling, then combine them to get the result

Inference via Weighted Model Counting



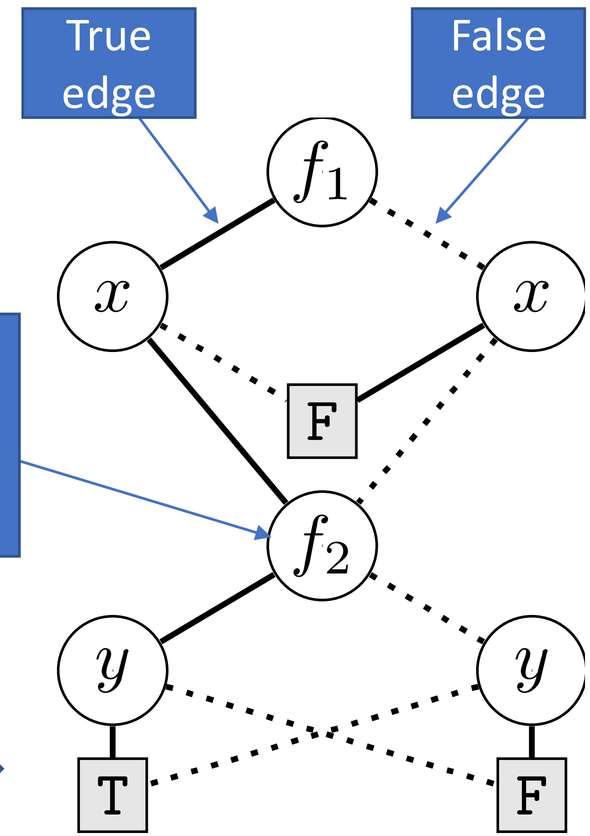
Compiling to BDDs

- Consider an example program:

```
x~flip(0.4);
y~flip(0.6)
```

$$(\mathtt{x} \iff f_1) \wedge (y \iff f_2)$$

This sub-function does not depend on x: exploits independence

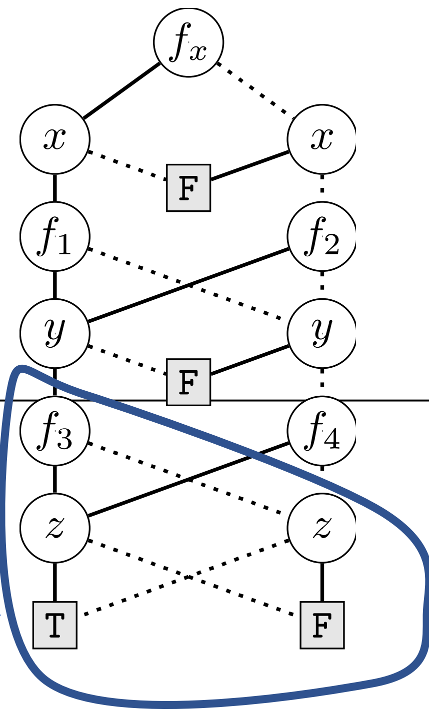


- WMC is efficient for BDDs: *time linear* in size
  - Small BDD = Fast Inference

BDDs Exploit Conditional Independence

Size of BDD grows linearly with length of Markov chain

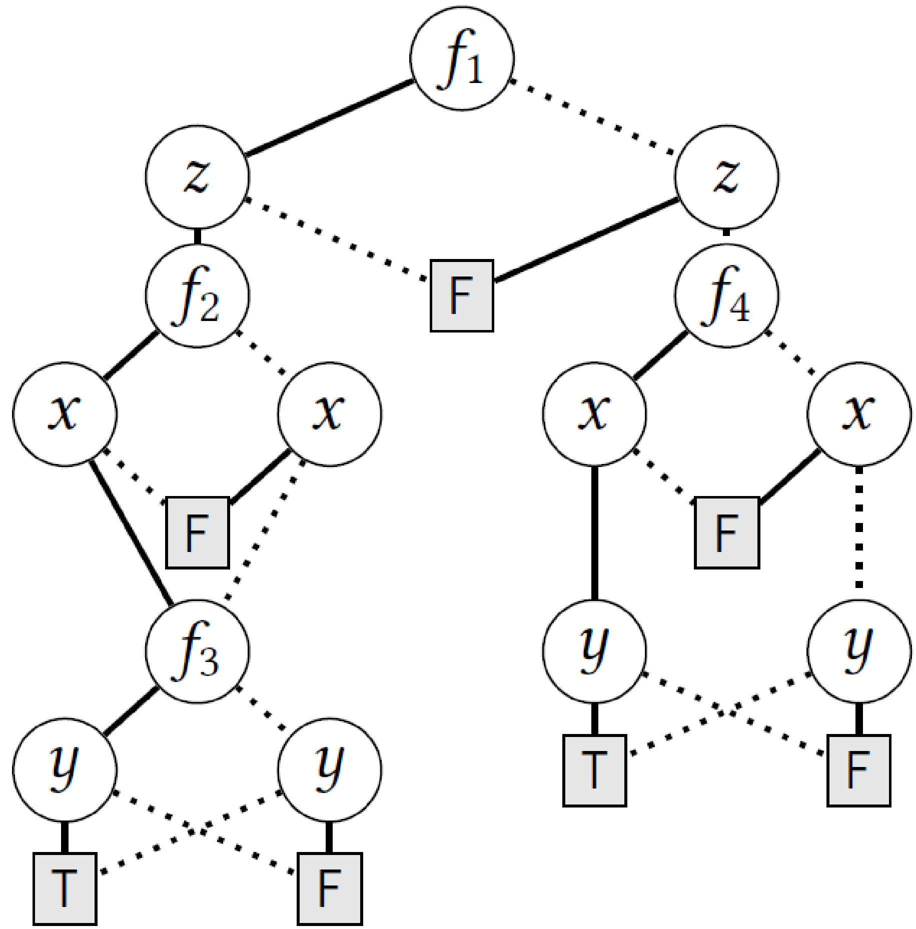
```
x ~flip_x(0.5);
if(x) { y ~flip_1(0.6) }
else { y ~flip_2(0.4) };
if(y) { z ~flip_3(0.6) }
else { z ~flip_4(0.9) }
```



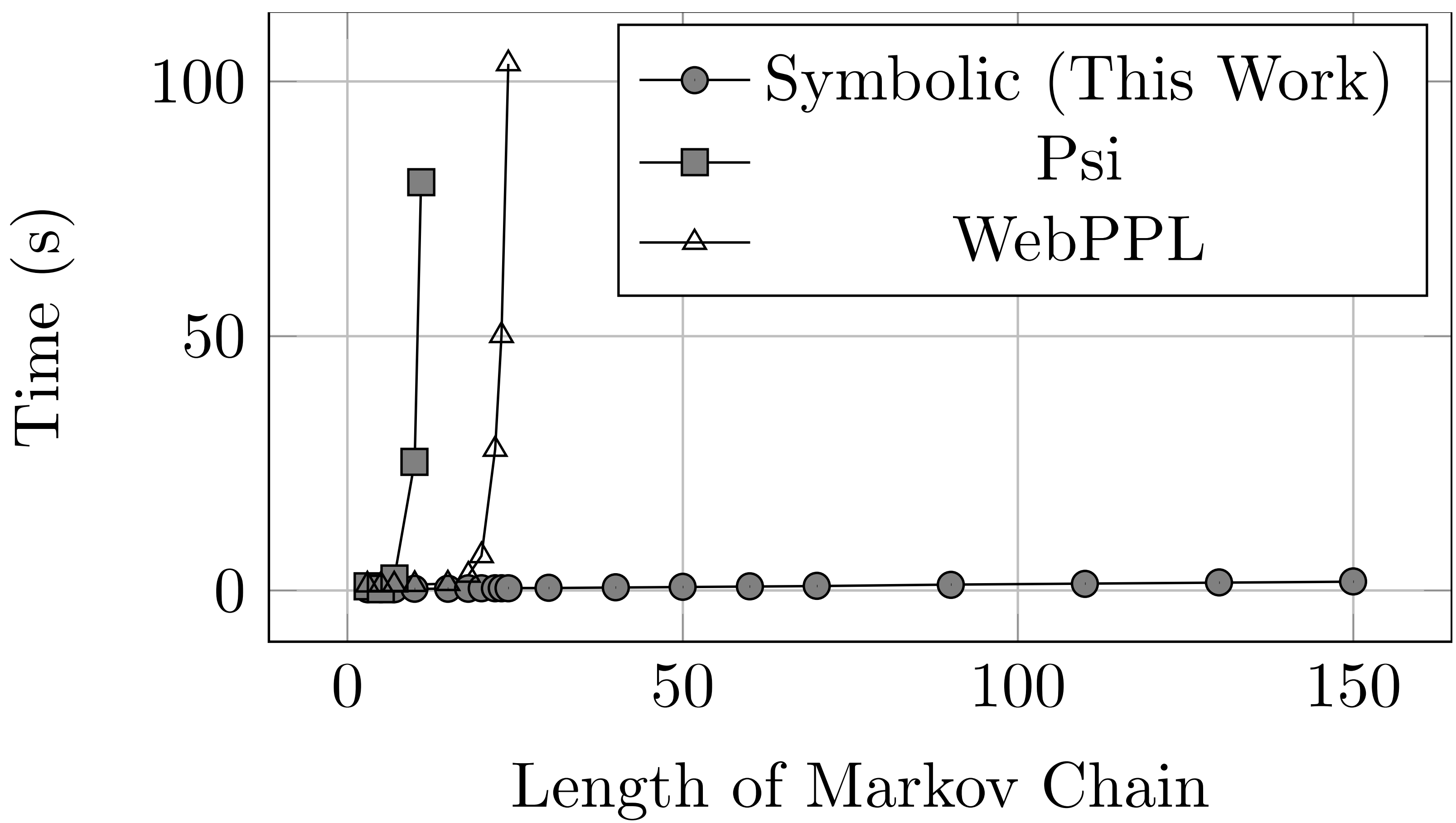
Given y=T, does not depend on the value of X: exploits conditional independence

BDDs Exploit Context-Specific Independence

```
z ~flip₁(0.5);
if(z) {
  x ~flip₂(0.6);
  y ~flip₃(0.7)
} else {
  x ~flip₄(0.4);
  y := x
}
```



Experiments: Markov Chain

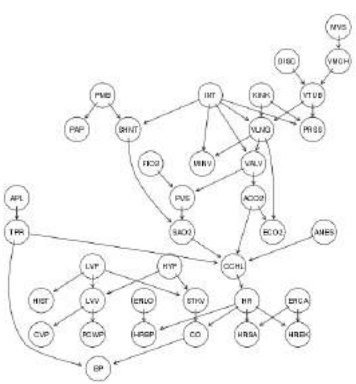


Experiment: Bayesian Networks

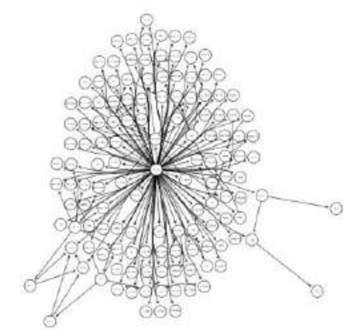
Large programs (thousands of lines, tens of thousands of flips)

Model	Us (s)	BN Time (s) [6]	Size of BDD
Alarm [6]	1.872	0.21	52k
Halfinder	12.652	1.37	157k
Hepar2	7.834	0.28 [11]	139k
pathfinder	62.034	14.94	392k

Specialized BN
inference algorithm



Alarm Network

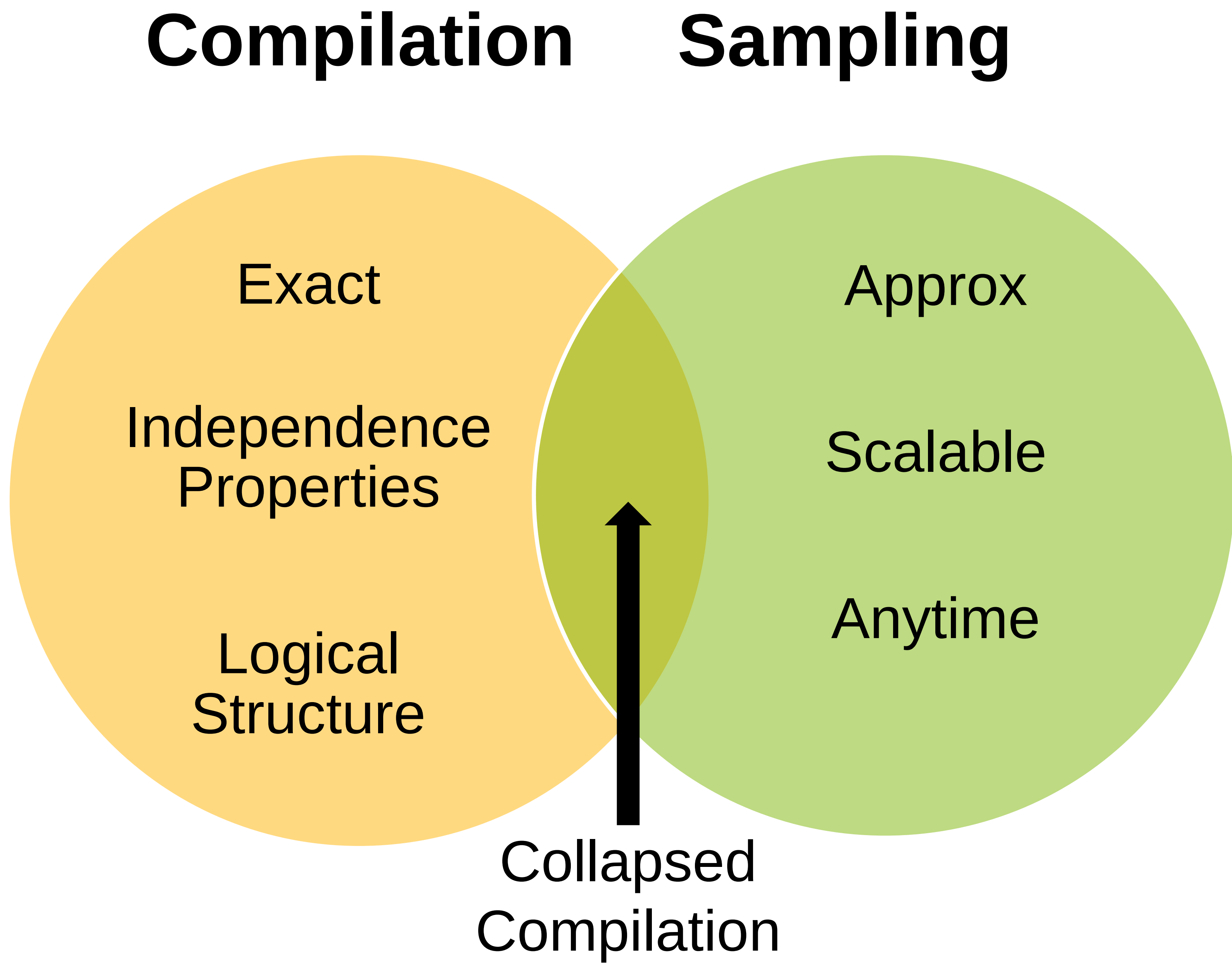


Pathfinder Network

Symbolic Compilation

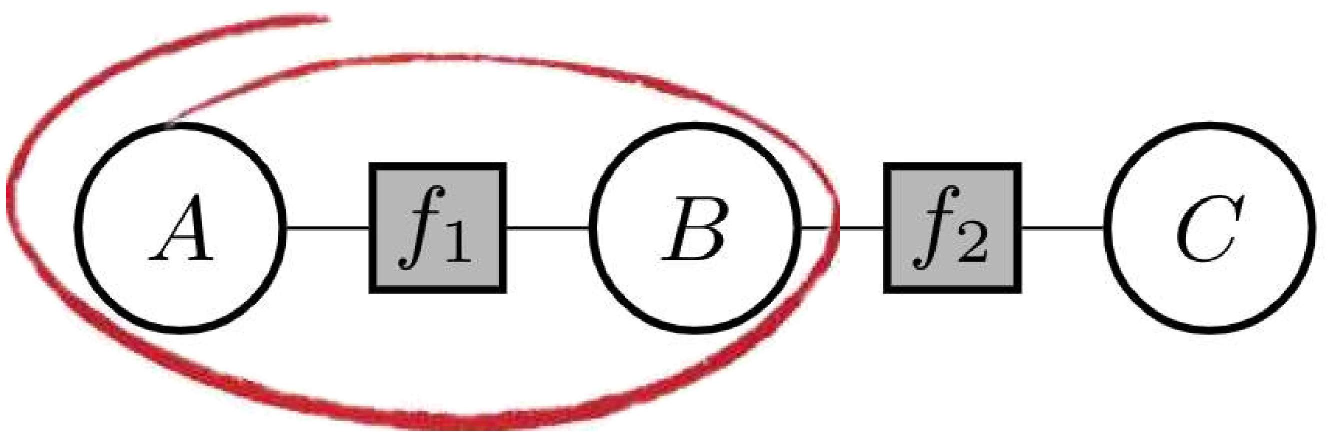
- Exact inference algorithm for discrete programs
  - Relies on PL ideas to construct state space: symbolic execution, symbolic model checking
  - Relies on AI ideas to perform inference: weighted model counting, knowledge compilation
- Proved **correct** (= denotational semantics)
- Competitive performance
- Will release a language+system soon!
- Also see probabilistic **logic** programming work
  - Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert and Luc De Raedt. Tp-Compilation for Inference in Probabilistic Logic Programs, *In International Journal of Approximate Reasoning*, 2016.
  - Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens and Luc De Raedt. Inference and Learning in Probabilistic Logic Programs using Weighted Boolean Formulas, *In Theory and Practice of Logic Programming*, volume 15, 2015.

What about approximate inference?


Compilation
Sampling
Exact
Independence Properties
Logical Structure
Approx
Scalable
Anytime
Collapsed Compilation

Collapsed Sampling (Rao-Blackwell)

Sampling on some variables,
exact inference conditioned on sample

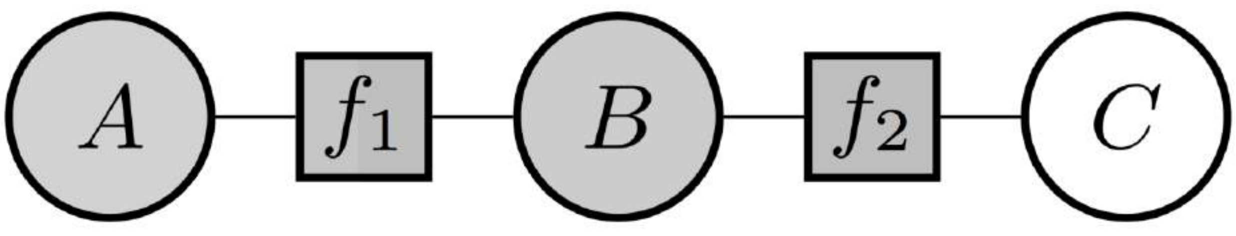


Sample A,B

Collapsed Sampling (Rao-Blackwell)

Sampling on some variables,

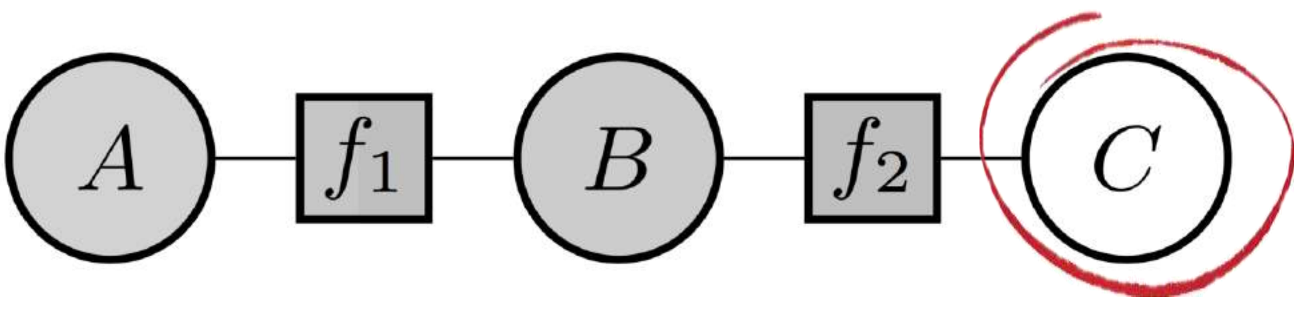
exact inference conditioned on sample



Observe sampled values

Collapsed Sampling (Rao-Blackwell)

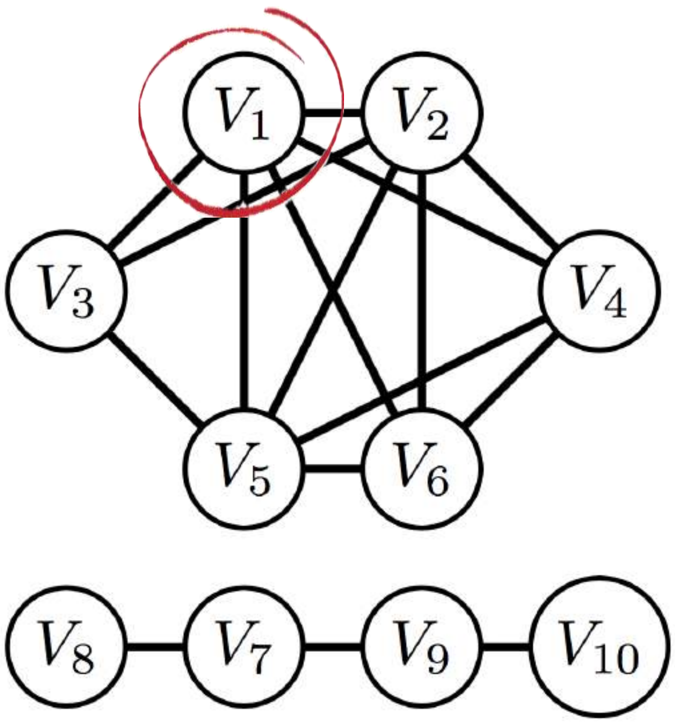
Sampling on some variables,
exact inference conditioned on sample



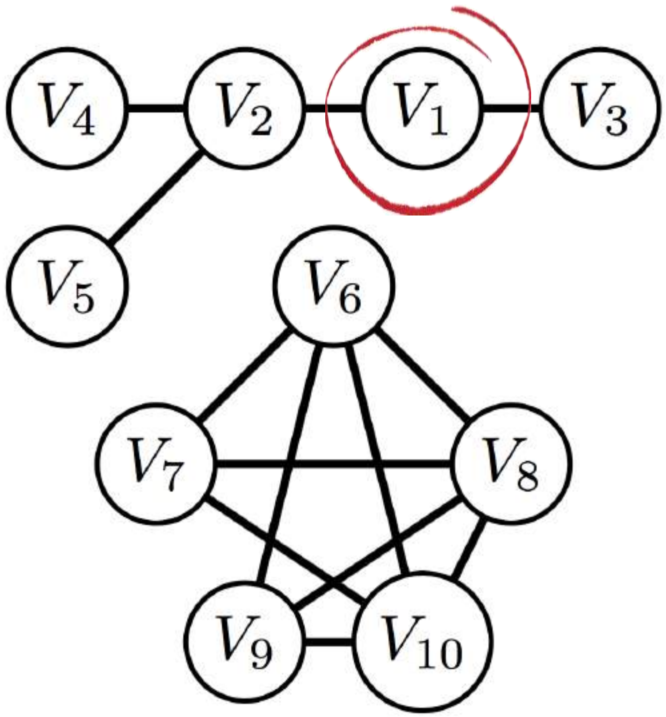
Compute exactly P(C|A,B)

What to Sample?

- Is it even possible to pick a correct set a priori?
- Consider a network of potential smokers, with friendships sampled



Sample 1



Sample 2

Online Collapsed Sampling

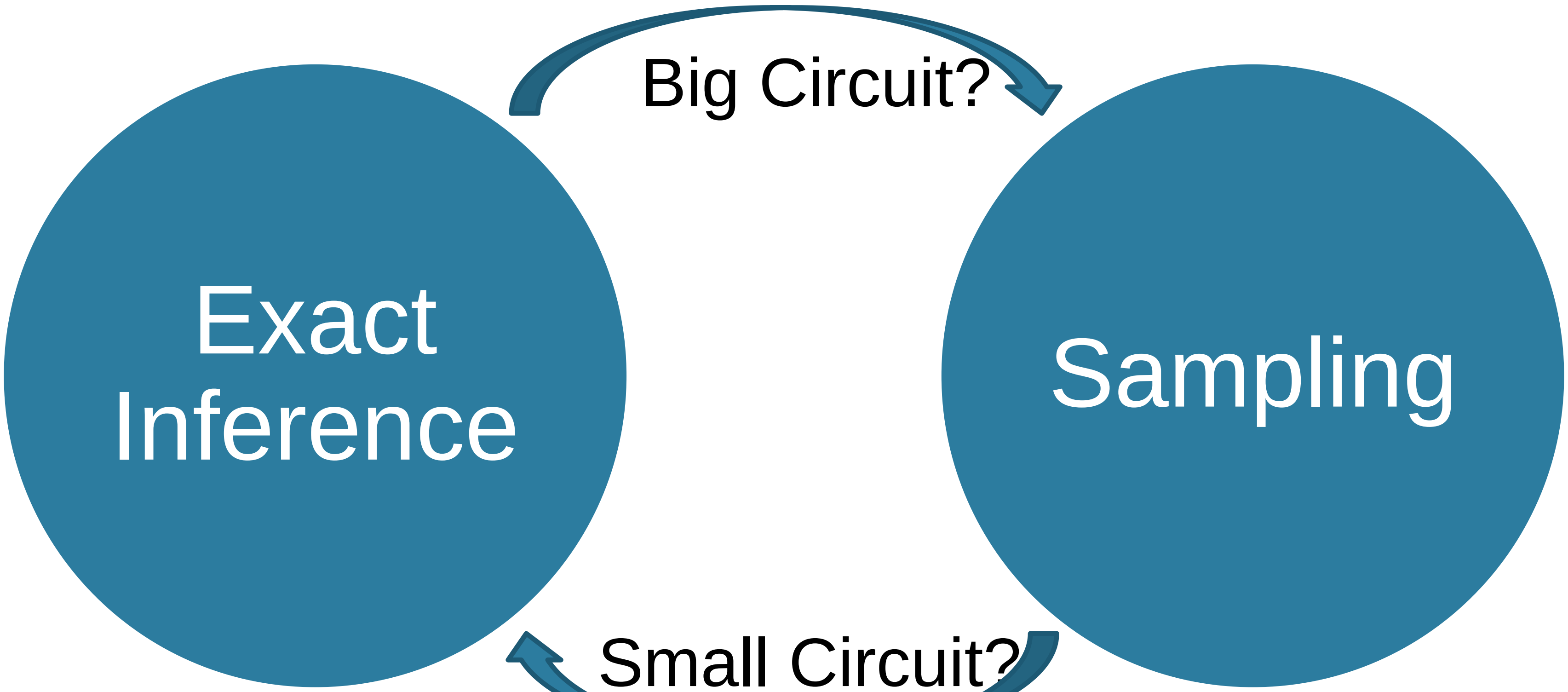
Choose *on-the-fly* which variable to sample next, based on result of sampling previous variables

Theorem: Still unbiased

How to do Collapsed Sampling?

1. What/when do we sample?
2. How do we sample?
3. How do we do exact inference?

Collapsed Compilation



Result: A circuit with some sampled variables

How to do Collapsed Compilation?

1. What/when do we sample?
   - *When*: Circuit too big
   - *What*: Heuristic on current circuit
     Intuition: variables with dense weak dependencies
2. How do we sample?
3. How do we do exact inference?

How to do Collapsed Compilation?

1. What/when do we sample?
2. How do we sample?
   - Importance Sampling
   - Need a proposal for **any** variable conditioned on **any other** variables
   - Sample according to marginal in current partially compiled circuit
3. How do we do exact inference?

How to do Collapsed Compilation?

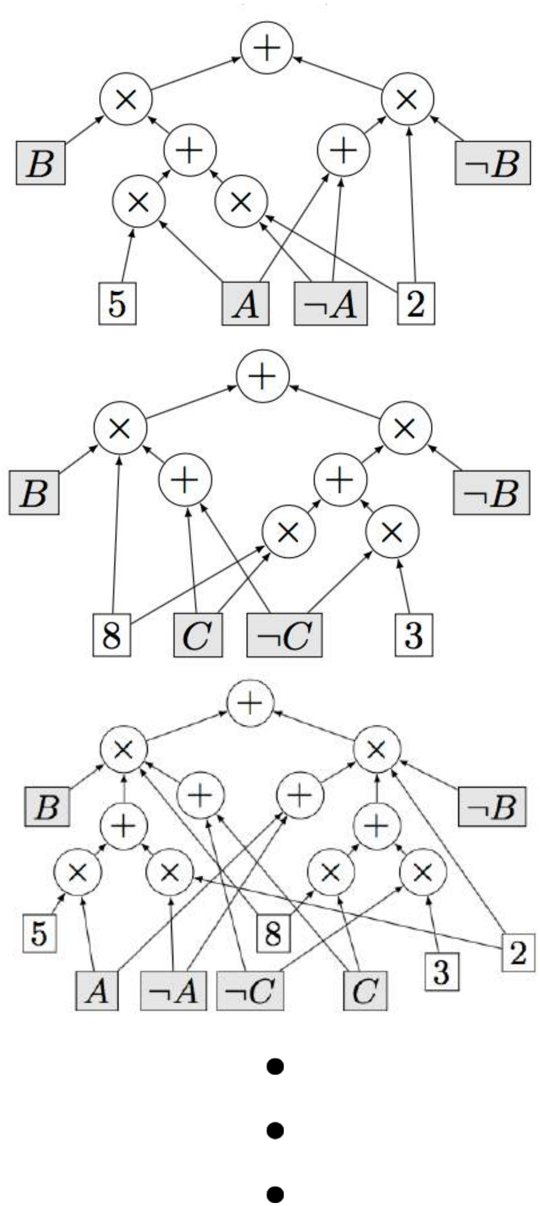
1. What/when do we sample?
2. How do we sample?
3. How do we do exact inference?
   – Compiled circuit for each sample
   – Tractable for all required computations (marginals, particle weights, etc.)

Collapsed Compilation Algorithm

To sample a circuit:

1. Compile bottom up until you reach the size limit
2. Pick a variable you want to sample
3. Sample it according to its marginal distribution in the current circuit
4. Condition on the sampled value
5. (Repeat)

Asymptotically unbiased importance sampler ☺



Circuits +
importance weights
approximate any query

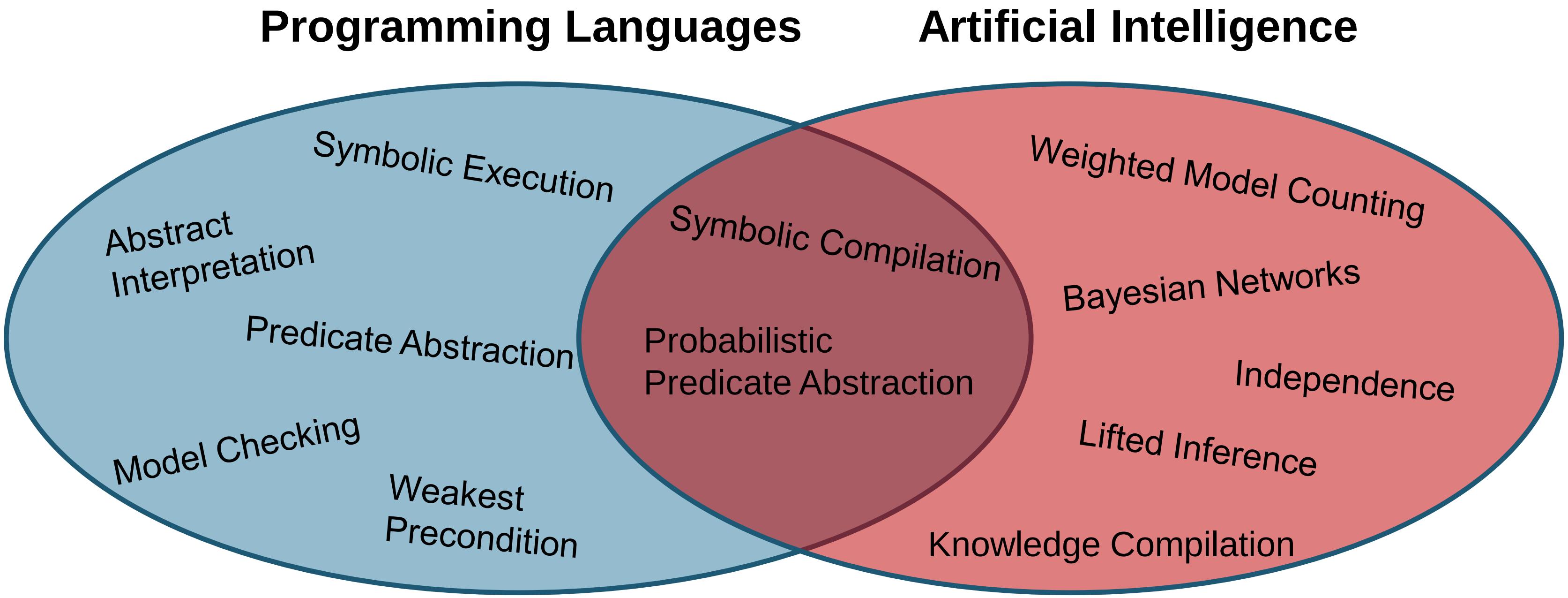
Experiments

Table 2: Hellinger distances across methods with internal treewidth and size bounds

Method	50-20	75-26	DBN	Grids	Segment	linkage	frust
EDBP-100k	2.19e−3	3.17e−5	6.39e−1	1.24e−3	1.63e−6	6.54e−8	4.73e−3
EDBP-1m	7.40e−7	2.21e−4	6.39e−1	1.98e−7	1.93e−7	5.98e−8	4.73e−3
SS-10	2.51e−2	2.22e−3	6.37e−1	3.10e−1	3.11e−7	4.93e−2	1.05e−2
SS-12	6.96e−3	1.02e−3	6.27e−1	2.48e−1	3.11e−7	1.10e−3	5.27e−4
SS-15	9.09e−6	1.09e−4	(Exact)	8.74e−4	3.11e−7	4.06e−6	6.23e−3
FD	9.77e−6	1.87e−3	1.24e−1	1.98e−4	6.00e−8	5.99e−6	5.96e−6
MinEnt	1.50e−5	3.29e−2	1.83e−2	3.61e−3	3.40e−7	6.16e−5	3.10e−2
RBVar	2.66e−2	4.39e−1	6.27e−3	1.20e−1	3.01e−7	2.02e−2	2.30e−3

Competitive with state-of-the-art approximate inference in graphical models.
Outperforms it on several benchmarks!

Conclusions



Thanks

- Steven Holtzen, Todd Millstein and Guy Van den Broeck. Symbolic Exact Inference for Discrete Probabilistic Programs, *In Proceedings of the ICML Workshop on Tractable Probabilistic Modeling (TPM)*, 2019.
- Tal Friedman and Guy Van den Broeck. Approximate Knowledge Compilation by Online Collapsed Importance Sampling, *In Advances in Neural Information Processing Systems 31 (NeurIPS)*, 2018.
- Steven Holtzen, Guy Van den Broeck and Todd Millstein. Sound Abstraction and Decomposition of Probabilistic Programs, *In Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.
- Steven Holtzen, Todd Millstein and Guy Van den Broeck. Probabilistic Program Abstractions, *In Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI)*, 2017.

…with slides stolen from Steven Holtzen and Tal Friedman.