Probabilistic Programming for Planning as Inference Problems

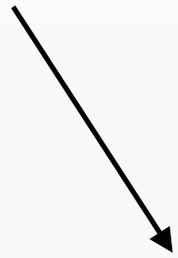
Jan-Willem van de Meent, Northeastern University

[van de Meent, Paige, Tolpin, Wood, AISTATS 2016]

[Siddharth, Paige, van de Meent, Desmaison, Goodman, Kohli, Wood, Torr, NIPS 2017]

Probabilistic Programming

Modeling Language (Domain Structure) + Inference Algorithms (Expected Values)

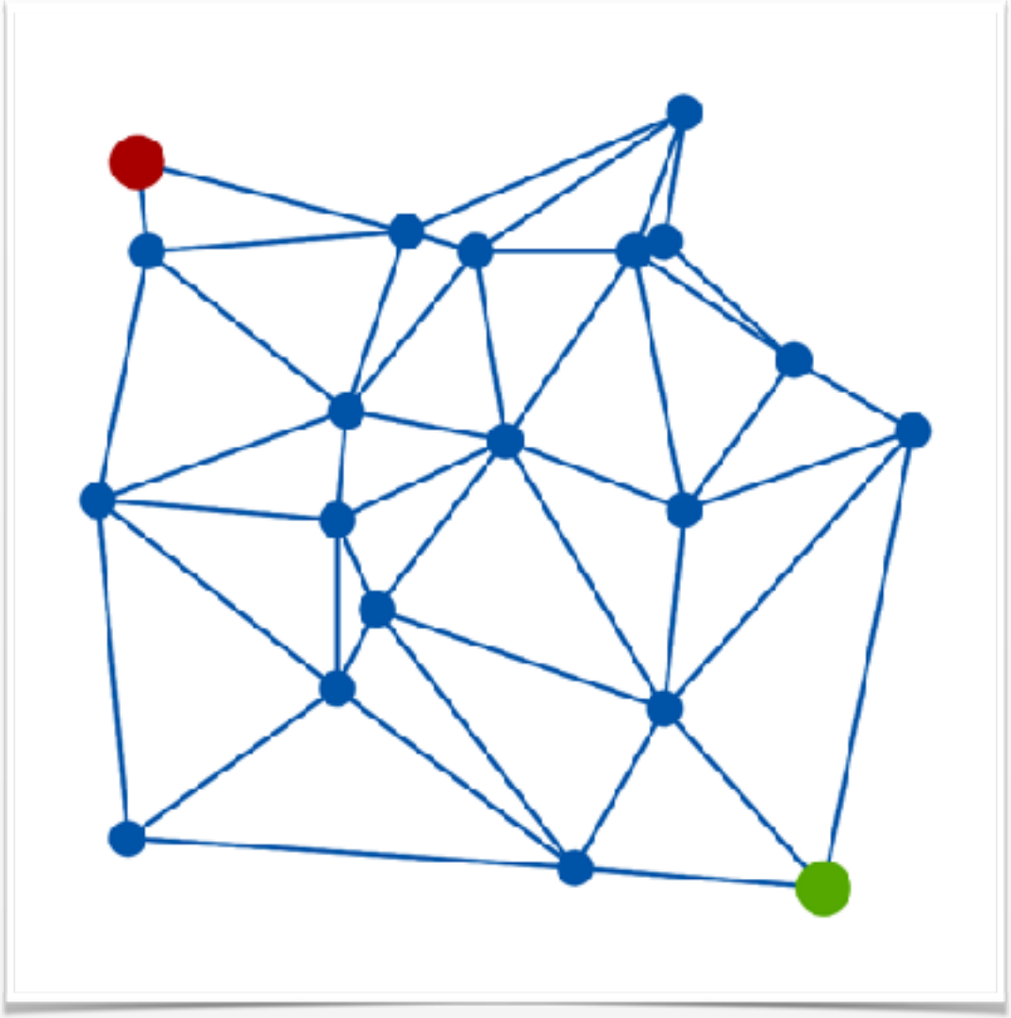


Gradient Estimators (Monte Carlo)

Planning as Inference

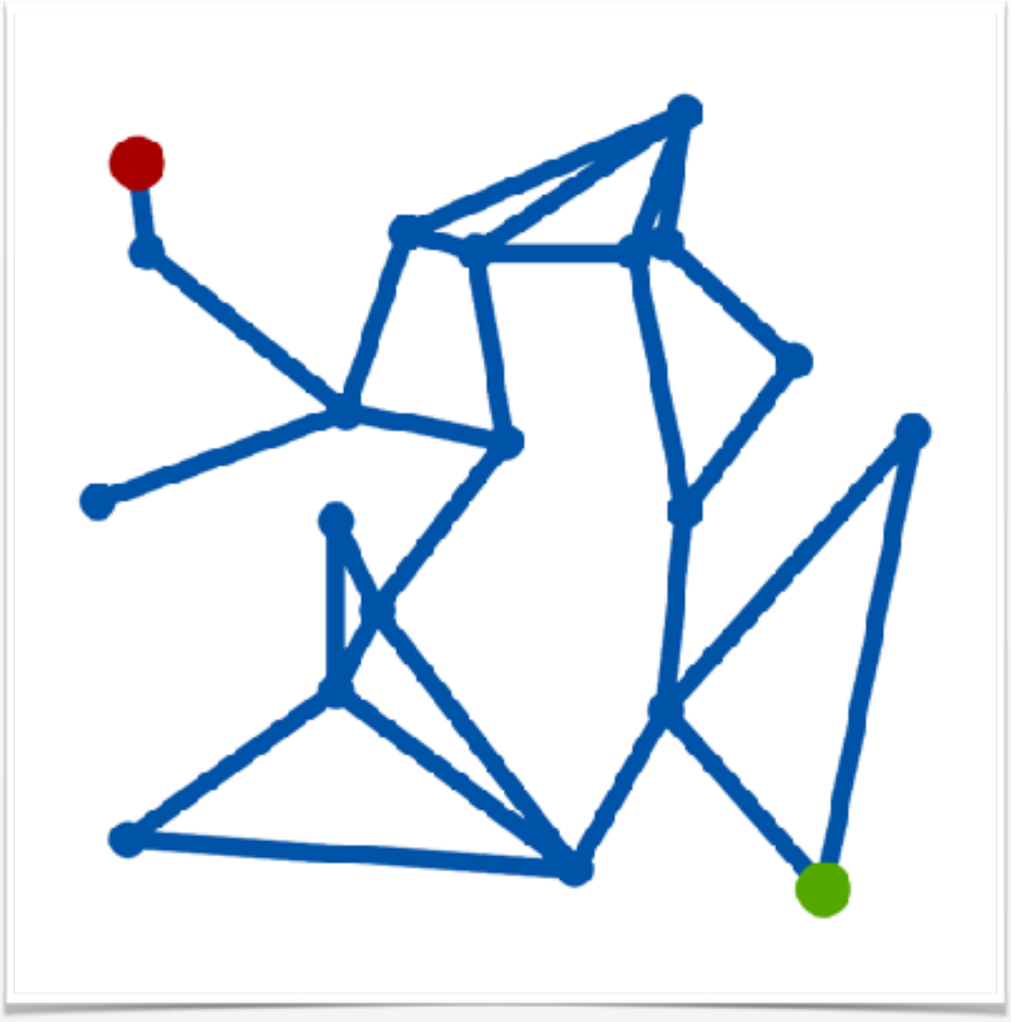
Policy Search (Algorithmic Policies) = Maximum Likelihood (Gradient Descent)

Canadian Traveler Problem



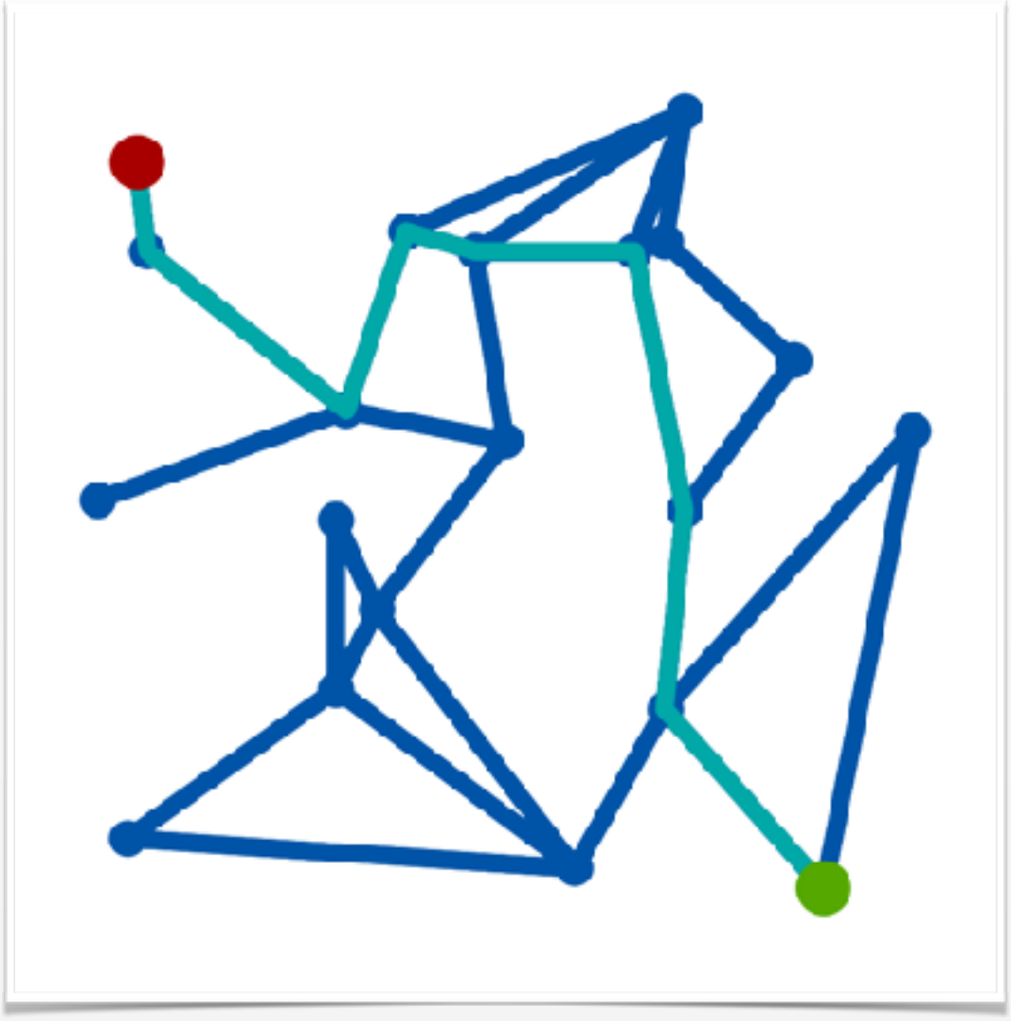
Known network of roads

Canadian Traveler Problem



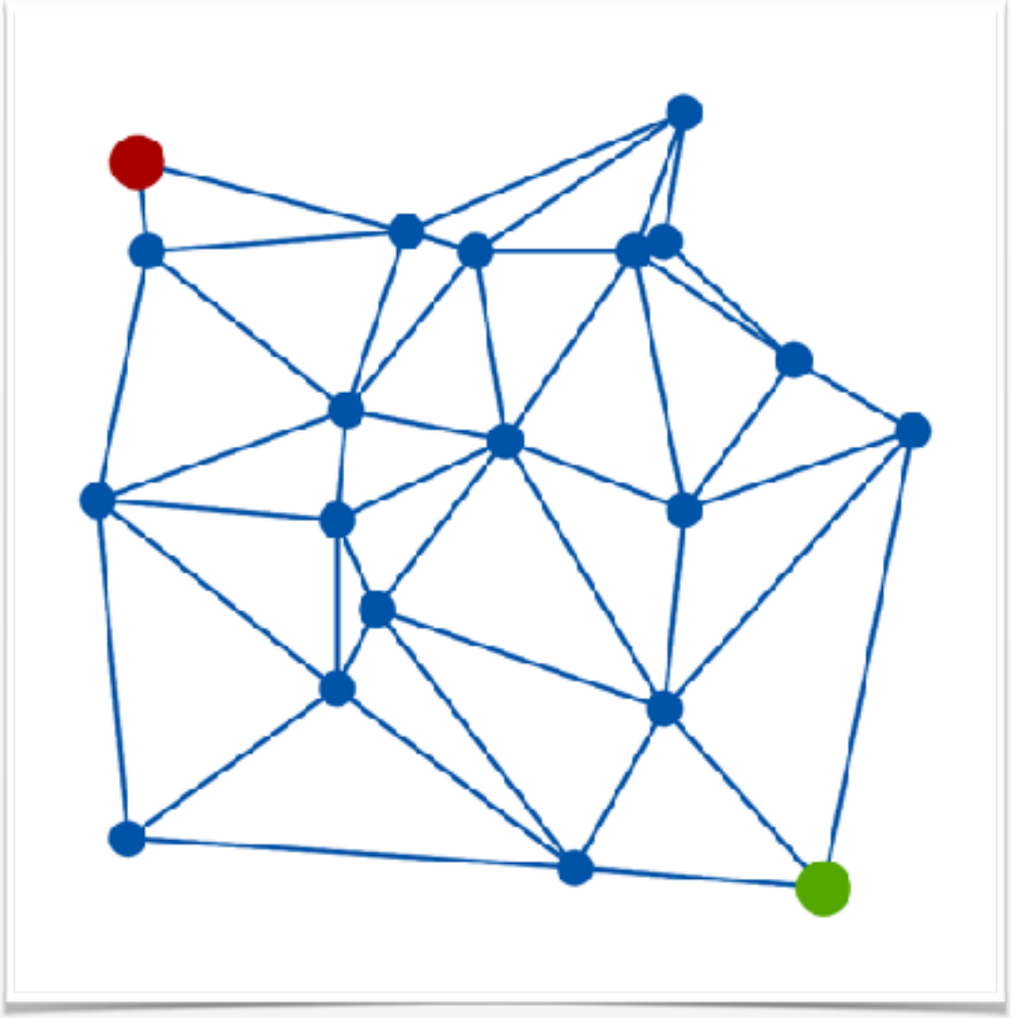
Edges missing at Random

Canadian Traveler Problem



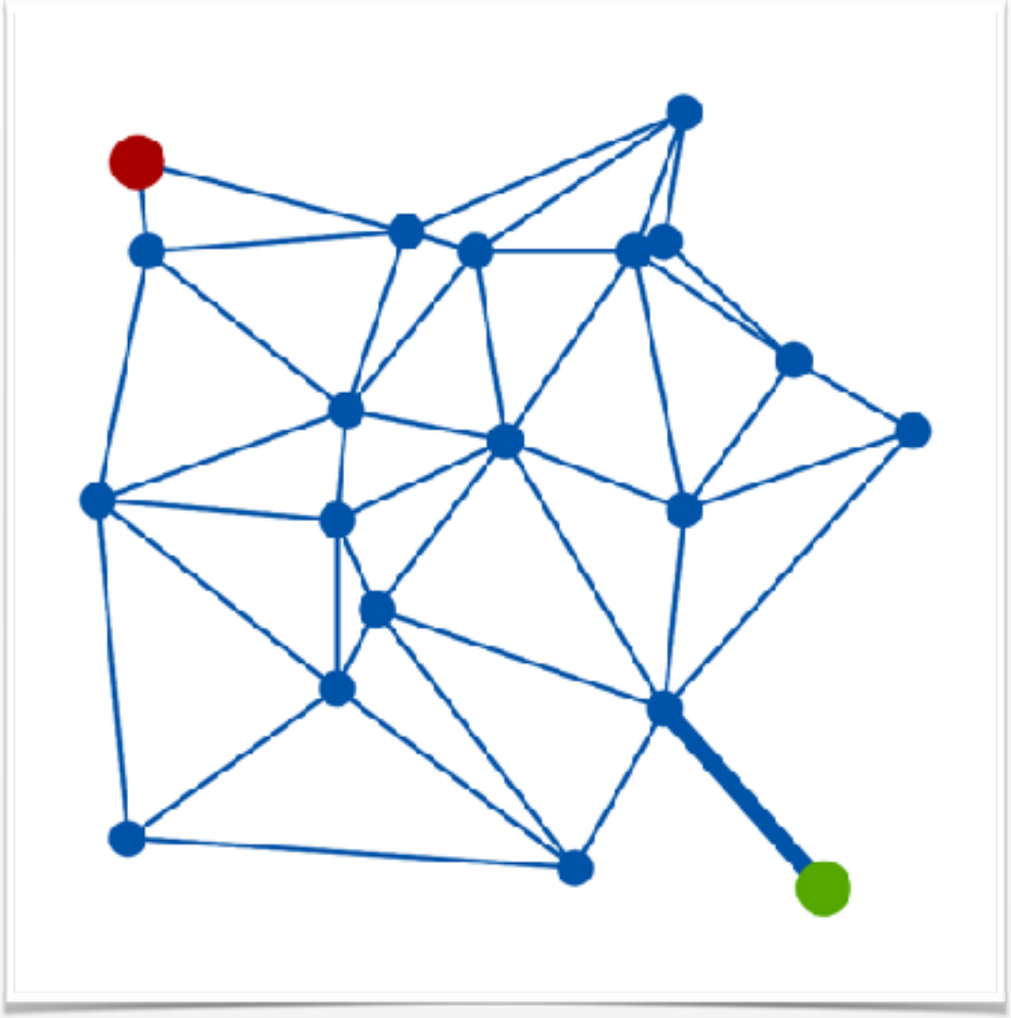
Perfect information:
Dijkstra's algorithm

Canadian Traveler Problem



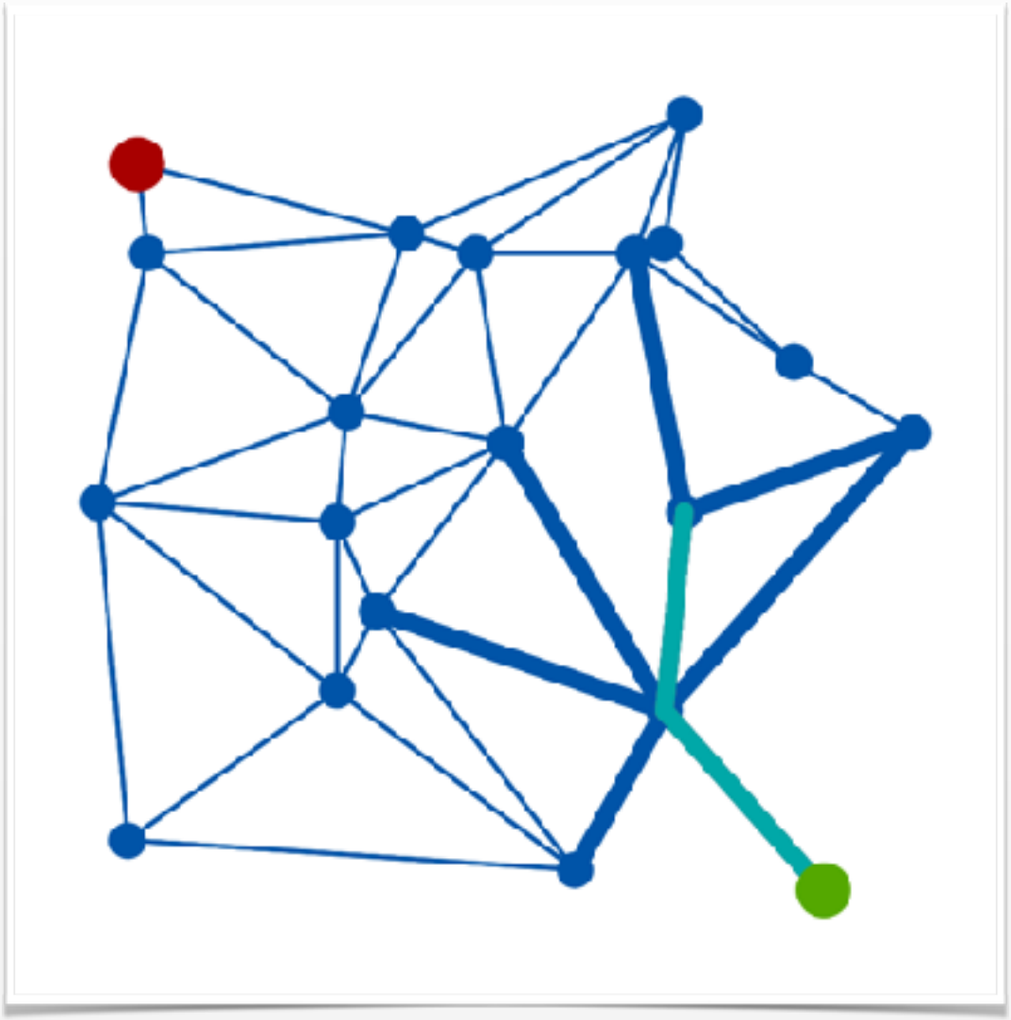
State of roads unknown:
Planning problem

Canadian Traveler Problem



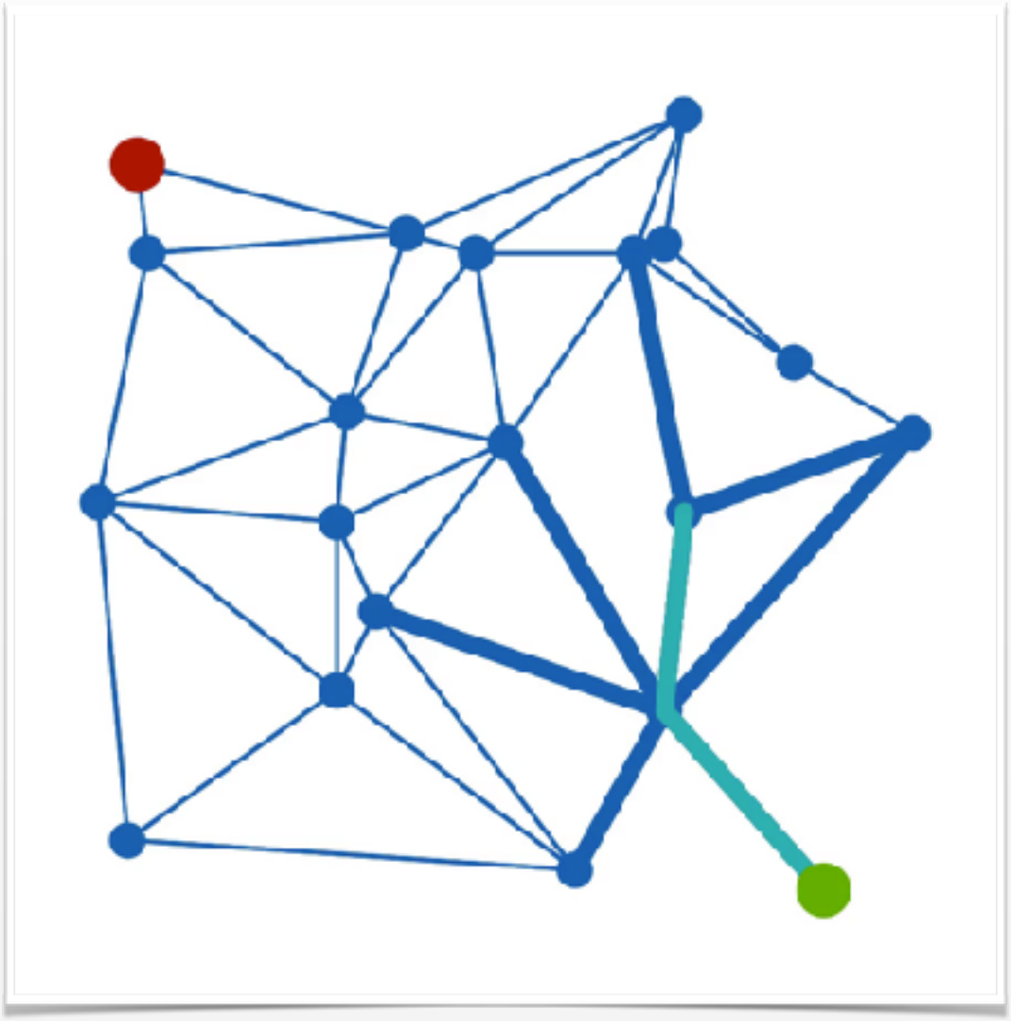
State of roads unknown:
Planning problem

Canadian Traveler Problem



State of roads unknown:
Planning problem

Canadian Traveler Problem



State of roads unknown:
Planning problem

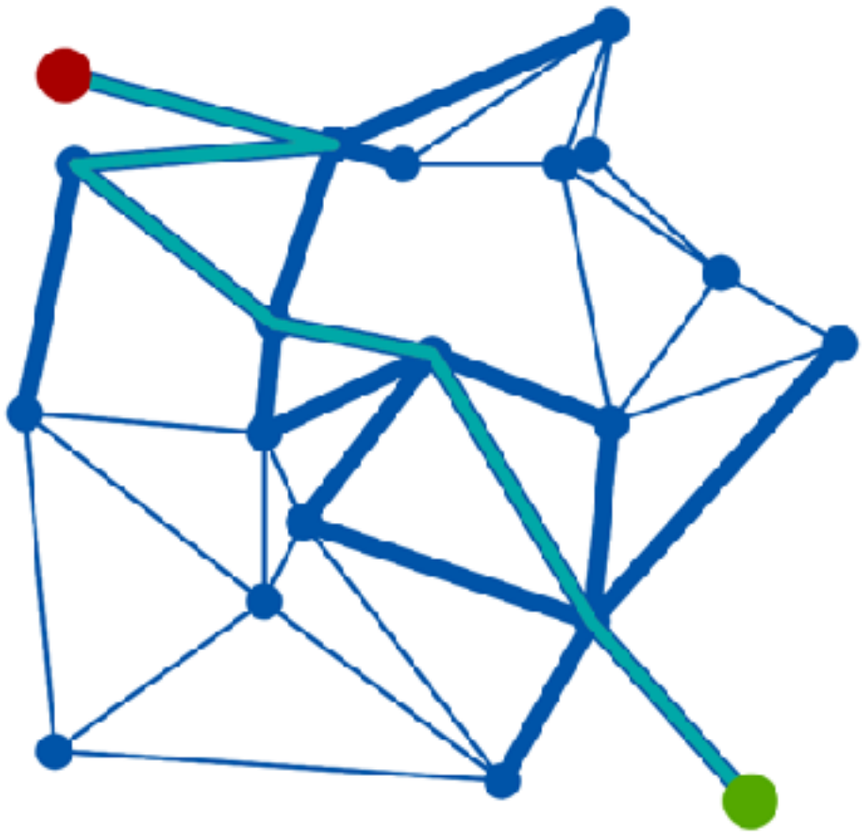
Strawman Policy

- Perform Depth First Search
- Choose unexplored paths uniformly at random
- Guaranteed to reach goal (if accessible)

Planning with Prob. Prog.

- Express policies as probabilistic programs
- Optimize free parameters to maximize expected reward

Policy Search (*upper-level*)



State of roads unknown:
Planning problem

Policy (Probabilistic Program)

$$\theta \sim p(\theta; \lambda)$$

$$a_t \sim p(a_t \mid s_t, \theta)$$

World (Probabilistic Program)

$$s_t \sim p(s_t \mid s_{t-1}, a_t)$$

Maximize Expected Reward

$$x := (a_0, s_1, \ldots, s_T, a_T)$$

$$R(x) := \sum_t -\text{distance}(a_t)$$

$$J(\lambda) := \mathbb{E}_{p(x \mid \lambda)}[R(x)]$$

Probabilistic Programs for Inference

Probabilistic Program

```
(defquery lin-reg [x-vals y-vals]
 (let [a (sample (normal 0 1))
       b (sample (normal 0 1))
       f (fn [x] (+ (* a x) b))]
  (map (fn [x y]
         (observe
          (normal (f x) 0.1) y))
       x-vals y-vals)
  [a b])
```

Joint Probability

$$p(y, x) = p(y \mid x)\,p(x)$$

Likelihood Prior

Special Forms

sample *random* value x

observe *condition* on value y

Probabilistic Programs for Inference

Probabilistic Program

```
(defquery lin-reg [x-vals y-vals]
 (let [a (sample (normal 0 1))
       b (sample (normal 0 1))
       f (fn [x] (+ (* a x) b))]
  (map (fn [x y]
         (observe
          (normal (f x) 0.1) y))
       x-vals y-vals)
  [a b])
```

Special Forms

sample *random* value ***x***
observe *condition* on value ***y***

Posterior Density

$$p(x \mid y) = p(y \mid x)p(x)/p(y)$$

Distribution on **sample** *values given* **observe** *values.*

Probabilistic Programs for Inference

Probabilistic Program

```
(defquery lin-reg [x-vals y-vals]
 (let [a (sample (normal 0 1))
       b (sample (normal 0 1))
       f (fn [x] (+ (* a x) b))]
  (map (fn [x y]
         (observe
          (normal (f x) 0.1) y))
       x-vals y-vals)
  [a b])
```

Special Forms

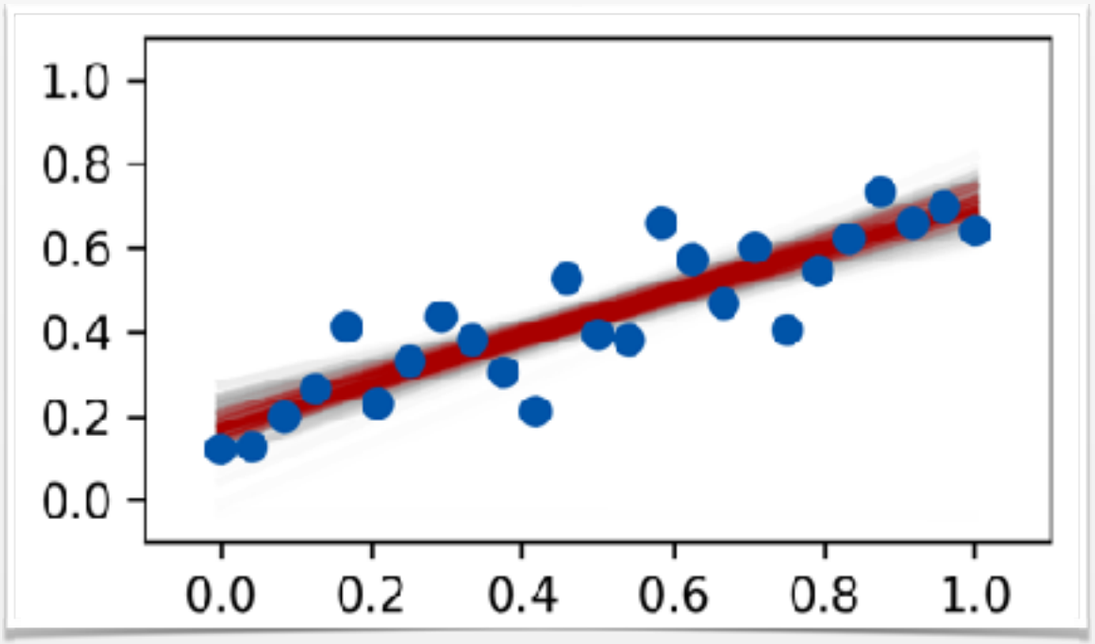
sample *random* value ***x***
observe *condition* on value ***y***

Posterior Density

$$p(x \mid y) = p(y \mid x)p(x)/p(y)$$

Distribution on **sample** *values given* **observe** *values.*

Inference



Probabilistic Programs for Planning

Probabilistic Program

```
(let [θ (sample (policy-prior λ))
      P (make-policy θ)
      x (sample-episode P)]
 (factor (reward x))
 [x θ])
```

Special Forms

sample	*random* value x
factor	*assign* log probability

Conditioning on Reward

$$\exp R(x) \text{ replaces } p(y|x)$$

Policy Search

Maximize expected ***reward****,*
marginalizing over trajectories x*,*
with respect to hyperparameters λ

Bayesian Analogue: Maximum Likelihood, Empirical Bayes

Planning as Inference

Target Density

$$\pi(\boldsymbol{x};\boldsymbol{\phi}) := \gamma(\boldsymbol{x};\boldsymbol{\phi})/Z$$

$$Z := \int \gamma(\boldsymbol{x};\boldsymbol{\phi})\, d\boldsymbol{x} \qquad \text{(intractable integral)}$$

Bayesian Inference

$$\gamma(\boldsymbol{x};\boldsymbol{\phi}) := p(\boldsymbol{y},\boldsymbol{x};\boldsymbol{\phi})$$

$$Z := \int p(\boldsymbol{y},\boldsymbol{x};\boldsymbol{\phi})\, d\boldsymbol{x} = p(\boldsymbol{y};\boldsymbol{\phi})$$

Planning

$$\gamma(\boldsymbol{x};\boldsymbol{\phi}) := \begin{cases} p(\boldsymbol{x};\boldsymbol{\phi})\, R(\boldsymbol{x}) & R(\boldsymbol{x}) \geq 0 \\ p(\boldsymbol{x};\boldsymbol{\phi}) \exp[R(\boldsymbol{x})] & R(\boldsymbol{x}) \in \mathbb{R} \end{cases}$$

$$Z := \begin{cases} \mathbb{E}_{p(\boldsymbol{x};\boldsymbol{\phi})}[R(\boldsymbol{x})] & \text{Expected reward} \\ \mathbb{E}_{p(\boldsymbol{x};\boldsymbol{\phi})} \exp[R(\boldsymbol{x})] & \text{Desirability} \end{cases}$$

[Dayan & Hinton 1997, Todorov 2009, Rawlik et al. 2012, Kappen et al. 2012]

Optimizing Lower Bounds

$$Z(\boldsymbol{\phi}) := \int \gamma(\boldsymbol{x};\boldsymbol{\phi})\, d\boldsymbol{x} = \int q(\boldsymbol{x};\boldsymbol{\lambda}) \frac{\gamma(\boldsymbol{x};\boldsymbol{\phi})}{q(\boldsymbol{x};\boldsymbol{\lambda})}\, d\boldsymbol{x} = \mathbb{E}_{q(\boldsymbol{x};\boldsymbol{\lambda})}\left[\frac{\gamma(\boldsymbol{x};\boldsymbol{\phi})}{q(\boldsymbol{x};\boldsymbol{\lambda})}\right]$$

Approximation: Lower Bound (*from Jensen's Inequality*)

$$\mathscr{L}(\boldsymbol{\phi},\boldsymbol{\lambda}) := \mathbb{E}_{q(\boldsymbol{x};\boldsymbol{\lambda})}\left[\log \frac{\gamma(\boldsymbol{x};\boldsymbol{\phi})}{q(\boldsymbol{x};\boldsymbol{\lambda})}\right] \leq \log \mathbb{E}_{q(\boldsymbol{x};\boldsymbol{\lambda})}\left[\frac{\gamma(\boldsymbol{x};\boldsymbol{\phi})}{q(\boldsymbol{x};\boldsymbol{\lambda})}\right] = \log Z(\boldsymbol{\phi})$$

Variational Inference: $\max_{\boldsymbol{\lambda}} \mathscr{L}(\boldsymbol{\phi},\boldsymbol{\lambda}) = \min_{\boldsymbol{\lambda}} \mathrm{KL}(q(\boldsymbol{x};\boldsymbol{\lambda}) \mid \pi(\boldsymbol{x};\boldsymbol{\phi}))$

Maximum Likelihood: $\max_{\boldsymbol{\phi}} \mathscr{L}(\boldsymbol{\phi},\boldsymbol{\lambda}) = \max_{\boldsymbol{\phi}} \log Z(\boldsymbol{\phi})$

Optimizing Lower Bounds

$$\mathscr{L}(\boldsymbol{\phi}, \boldsymbol{\lambda}) := \mathbb{E}_{q(\boldsymbol{x};\boldsymbol{\lambda})}\left[\log \frac{\gamma(\boldsymbol{x};\boldsymbol{\phi})}{q(\boldsymbol{x};\boldsymbol{\lambda})}\right] \leq \log Z$$

Policy Search

$$\gamma(\boldsymbol{x};\boldsymbol{\phi}) = p(\boldsymbol{x};\boldsymbol{\phi}) \exp[R(\boldsymbol{x})]$$

$$q(\boldsymbol{x};\boldsymbol{\lambda}) = p(\boldsymbol{x};\boldsymbol{\lambda})$$

$$Z = \mathbb{E}_{p(\boldsymbol{x};\boldsymbol{\phi})}[\exp R(\boldsymbol{x})]$$

$$\mathscr{L}(\boldsymbol{\lambda}, \boldsymbol{\lambda}) = \mathbb{E}_{p(\boldsymbol{x};\boldsymbol{\lambda})}\left[\log \frac{\cancel{p(\boldsymbol{x}, \boldsymbol{\lambda})} \exp R(\boldsymbol{x})}{\cancel{p(\boldsymbol{x}, \boldsymbol{\lambda})}}\right]$$

$$= \mathbb{E}_{p(\boldsymbol{x};\boldsymbol{\lambda})}[R(\boldsymbol{x})]$$

$$= J(\boldsymbol{\lambda})$$

Core Problem: Gradient Estimation

Easy: Gradient with respect to generative parameters $\boldsymbol{\phi}$

$$\nabla_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\lambda}) := \nabla_{\boldsymbol{\phi}} \mathbb{E}_{q(\boldsymbol{x};\boldsymbol{\lambda})}\left[\log \frac{\gamma(\boldsymbol{x};\boldsymbol{\phi})}{q(\boldsymbol{x};\boldsymbol{\lambda})}\right] = \mathbb{E}_{q(\boldsymbol{x};\boldsymbol{\lambda})}\left[\nabla_{\boldsymbol{\phi}} \log \gamma(\boldsymbol{x};\boldsymbol{\phi})\right]$$

$$\simeq \frac{1}{S}\sum_{s=1}^{S} \nabla_{\boldsymbol{\phi}} \log \gamma(\boldsymbol{x}^{(s)};\boldsymbol{\phi}) \qquad \boldsymbol{x}^{(s)} \sim q(\boldsymbol{x};\boldsymbol{\lambda})$$

Problem: generative model $\gamma(\boldsymbol{x}\,;\,\boldsymbol{\phi})$ must be differentiable

(*<u>not</u> generally true for probabilistic programs*)

Trick 1: Likelihood-ratio Estimators

Harder: Gradient w.r.t generative parameters $\boldsymbol{\lambda}$

$$\nabla_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\lambda}) := \nabla_{\boldsymbol{\lambda}} \mathbb{E}_{q(\boldsymbol{x};\boldsymbol{\lambda})} \left[\log \frac{\gamma(\boldsymbol{x};\boldsymbol{\phi})}{q(\boldsymbol{x};\boldsymbol{\lambda})} \right]$$

$$= \int \nabla_{\boldsymbol{\lambda}} q(\boldsymbol{x};\boldsymbol{\lambda}) \log \left[\frac{\gamma(\boldsymbol{x};\boldsymbol{\phi})}{q(\boldsymbol{x};\boldsymbol{\lambda})} \right] - q(\boldsymbol{x};\boldsymbol{\lambda}) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{x};\boldsymbol{\lambda}) \, d\boldsymbol{x}$$

Trick 1 (REINFORCE)

$$\nabla_{\boldsymbol{\lambda}} q(\boldsymbol{x};\boldsymbol{\lambda}) = q(\boldsymbol{x};\boldsymbol{\lambda}) \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{x};\boldsymbol{\lambda})$$

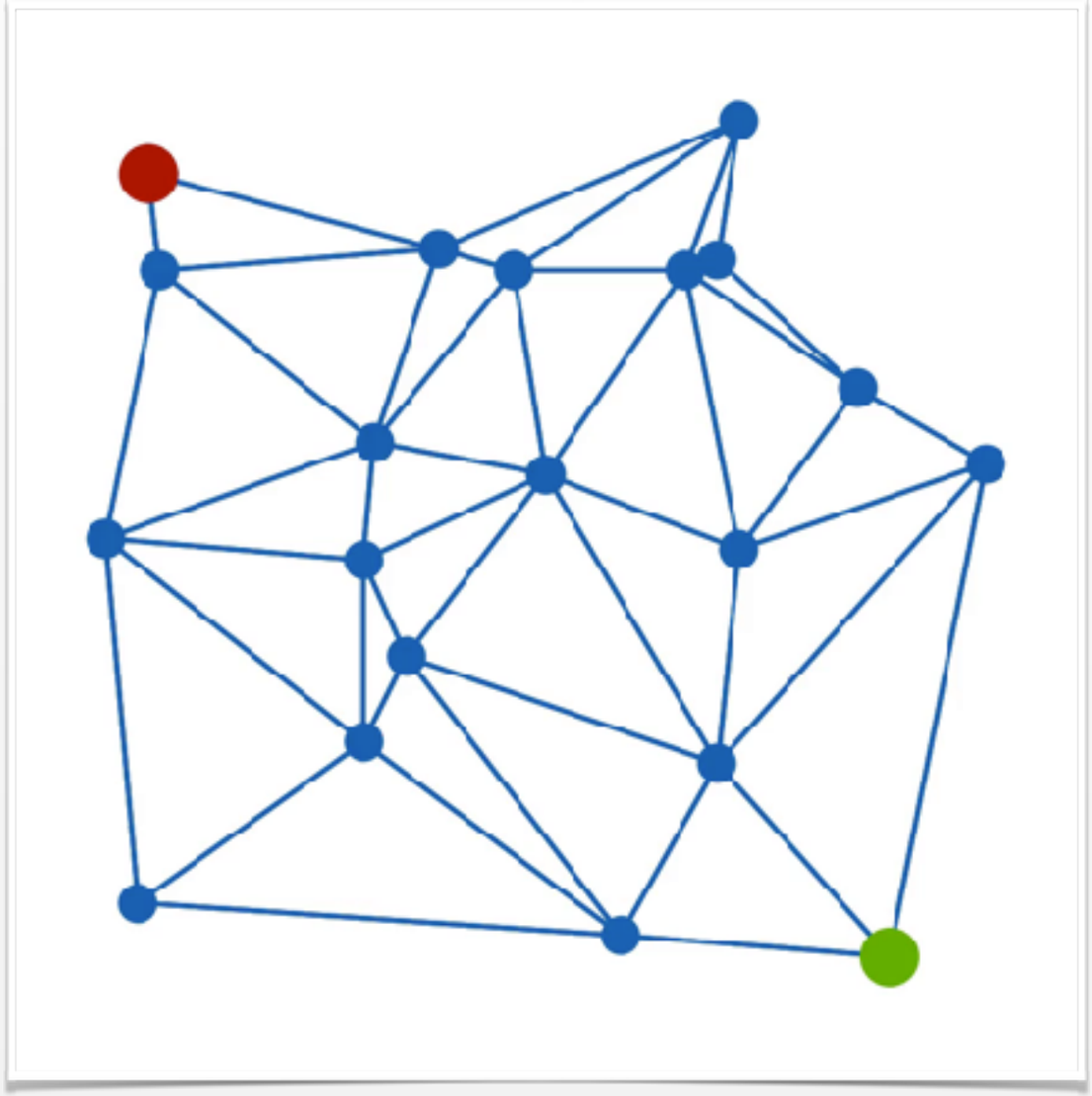
Requires grad of $\log q(\boldsymbol{x};\boldsymbol{\phi})$ but not $\gamma(\boldsymbol{x};\boldsymbol{\phi})$

$$\simeq \frac{1}{S} \sum_{s=1}^{S} \nabla_{\boldsymbol{\lambda}} \log q(\boldsymbol{x}^{(s)};\boldsymbol{\lambda}) \left[\log \frac{\gamma(\boldsymbol{x}^{(s)};\boldsymbol{\phi})}{q(\boldsymbol{x}^{(s)};\boldsymbol{\lambda})} - \hat{b} \right] \qquad \boldsymbol{x}^{(s)} \sim q(\boldsymbol{x};\boldsymbol{\lambda})$$

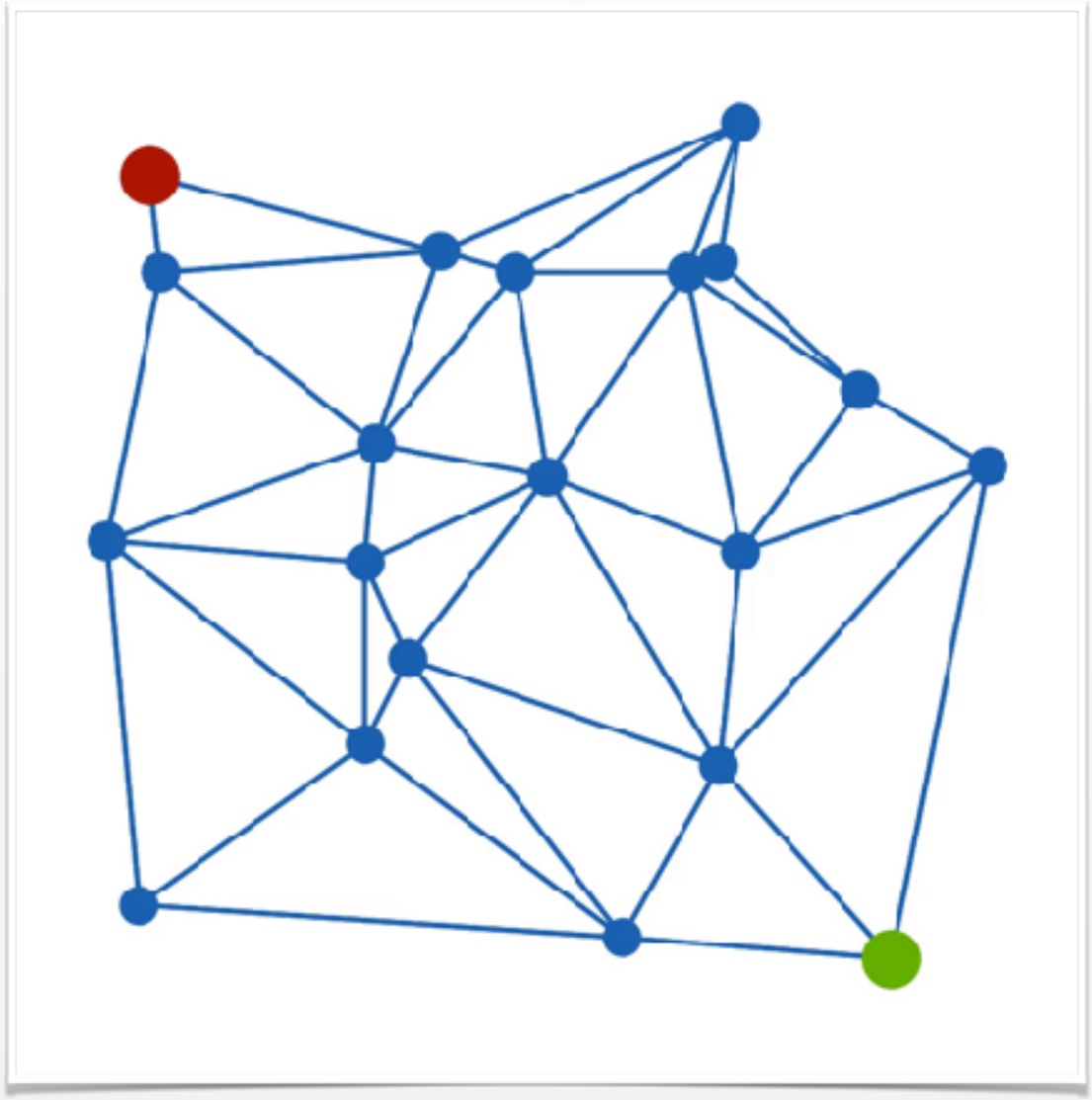
[see: Williams 1992, Wingate & Weber 2013, Ranganath et al. 2014]

Canadian Traveler Problem

Uniform Policy



Learned Policy (REINFORCE)



Limitation: Need 1000 samples per gradient step

Trick 2: Reparameterization

Idea: Define parameterized transformation $\boldsymbol{x}=\boldsymbol{X}(\boldsymbol{\epsilon},\boldsymbol{\lambda})$

$$\left.\begin{aligned}\boldsymbol{\epsilon} &\sim p(\boldsymbol{\epsilon}) \\ \boldsymbol{x} &= \boldsymbol{X}(\boldsymbol{\epsilon};\boldsymbol{\lambda})\end{aligned}\right\} \boldsymbol{x} \sim q(\boldsymbol{x};\boldsymbol{\lambda}) \qquad \left.\begin{aligned}\boldsymbol{\epsilon} &\sim \text{Normal}(\mathbf{0}, \boldsymbol{I}) \\ \boldsymbol{x} &= \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}\end{aligned}\right\} \boldsymbol{x} \sim \text{Normal}(\boldsymbol{\mu}, \boldsymbol{\sigma}\boldsymbol{I})$$

Advantage: No longer need REINFORCE trick

$$\nabla_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\lambda}) := \nabla_{\boldsymbol{\lambda}} \mathbb{E}_{p(\boldsymbol{\epsilon})} \left[\log \frac{\gamma(\boldsymbol{X}(\boldsymbol{\epsilon};\boldsymbol{\lambda});\boldsymbol{\phi})}{q(\boldsymbol{X}(\boldsymbol{\epsilon};\boldsymbol{\lambda});\boldsymbol{\lambda})} \right] = \mathbb{E}_{p(\boldsymbol{\epsilon})} \left[\nabla_{\boldsymbol{\lambda}} \log \frac{\gamma(\boldsymbol{X}(\boldsymbol{\epsilon};\boldsymbol{\lambda});\boldsymbol{\phi})}{q(\boldsymbol{X}(\boldsymbol{\epsilon};\boldsymbol{\lambda});\boldsymbol{\lambda})} \right]$$

$$\simeq \frac{1}{S} \sum_{s=1}^{S} \underbrace{\nabla_{\boldsymbol{\lambda}} \log \frac{\gamma(\boldsymbol{X}(\boldsymbol{\epsilon}^{(s)};\boldsymbol{\lambda});\boldsymbol{\phi})}{q(\boldsymbol{X}(\boldsymbol{\epsilon}^{(s)};\boldsymbol{\lambda});\boldsymbol{\lambda})}}_{\text{Compute with AD}} \qquad \boldsymbol{\epsilon}^{(s)} \sim p(\boldsymbol{\epsilon})$$

Deep Probabilistic Programs

Encoder $x \sim q(x; y, \lambda)$

```
class Encoder(torch.nn.Module):
  def __init__(self, x_sz, h_sz, y_sz, z_sz):
    # intializes layers: h, y_log_weights, ...
    ...

  def forward(self, x, y_values=None):
    q = probtorch.Trace()
    h = self.h(x)
    y = q.concrete(
          self.y_log_weights(h), 0.66,
          value=y_values, name='y')
    hy = torch.cat([h, y], -1)
    z = q.normal(self.z_mean(hy),
                 self.z_std(hy),
                 name='z')
    return q
```

Decoder $\gamma(x; y, \phi)$

```
class Decoder(torch.nn.Module):
  def __init__(self, x_sz, h_sz, y_sz, z_sz):
    # intializes layers: h, x_mean, ...
    ...

  def forward(self, x, q):
    p = probtorch.Trace()
    y = p.concrete(self.y_log_weights, 0.66,
                   value=q['y'], name='y')
    z = p.normal(0.0, 1.0,
                 value=q['z'], name='z')
    h = self.h(torch.cat([y, z], -1))
    x = p.loss(self.bce,
               self.x_mean(h), x,
               name='x')
    return p
```

Edward

https://github.com/bleilab/edward

Probabilistic Torch

PROB TORCH

https://github.com/probtorch/probtorch

Pyro

https://github.com/uber/pyro

Gradient Estimation for Probabilistic Programs

Differentiable?		Generative Model $\gamma(x; y, \phi)$: No	Generative Model $\gamma(x; y, \phi)$: Yes
Approximating Distribution $q(x; y, \lambda)$: No		Infer.NET	Relatively Unexplored ↑
Approximating Distribution $q(x; y, \lambda)$: Yes		BBVI ↓ Inf. Comp	ProbTorch, Edward, Pyro, Stan

Learning neural approximations [Ritchie et al. 2016, Le et al. 2017]

RELAX: REINFORCE + Differentiable surrogate [Grathwohl et al. 2017]

Algorithmic Proposals (e.g. incorporate MCTS)

Contributors

Anglican, Policy Search, Inference as Planning

David Tolpin

Brooks Paige

Hongseok Yang

Frank Wood

Tom Rainforth

Probabilistic Torch

Brooks Paige

Siddharth N.

Alban Desmaison

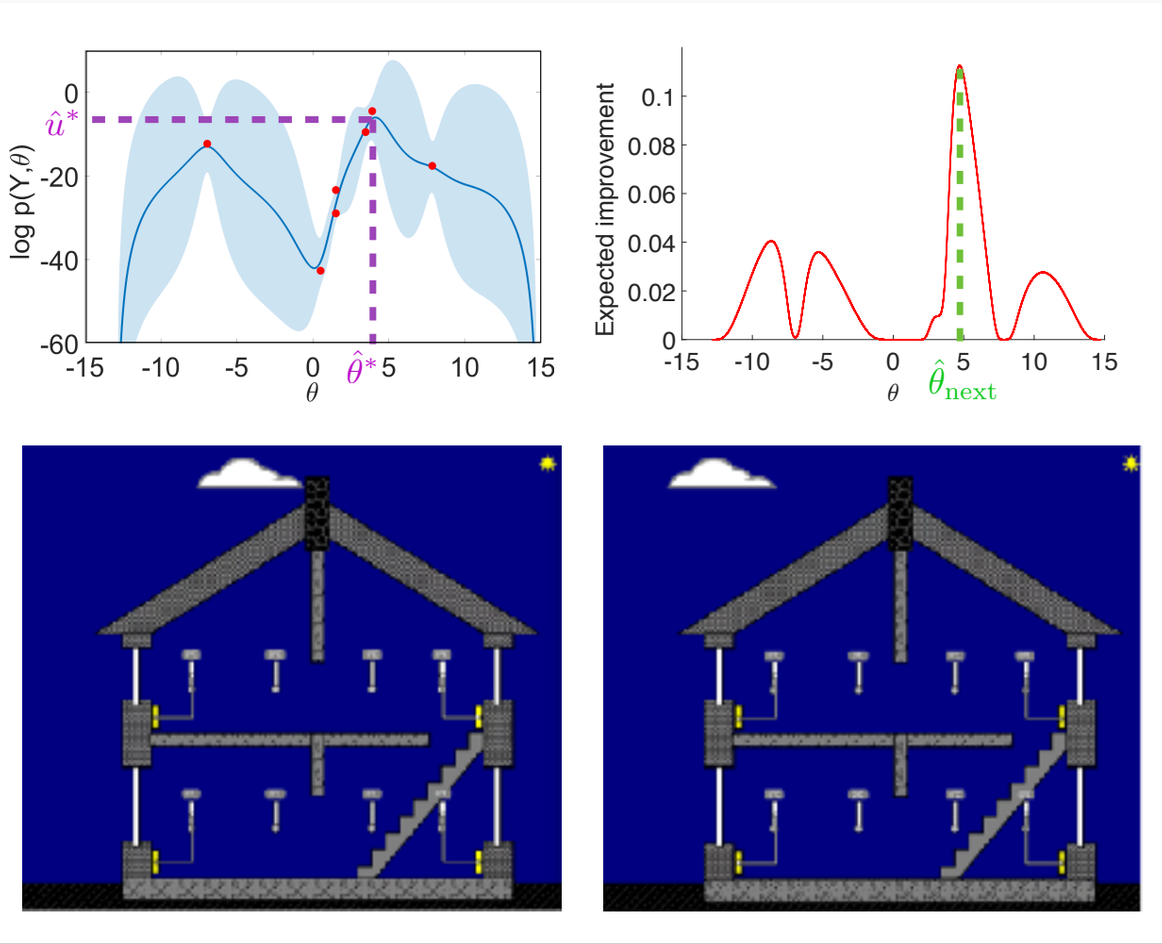
Alican Bozkurt

Babak Esmaeili

We’re hiring: Faculty, Postdocs, Phd Students

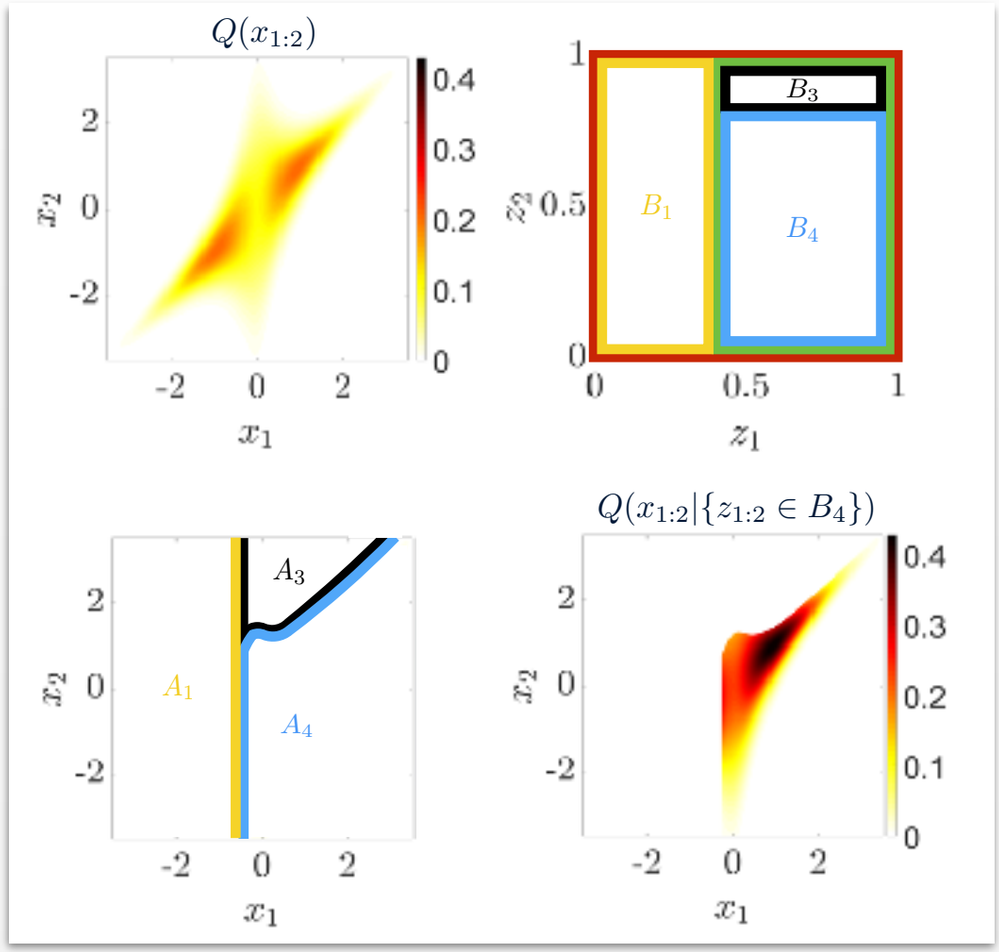
Exploration vs Exploitation in Inference

Bayesian Optimization



Idea: Use probabilistic programs to define marginal MAP problems [Rainforth et al. 2016]

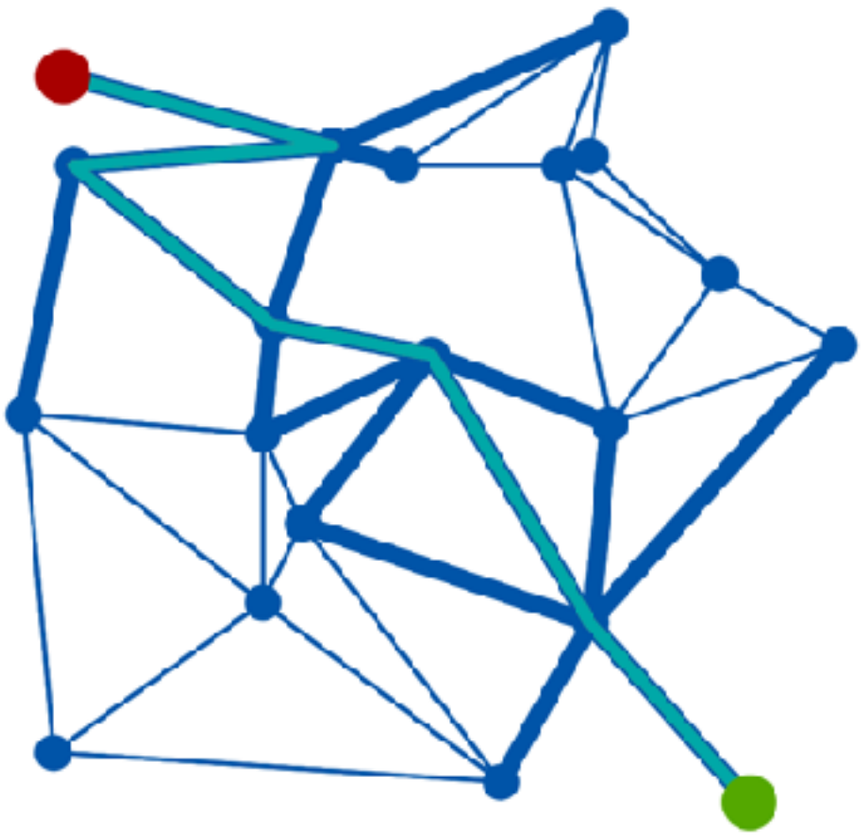
Inference Trees



Idea: Partition space, use MCTS to allocate computation [Rainforth et al. 2017]

Implementation Example

Canadian Traveler Problem

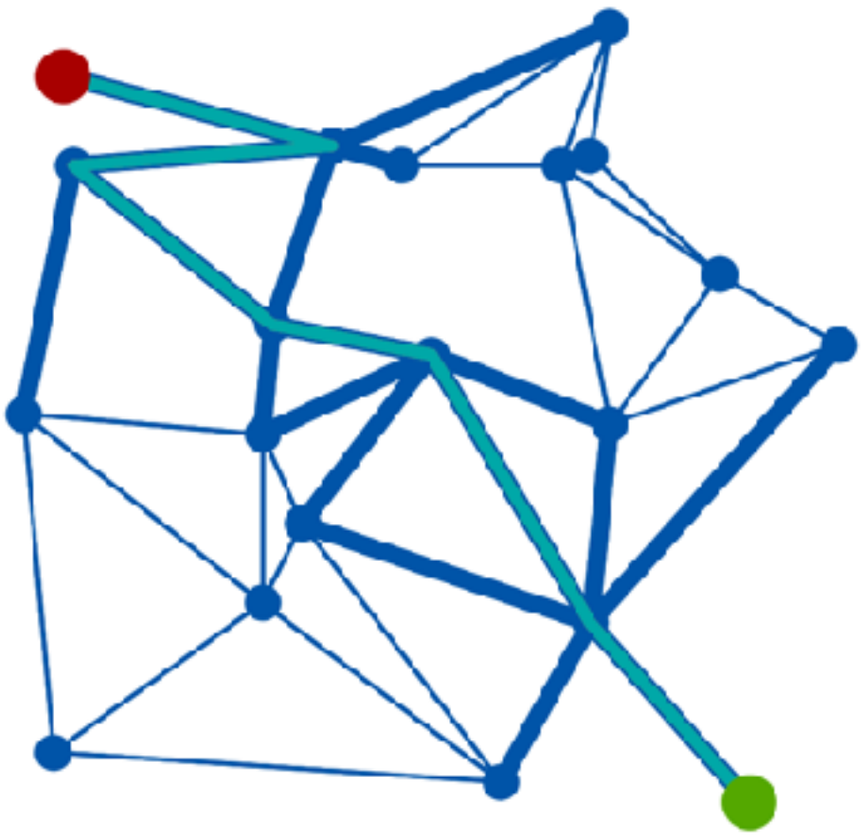


Model Specification

```
(defquery ctp
 [graph start target
  open-prob make-policy]
 (let [;; sample open/closed edges
       instance (simulate-weather
                 graph open-prob)
       ;; do depth-first search
       distance (dfs-agent
                 instance
                 start target
                 (make-policy))]
   ;; weigh according to distance,
   ;; reject when not connected
   (factor (- (or distance inf)))
   (predict :distance distance))))
```

Implementation Example

Canadian Traveler Problem

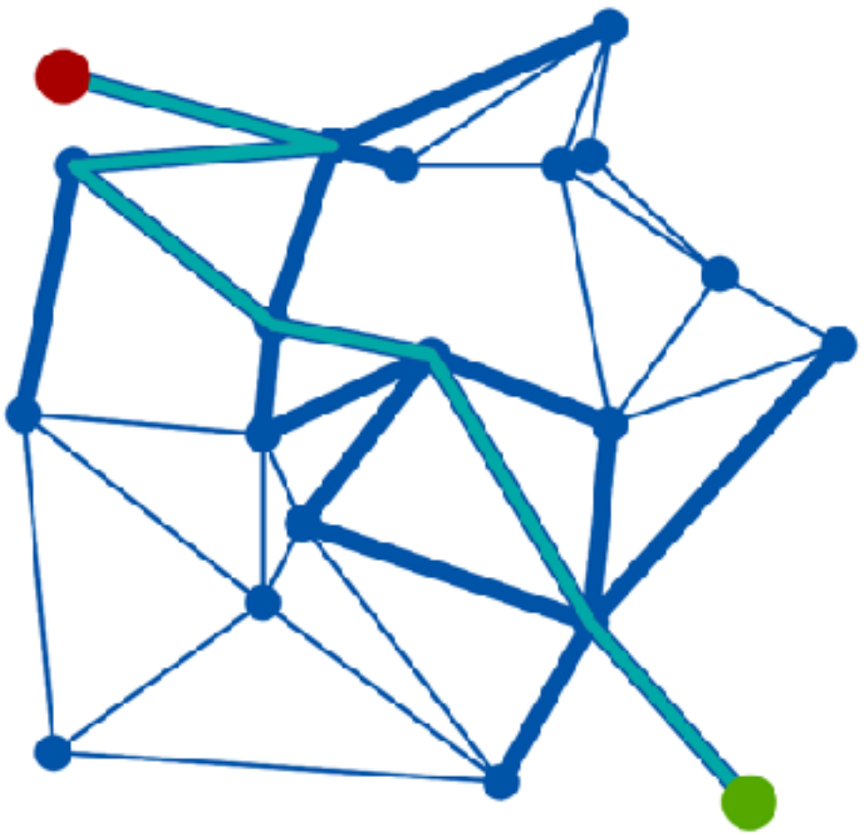


Model Specification

```
(defquery ctp
 [graph start target
  open-prob make-policy]
 (let [;; sample open/closed edges
       instance (simulate-weather
                 graph open-prob)
       ;; do depth-first search
       distance (dfs-agent
                 instance
                 start target
                 (make-policy))]
   ;; weigh according to distance,
   ;; reject when not connected
   (factor (- (or distance inf)))
   (predict :distance distance))))
```

Implementation Example

Canadian Traveler Problem

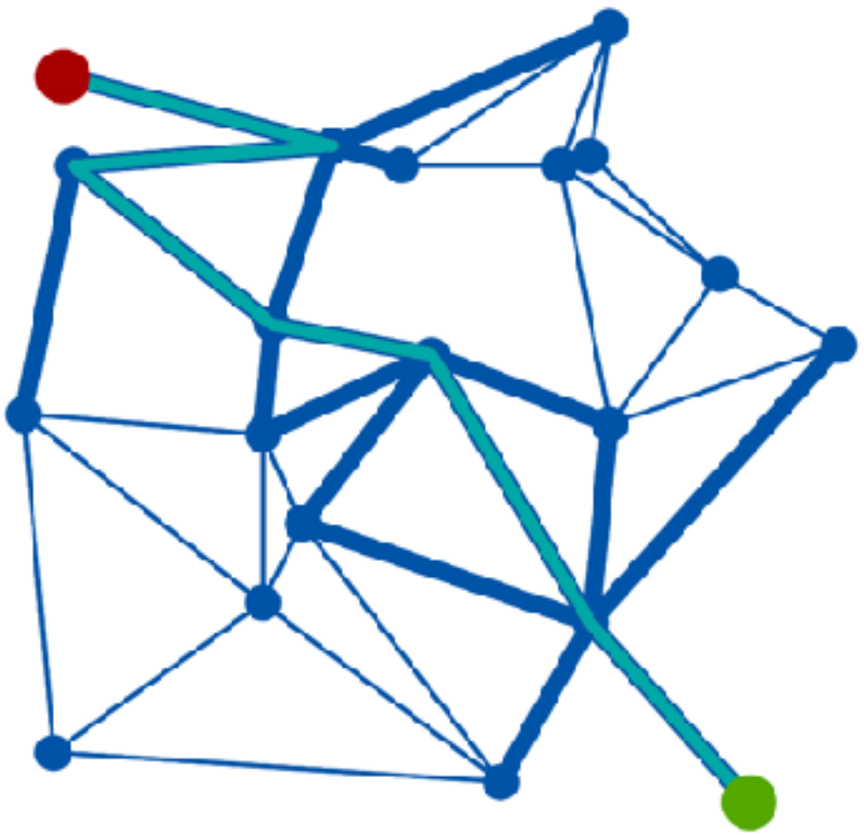


Model Specification

```
(defquery ctp
 [graph start target
  open-prob make-policy]
 (let [;; sample open/closed edges
       instance (simulate-weather
                 graph open-prob)
       ;; do depth-first search
       distance (dfs-agent
                 instance
                 start target
                 (make-policy))]
   ;; weigh according to distance,
   ;; reject when not connected
   (factor (- (or distance inf)))
   (predict :distance distance))))
```

Implementation Example

Canadian Traveler Problem

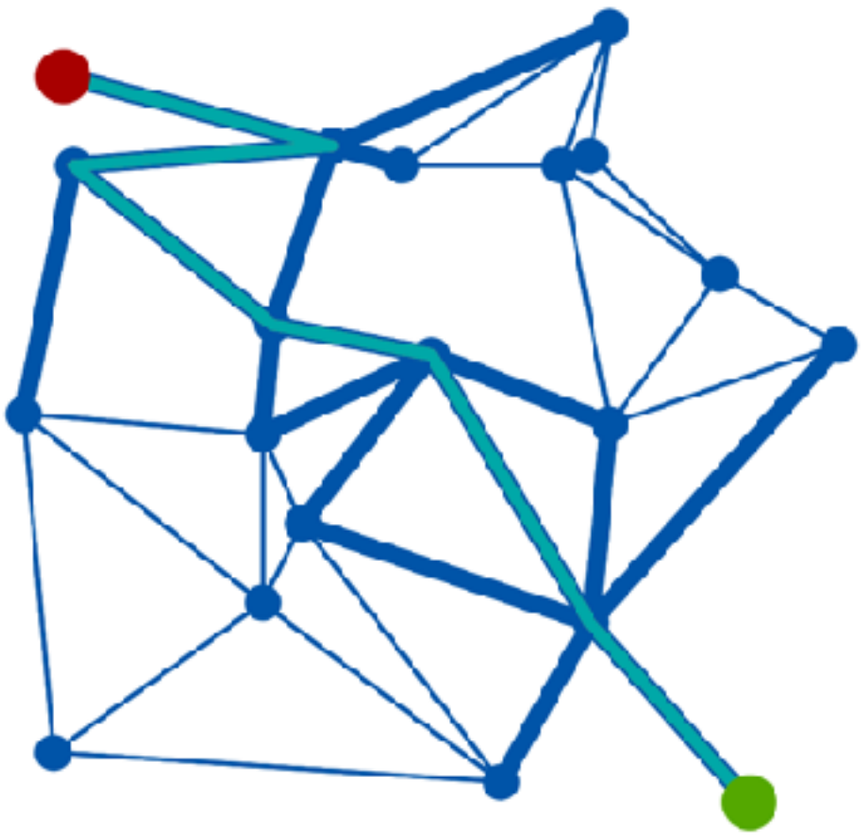


Model Specification

```
(defquery ctp
 [graph start target
  open-prob make-policy]
 (let [;; sample open/closed edges
       instance (simulate-weather
                 graph open-prob)
       ;; do depth-first search
       distance (dfs-agent
                 instance
                 start target
                 (make-policy))]
   ;; weigh according to distance,
   ;; reject when not connected
   (factor (- (or distance inf)))
   (predict :distance distance))))
```

Implementation Example

Canadian Traveler Problem

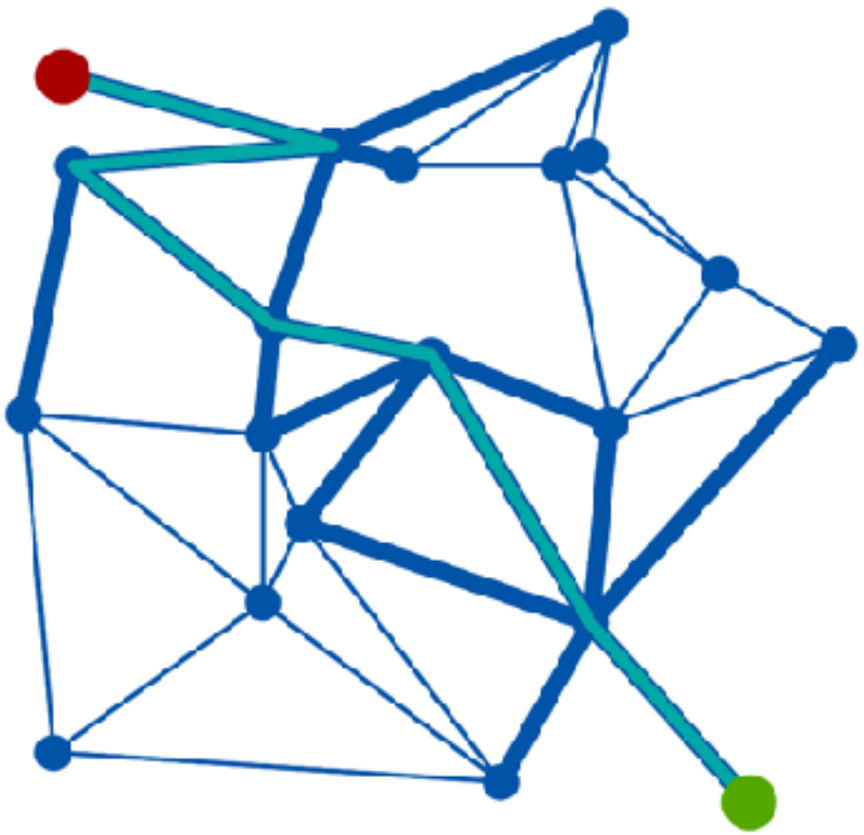


Model Specification

```
(defquery ctp
 [graph start target
  open-prob make-policy]
 (let [;; sample open/closed edges
       instance (simulate-weather
                 graph open-prob)
       ;; do depth-first search
       distance (dfs-agent
                 instance
                 start target
                 (make-policy))]
   ;; weigh according to distance,
   ;; reject when not connected
   (factor (- (or distance inf)))
   (predict :distance distance))))
```

Implementation Example

Canadian Traveler Problem

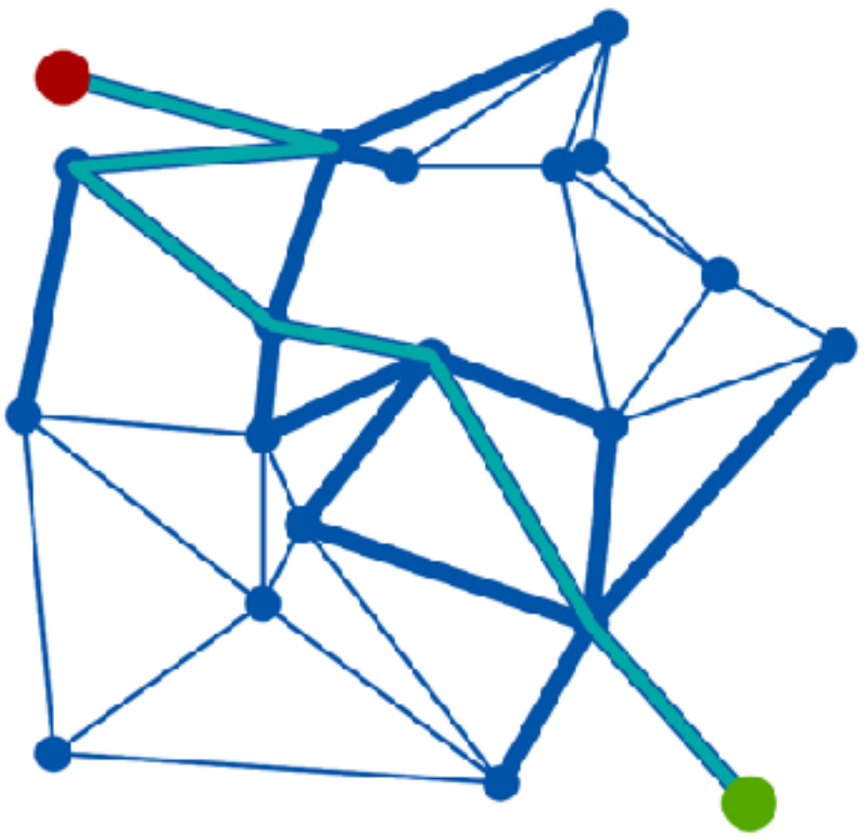


Model Specification

```
(defquery ctp
 [graph start target
  open-prob make-policy]
 (let [;; sample open/closed edges
       instance (simulate-weather
                 graph open-prob)
       ;; do depth-first search
       distance (dfs-agent
                 instance
                 start target
                 (make-policy))]
   ;; weigh according to distance,
   ;; reject when not connected
   (factor (- (or distance inf)))
   (predict :distance distance))))
```

Implementation Example

Canadian Traveler Problem



Model Specification

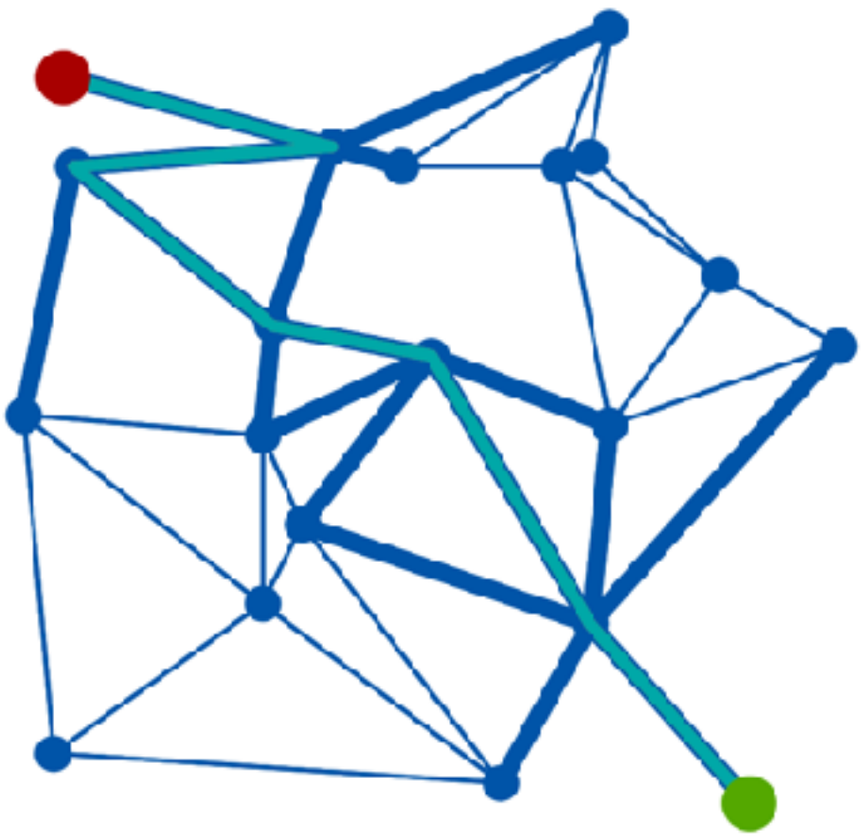
```
(defm make-random-policy []
 (fn policy [u opts]
   (sample (uniform opts))))
```

Utility-based Policy

```
(defm make-edge-policy []
 (let [Q (mem (fn [u v]
                (sample
                 (learn [u v]
                  (gamma 1. 1.))))))]
  (fn policy [u opts]
   (let [Qs (map (fn [v]
                   (Q u v)) opts)]
    (argmax (zipmap opts Qs))))))
```

Implementation Example

Canadian Traveler Problem



Model Specification

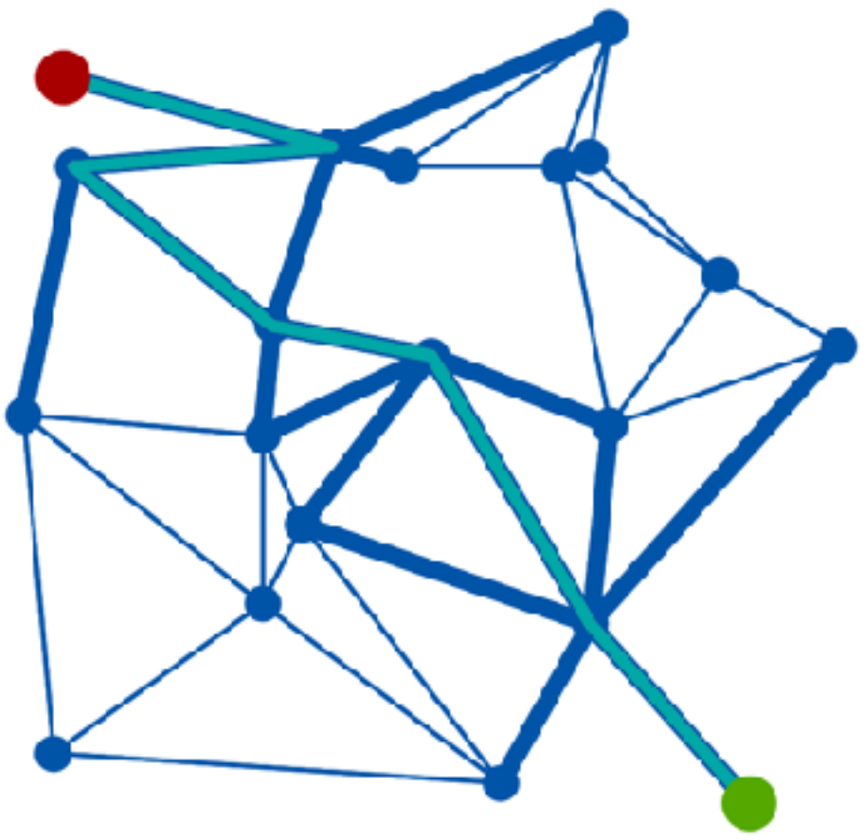
```
(defm make-random-policy []
 (fn policy [u opts]
   (sample (uniform opts))))
```

Utility-based Policy

```
(defm make-edge-policy []
 (let [Q (mem (fn [u v]
                (sample
                 (learn [u v]
                  (gamma 1. 1.)))))]
  (fn policy [u opts]
   (let [Qs (map (fn [v]
                   (Q u v)) opts)]
    (argmax (zipmap opts Qs))))))
```

Implementation Example

Canadian Traveler Problem



Model Specification

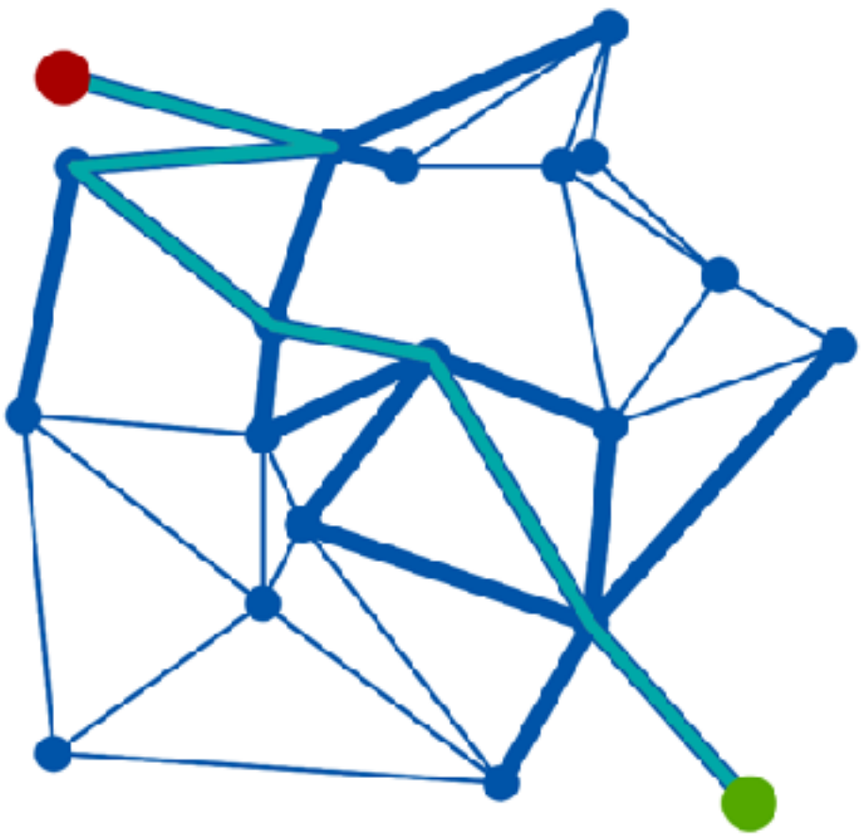
```
(defm make-random-policy []
 (fn policy [u opts]
   (sample (uniform opts))))
```

Utility-based Policy

```
(defm make-edge-policy []
 (let [Q (mem (fn [u v]
                (sample
                 (learn [u v]
                  (gamma 1. 1.)))))]
  (fn policy [u opts]
   (let [Qs (map (fn [v]
                   (Q u v)) opts)]
    (argmax (zipmap opts Qs))))))
```

Implementation Example

Canadian Traveler Problem



Model Specification

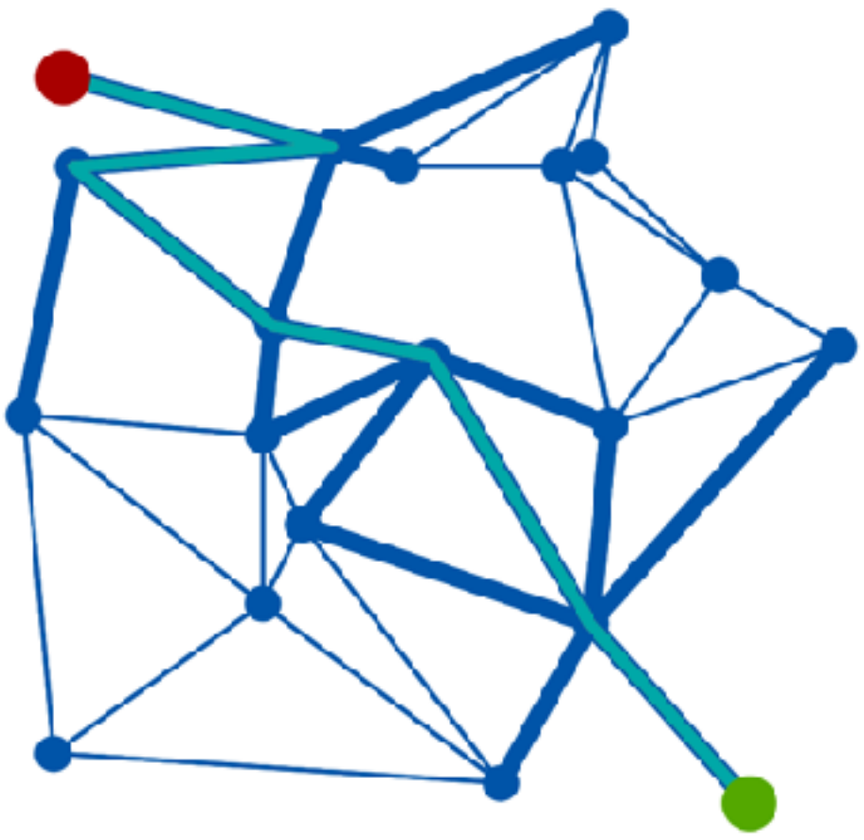
```
(defm make-random-policy []
 (fn policy [u opts]
   (sample (uniform opts))))
```

Utility-based Policy

```
(defm make-edge-policy []
 (let [Q (mem (fn [u v]
                (sample
                 (learn [u v]
                  (gamma 1. 1.)))))]
  (fn policy [u opts]
   (let [Qs (map (fn [v]
                   (Q u v)) opts)]
    (argmax (zipmap opts Qs))))))
```

Implementation Example

Canadian Traveler Problem



Model Specification

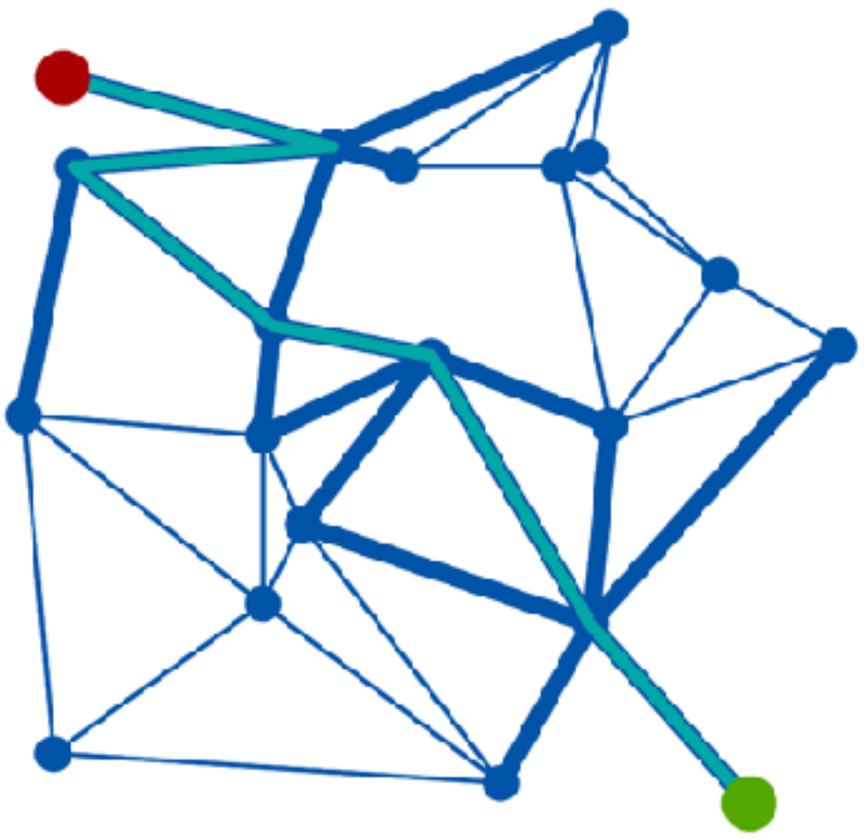
```
(defm make-random-policy []
 (fn policy [u opts]
   (sample (uniform opts))))
```

Utility-based Policy

```
(defm make-edge-policy []
 (let [Q (mem (fn [u v]
               (sample
                (learn [u v]
                 (gamma 1. 1.)))))]
  (fn policy [u opts]
   (let [Qs (map (fn [v]
                   (Q u v)) opts)]
    (argmax (zipmap opts Qs))))))
```

Implementation Example

Canadian Traveler Problem



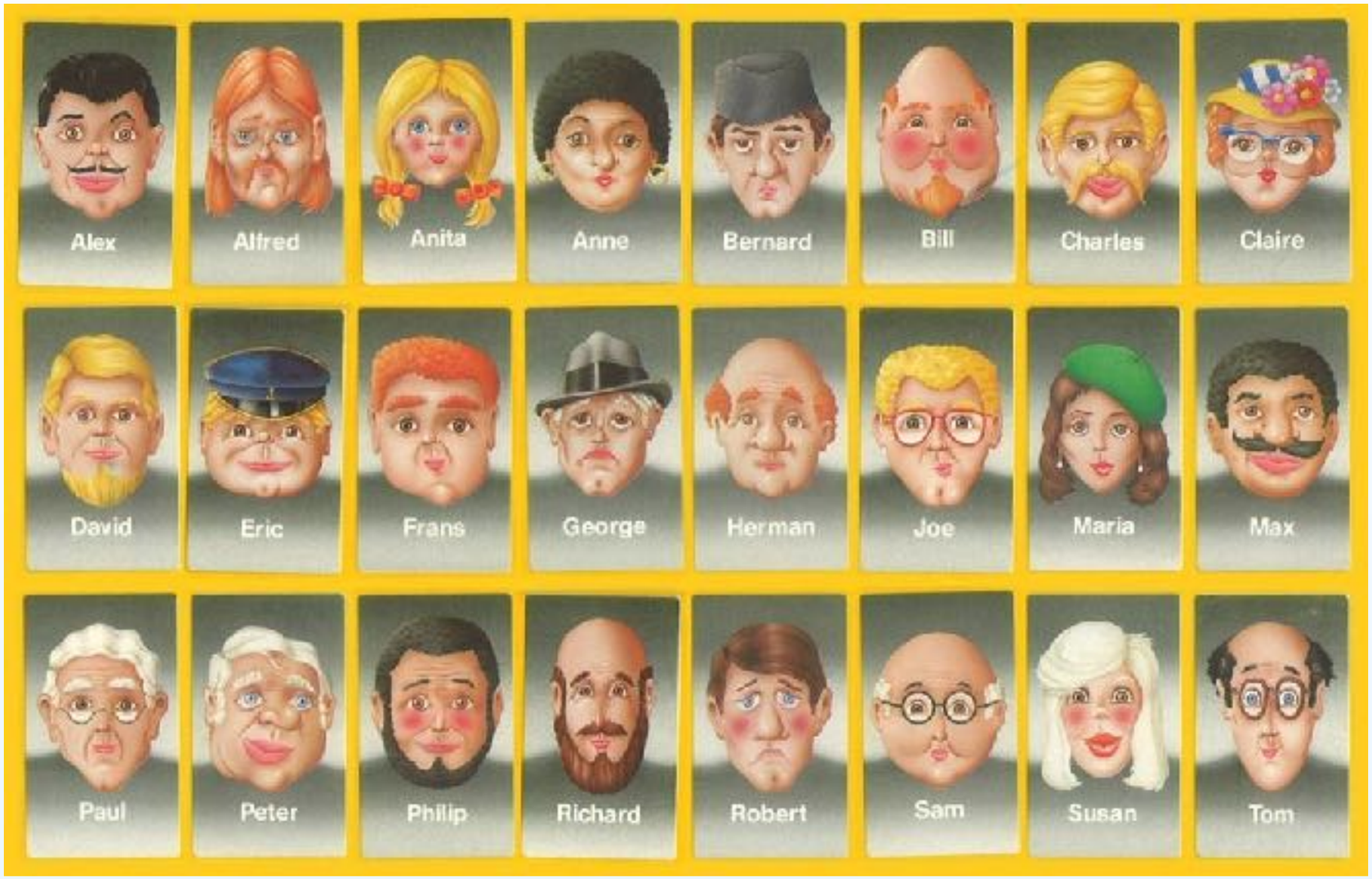
Model Specification

```
(defm make-random-policy []
 (fn policy [u opts]
   (sample (uniform opts))))
```

Utility-based Policy

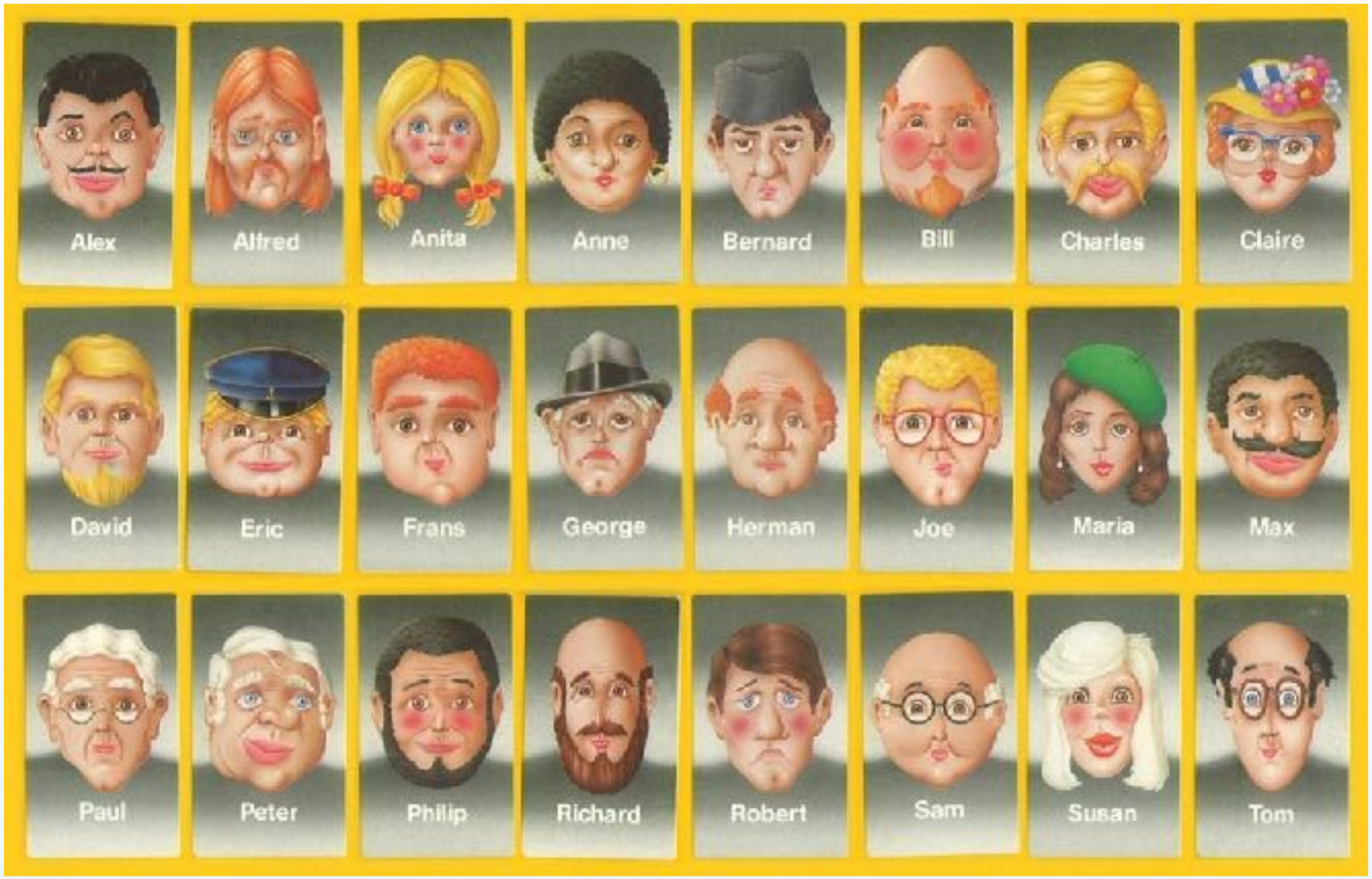
```
(defm make-edge-policy []
 (let [Q (mem (fn [u v]
                (sample
                 (learn [u v]
                  (gamma 1. 1.)))))]
  (fn policy [u opts]
   (let [Qs (map (fn [v]
                   (Q u v)) opts)]
    (argmax (zipmap opts Qs))))))
```

Example: Guess Who



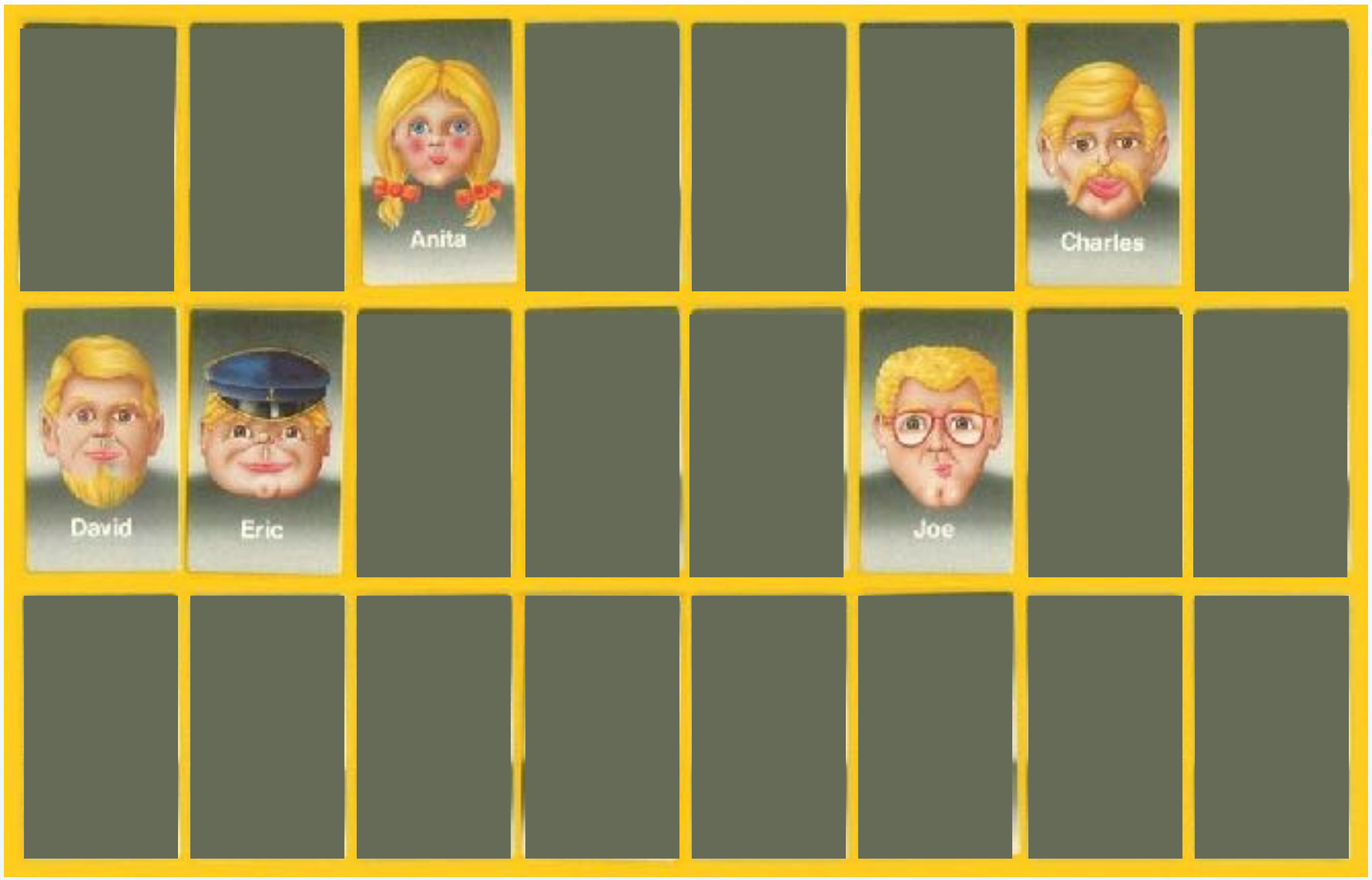
Premise: Player asks questions
to identify an individual in a set

Example: Guess Who



Question: Does the person have **blonde hair**?

Example: Guess Who



Question: Does the person have **blonde hair**?

***Yes*: 5 candidates**, *No:* 19 candidates

Example: Guess Who

Question: Does the person have **blonde hair**?

Yes: 5 candidates, ***No:* 19 candidates**

Example: Guess Who

Question: Does the person have **short hair**?

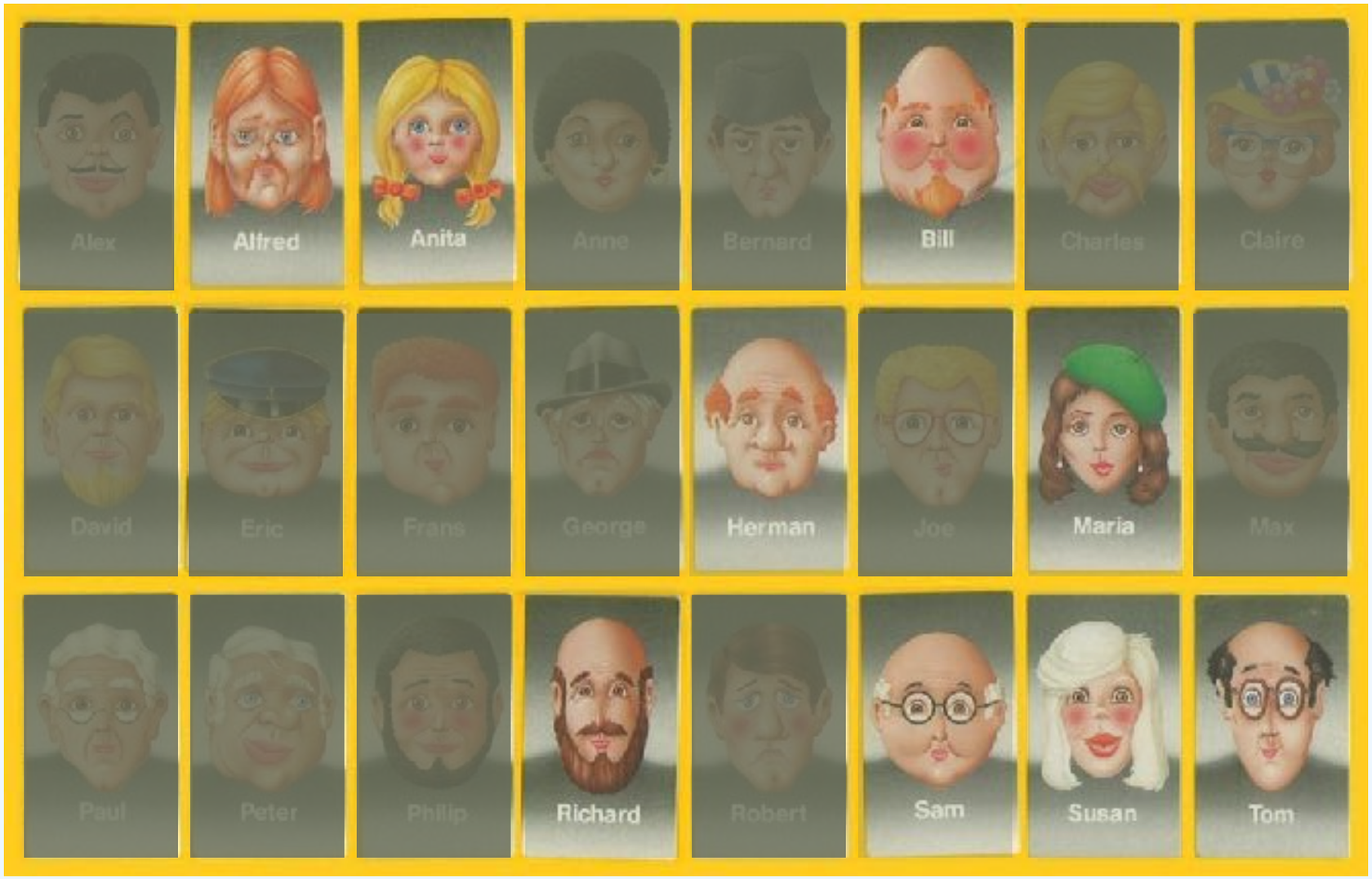
***Yes*: 15 candidates**, *No:* 9 candidates

Example: Guess Who

Question: Does the person have **blonde hair**?

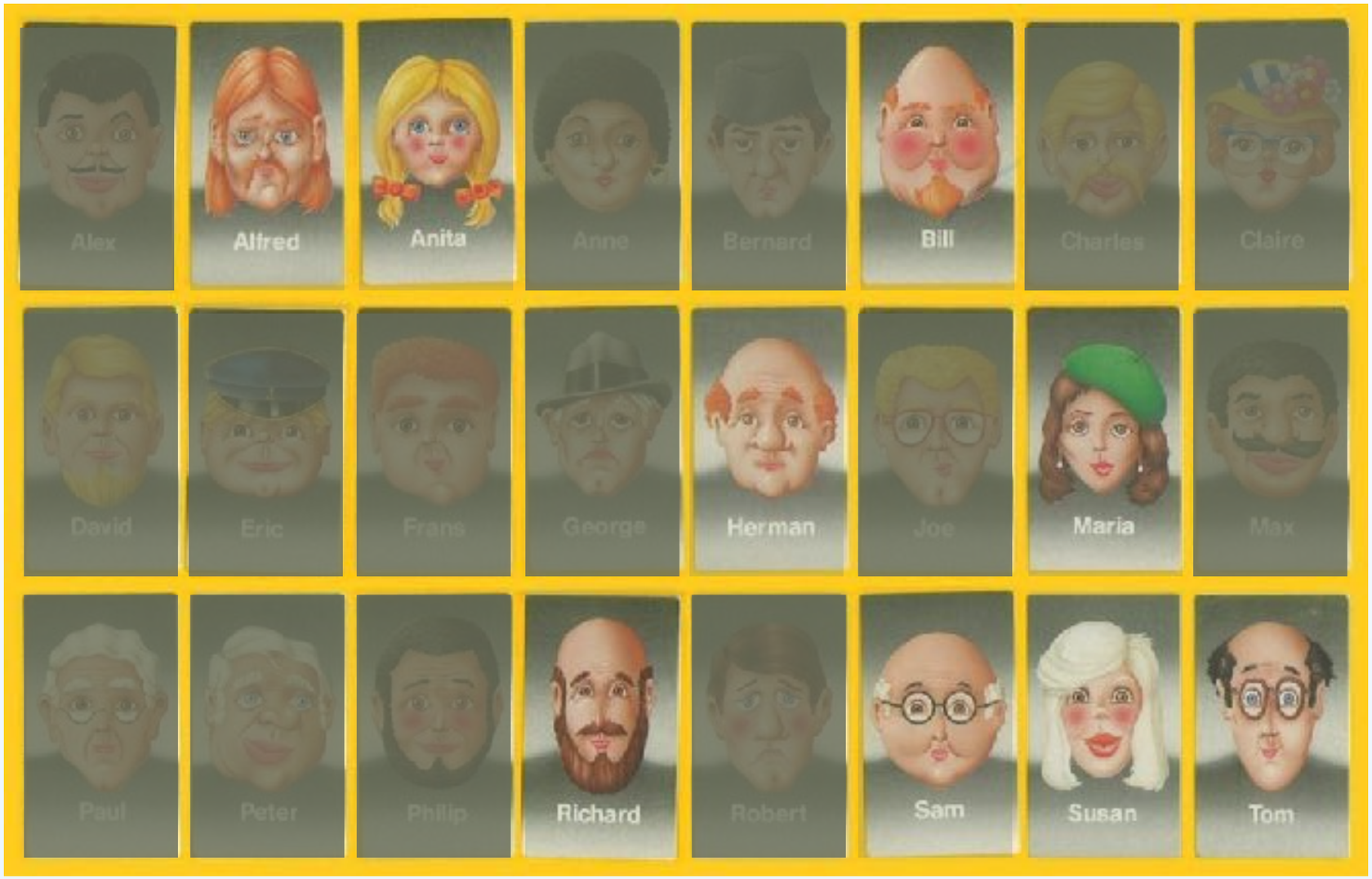
Yes: 15 candidates, ***No:* 9 candidates**

Example: Guess Who



What if answers are only 80% reliable?

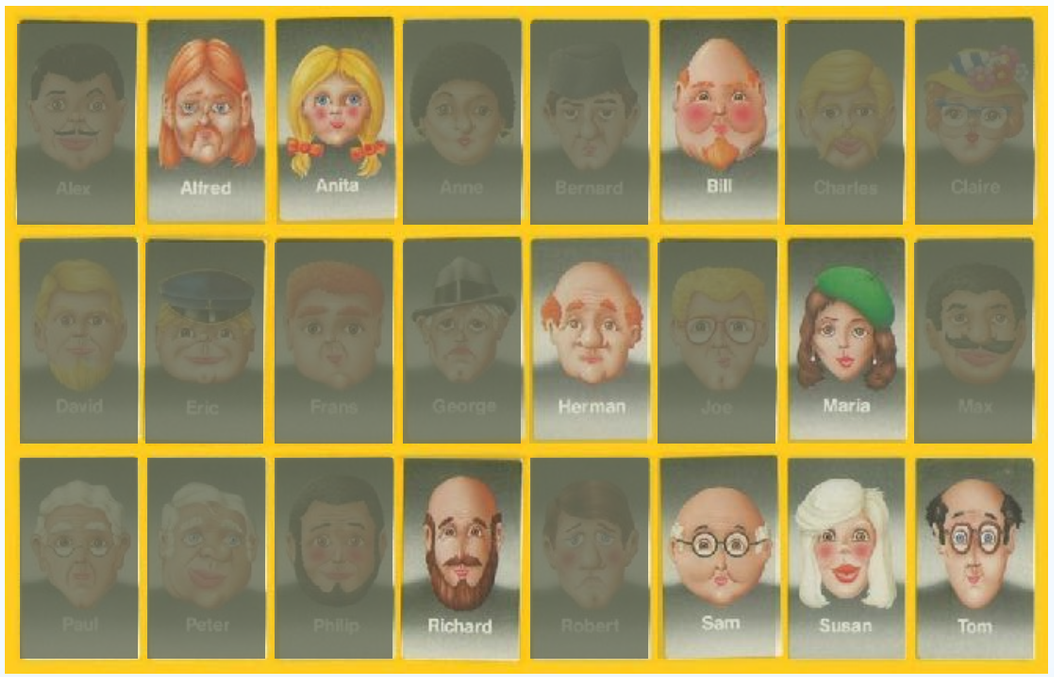
Example: Guess Who

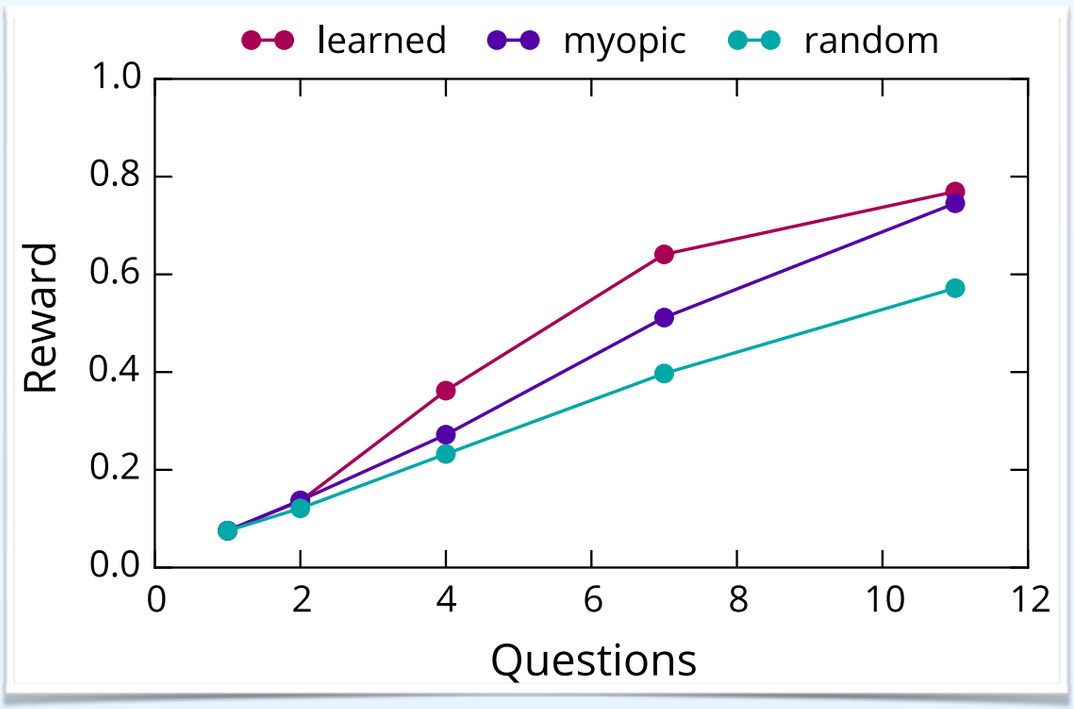


What if answers are only 80% reliable?

What question ***is most likely to inform*** my current belief?

Guess Who (noisy)





Heuristic: Value of Information

- Difference between expected return for *current* best guess that for *new* best guess.
- Make myopic approximation (assume question is last in game)

Policy: Discounted Utility

$$U_q = \gamma^{c_q} (A \cdot b)_q$$

U_q	Utility of question q
γ	Heuristic discount factor
c_q	Question count
A	Weight matrix
b	Belief vector