Scaling Decision—Theoretic Probabilistic Programming Through Factorization

Northeastern University Khoury College of Computer Sciences

Minsung Cho, Steven Holtzen

Programming Research Lab, Northeastern University

Probabilistic programs are designed to specify probabilistic models.

Probabilistic models can do more than inference, like decision making. Why can't probabilistic programs?

Goal: Let's design a probabilistic language that can reason about rational decision making under uncertainty!

What are our language specifications? Our semantics?

We construct a decision choosing between n choices by declaring `[a1,...,an]`, all fresh variables

We destruct a decision by declaring `choose d | a1 => e1 |...| an => en`

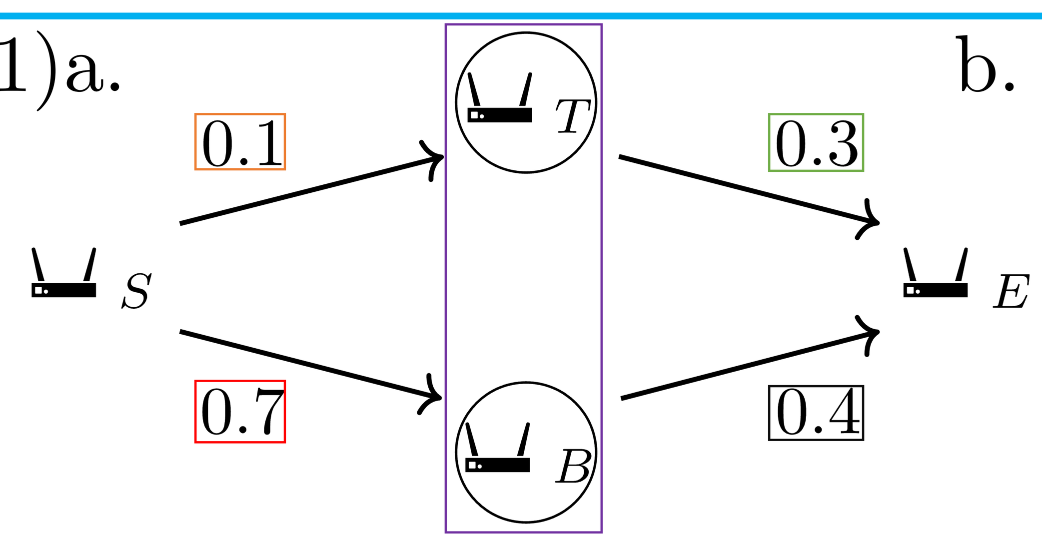
We add in reward primitives `reward k`, where `k` is a real number, to denote reward/cost

Two mutually recursive denotational semantics $[\![\cdot]\!]_{\mathrm{Pr}}, [\![\cdot]\!]_{\mathbb{E}U}$:

- $[\![\cdot]\!]_{\mathrm{Pr}}$ specifies the underlying probability distribution given the choices made
- $[\![\cdot]\!]_{\mathbb{E}U}$ specifies the accumulated reward/cost given the choices made, scaled by probability

1) A sample decision making scenario where investigate whether router T or B is faulty after a network failure. If we find the faulty router, we get a reward of 10. a. A graphical depiction, b. corresponding decision—theoretic probabilistic program, c. the compiled Boolean formula.
2) Selected denotational semantics and their respective types.
3) Selected compilation rules. Programs compile into a quadruple of the Boolean formula, an "accepting" function witnessing observations, a weight function on literals, and a set of accumulated reward variables.
4) A graph demonstrating promising initial results of our compilation and inference algorithm over ProbLog in a generalization of the example in 1). n denotes the number of vertices along the top or bottom route. D-constrained is best in this example but fails to support fast nested decision making.

1) a.



b.

```
let st, te = flip 0.1, flip 0.3 in
let sb, be = flip 0.7, flip 0.4 in
let toproute = if st then te else false in
let botroute = if sb then be else false in
let _ = observe(!toproute and !botroute) in
choose [t,b]
| t => if st and !te then reward 10 else reward 0
| b => if sb and !be then reward 10 else reward 0
```

c.

$$\begin{aligned}\varphi = &(\overline{toproute} \wedge \overline{botroute})\\ &\wedge [(t \wedge ((st \wedge \overline{te} \wedge r_{10}) \vee (\overline{st \wedge \overline{te}} \wedge r_0)))\\ &\vee (b \wedge ((sb \wedge \overline{be} \wedge r_{10}) \vee (\overline{sb \wedge \overline{be}} \wedge r_0)))]\\ &\wedge \mathrm{ExactlyOne}(t,b) \wedge \mathrm{ExactlyOne}(r_{10}, r_0)\end{aligned}$$

2)

$[\![\cdot]\!]_{\mathrm{Pr}} : \texttt{Programs} \to \texttt{Choices} \to (\{T,F\} \to [0,1])$

$[\![\texttt{reward}\ k]\!]_{\mathrm{Pr}}(\pi)(v) = \delta_T(v)$

$$[\![[a_1,\cdots,a_n]]\!]_{\mathrm{Pr}}(\pi)(v) = \begin{cases}\delta_T(v) & \exists i.\ a_i \in \pi\\ 0 & else\end{cases}$$

$[\![\texttt{choose}\ [a_1,\cdots,a_n]\ \{a_i \implies e_i\}]\!]_{\mathrm{Pr}}(\pi)(v) = [\![e_i]\!]_{\mathrm{Pr}}(\pi)(v)$ for i s.t. $a_i \in \pi$

$[\![\cdot]\!]_{\mathbb{E}U} : \texttt{Programs} \to \texttt{Choices} \to \mathbb{R}$

$[\![\texttt{reward}\ k]\!]_{\mathbb{E}U}(\pi) = k$

$[\![[a_1,\cdots,a_n]]\!]_{\mathbb{E}U}(\pi) = 0$

$[\![\texttt{choose}\ [a_1,\cdots,a_n]\ \{a_i \implies e_i\}]\!]_{\mathbb{E}U}(\pi) = [\![e_i]\!]_{\mathbb{E}U}(\pi)$ for i s.t. $a_i \in \pi$

$$[\![\texttt{let}\ x := e\ \texttt{in}\ e']\!]_{\mathbb{E}U} = \sum_{v' \in V} [\![e]\!]_{\mathrm{Pr}}(v')([\![e]\!]_{\mathbb{E}U}[\![e'[x \mapsto v']]\!]_{\mathrm{Pr}}(T) + [\![e'[x \mapsto v']]\!]_{\mathbb{E}U})$$

How do we make our language fast?

We compile to a Boolean formula where literals are weighted by the expectation semiring $\mathbb{R}^{\geq 0} \times \mathbb{R}$

This Boolean formula is then represented in a **factorized** manner in binary decision diagrams

We developed a **new, highly general branch-and-bound style algorithm** that computes the correct maximum expected utility/reward

Proven correct, implemented in Rust, outperforms ProbLog's default solver by >=10x on average

We prove an adequacy result: **maximizing the accumulated reward denotational semantics equals the maximum expected utility on the compiled Boolean formula. Moreover, the corresponding optimal decisions taken will also match.**

Proof is an induction on the Boolean compilation rules, such as the ones below!

3)

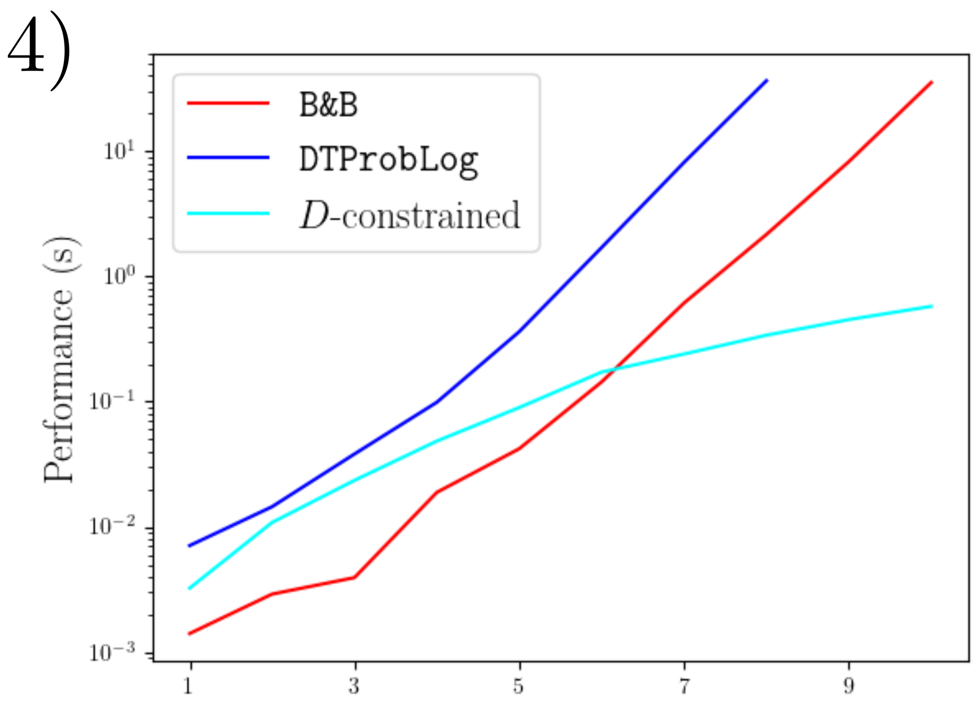
$$\frac{k \in \mathbb{R} \quad r_k \text{ fresh}}{\texttt{reward}\ k \rightsquigarrow (r_k, \top, (r_k \mapsto (1,k), \overline{r} \mapsto (1,0)), \{r_k\})}\ \texttt{bc/reward}$$

$$\frac{a_1, \cdots a_n \text{ fresh}}{[a_1,\cdots,a_n] \rightsquigarrow (\mathrm{ExactlyOne}(a_1,\cdots,a_n), \top, (a_i \mapsto (1,0), \overline{a_i} \mapsto (1,0)), \varnothing)}\ \texttt{bc/[]}$$

$$\frac{e \rightsquigarrow (\varphi, \gamma_1, w_1, R_1) \quad e' \rightsquigarrow (\psi, \gamma_2, w_2, R_2)}{\texttt{let}\ x := e\ \texttt{in}\ e' \rightsquigarrow (\psi[x \mapsto \varphi], \gamma_1 \wedge \gamma_2[x \mapsto \varphi], w_1 \cup w_2, R_1 \cup R_2)}\ \texttt{bc/let}$$

$$\frac{d \rightsquigarrow (\varphi, \top, w_d, \varnothing) \quad \forall i.\ e_i \rightsquigarrow (\psi_i, \gamma_i, w_i, R_i) \quad \forall i.\ a_i \in d}{\texttt{choose}\ d\ \{a_i \implies e_i\} \rightsquigarrow (\varphi \wedge \mathrm{ExactlyOne}(R_i) \wedge \bigvee_i (a_i \wedge \psi_i), \bigwedge_i(\gamma_i), w_d \cup \bigcup_i w_i, \bigcup_i R_i)}\ \texttt{bc/choose}$$

4)



What is future work?

Implementing the language with additional features to run more sophisticated benchmarks

Generalizing our branch-and-bound to work on a broader class of optimization problems in PPLs

Learn more at my DRAGSTERS talk:

If you have any questions or comments, contact me at minsung@ccs.neu.edu!