Black-Box Policy Search with Probabilistic Programs

Jan-Willem van de Meent **David Tolpin** **Brooks Paige** **Frank Wood**
Department of Engineering Science
University of Oxford
{jwvdm,dtolpin,brooks,fwood}@robots.ox.ac.uk

Abstract

In this work, we explore how probabilistic programs can be used to represent policies in sequential decision problems. In this formulation, a probabilistic program is a black-box stochastic simulator for both the problem domain and the agent. We relate classic policy gradient techniques to recently introduced black-box variational methods which generalize to probabilistic program inference. We present case studies in the Canadian traveler problem, Rock Sample, and a benchmark for optimal diagnosis inspired by Guess Who. Each study illustrates how programs can efficiently represent policies using moderate numbers of parameters.

1 Introduction

In planning under uncertainty the objective is to find an optimal policy that select actions, given current available information, so as to maximize a notion of expected reward. In most interesting cases the optimal policy cannot be found exactly, and approximation schemes are used to discover the policy, either represented explicitly or as an implicit property of the planning algorithm. Approximation schemes include value/policy iteration, Q-learning, methods based on heuristic search [Bonet and Geffner, 2001], Monte Carlo sampling such as Monte Carlo Tree Search [Kocsis and Szepesvári, 2006], and policy search methods. Policy search methods are a broad class of techniques that iteratively optimize a parameterized policy using simulations of actions and their outcomes (see [Deisenroth et al., 2011] for an extensive review).

Probabilistic programming systems [Milch et al., 2007, Goodman et al., 2008, Minka et al., 2010, Mansinghka et al., 2014, Wood et al., 2014, Gordon et al., 2014] represent generative models as programs in a language that has specialized syntax for definition and conditioning of random variables. A backend provides one or more general-purpose inference methods for any program in this language, resulting in an abstraction boundary between model and inference algorithm design. Models specified as programs are often concise, modular, and easy to modify or extend. This allows definition of structured priors that are specifically tailored to an application domain in a manner that is efficient in terms of the dimensionality of its latent variables, albeit at the expense of performing inference with generic methods that may not take advantage of model-specific optimizations.

In this work we present a first exploration of the use of probabilistic programming in policy search. We generalize black-box variational inference (BBVI), a recently introduced technique for approximation of the Bayesian posterior in a broad class of models [Ranganath et al., 2014], to inference in probabilistic programs. We connect this method to related classic gradient-based policy search methods such as REINFORCE [Williams, 1992], and demonstrate how policy search can be implemented as gradient optimization in exponential-reward weighted probabilistic programs. We present case studies in the Canadian traveler problem, the rock sample domain, and introduce a setting inspired by Guess Who [Coster and Coster, 1979] as a benchmark for optimal diagnosis problems. In each of these settings, which are characterized by a search space that grows as a double exponential of the search horizon, we show how probabilistic programming may be leveraged to efficiently represent the structure of the problem domain.

2 Background

2.1 Sequential Decision Problems

In sequential decision problems, an agent draws an action u_t from a policy distribution $\pi(u_t \,|\, x_t)$, which may be deterministic, conditioned on a context x_t at time t. After each action the agent observes a new context x_{t+1} drawn from a distribution $p(x_{t+1} \,|\, u_t, x_t)$. We here consider the finite horizon case, where the agent performs a fixed number of actions T. This results in a sequence $\tau = (x_0, u_0, x_1, u_1, x_2, \ldots, u_{T-1}, x_T)$, which is known as a trajectory, or rollout. Each trajectory gets a reward $R(\tau)$. Policy search methods maximize the expected reward $J_\theta = \mathbb{E}_{p_\theta}[R(\tau)]$ for a family of stochastic policies π_θ with parameters θ

$$J_\theta = \int R(\tau) p_\theta(\tau) \, d\tau, \qquad p_\theta(\tau) := p(x_0) \prod_{t=0}^{T-1} \pi(u_t \,|\, x_t, \theta) p(x_{t+1} \,|\, u_t, x_t). \tag{1}$$

We here define the problem in terms of the hyperparameters λ of a distribution $p(\tau, \theta \,|\, \lambda)$ that places a prior $p(\theta \,|\, \lambda)$ on the policy parameters, which is known as upper-level policy search

$$J_\lambda = \int R(\tau) p_\lambda(\tau, \theta) \, d\tau \, d\theta, \qquad p_\lambda(\tau, \theta) := p_\lambda(\theta) p(\tau \,|\, \theta). \tag{2}$$

In a Markov decision process (MDP), the context $x_t = s_t$ is a state $s_t \in S$ from a finite set. Each action $u_t = a_t$ likewise is an element of a finite set A. In a partially observable Markov decision process (POMDP), the current state s_t is not known, but the agent observes a value $o_t \in O$ from a finite set, which is drawn from a distribution $p(o_t \,|\, s_t, a_{t-1})$. In this setting, the context $x_t = (a_0, o_1, \ldots, a_{t-1}, o_t)$ is referred to as the information state, which is often represented in using a belief state $b_t(s) := p(s_t = s \,|\, a_0, \ldots, o_t)$.

2.2 Black-box Variational Inference

Variational Bayesian methods [Wainwright and Jordan, 2008] approximate an intractable posterior with a more tractable family of distributions. For purposes of exposition we consider the case of a posterior $p(z, \theta \,|\, y)$, in which y is a set of observations, θ is a set of model parameters, and z is a set of latent variables whose cardinality $|z|$ scales with $|y|$. We will write $p(z, \theta \,|\, y) = \gamma(z, \theta)/Z$ with

$$\gamma(z, \theta) = p(y \,|\, z, \theta) p(z, \theta), \qquad Z = \int \gamma(z, \theta) \, dz \, d\theta = \mathbb{E}_p[p(y \,|\, z, \theta)]. \tag{3}$$

Variational methods approximate the posterior using a parametric family of distributions q_λ by maximizing a lower bound on $\log Z$ with respect to hyperparameters λ

$$\mathcal{L}_\lambda = \mathbb{E}_{q_\lambda}[\log \gamma(z, \theta) - \log q_\lambda(z, \theta)] = \log Z - D_{\mathrm{KL}}(q_\lambda(z) \,||\, \gamma(z)/Z) > \log Z. \tag{4}$$

This objective may be optimized with stochastic gradient ascent

$$\lambda_{k+1} = \lambda_k + \rho_k \nabla_\lambda \mathcal{L}_\lambda, \qquad \nabla_\lambda \mathcal{L}_\lambda = \mathbb{E}_{q_\lambda(z)}[\nabla_\lambda \log q_\lambda(z)(\log \gamma(z, \theta) - \log q_\lambda(z, \theta))], \tag{5}$$

using a sequence of step sizes that satisfies the conditions $\sum_{k=1}^{\infty} \rho_k = \infty$ and $\sum_{k=1}^{\infty} \rho_k^2 < \infty$. The calculation of the gradient $\nabla_\lambda \mathcal{L}_\lambda$ requires an integral over q_λ. For certain models, specifically those where the likelihood and prior are in the conjugate exponential family [Hoffman et al., 2013], this integral can be performed analytically.

BBVI targets a much broader class of models by sampling $z^{[n]}, \theta^{[n]} \sim q_\lambda$ and replacing the gradient for each component i with a sample-based estimate [Ranganath et al., 2014]

$$\hat{\nabla}_{\lambda_i} \mathcal{L}_\lambda = \sum_{n=1}^{N} \nabla_{\lambda_i} \log q_\lambda(z^{[n]}, \theta^{[n]})(\log w^{[n]} - \hat{b}_i), \qquad w^{[n]} = \gamma(z^{[n]}, \theta^{[n]})/q_\lambda(z^{[n]}, \theta^{[n]}), \tag{6}$$

in which $\hat{b}_i$ is a control variate that reduces the variance of the estimator

$$\hat{b}_i = \sum_{n=1}^{N} (\nabla_{\lambda_i} \log q_\lambda(z^{[n]}, \theta^{[n]}))^2 w^{[n]} \Big/ \sum_{n=1}^{N} (\nabla_{\lambda_i} \log q_\lambda(z^{[n]}, \theta^{[n]}))^2. \tag{7}$$

2.3 Policy Search as Bayesian Inference

Upper-level policy search can be reinterpreted as Bayesian learning in graphical models. To illustrate this, we define an unnormalized density $\gamma_{\lambda_0}(\tau, \theta)$, analogous to the one defined in Equation 3, in which the exponent of the reward $\exp(\beta R(\tau))$ takes the role of the likelihood

$$\gamma_{\lambda_0}(\tau, \theta) = p_{\lambda_0}(\tau, \theta) \exp(\beta R(\tau)), \qquad Z_{\lambda_0} = \int \gamma_{\lambda_0}(\tau, \theta)\, d\tau\, d\theta = \mathbb{E}_{p_{\lambda_0}}[\exp(\beta R(\tau))]. \tag{8}$$

In this type of formulation [Rawlik et al., 2012, Neumann, 2011, Levine and Koltun, 2013], the normalization constant Z_{λ_0} is the expected value of the exponentiated reward $\exp(\beta R(\tau))$, which is known as the desirability in the context of optimal control [Kappen, 2005, Todorov, 2009b]. The constant $\beta > 0$ plays the role of an 'inverse temperature' that controls how strongly the density penalizes sub-optimal actions.

In a Bayesian context, maximization of the normalizing constant Z_{λ_0} with respect to λ_0 is known as empirical Bayes (EB) estimation. Posterior inference, or variational Bayes (VB) estimation, maximizes the analogue to the lower bound $\mathcal{L}_{\lambda,\lambda_0} \leq \log Z_{\lambda_0}$ from Equation 4. This bound can be rewritten as

$$\mathcal{L}_{\lambda,\lambda_0} = \mathbb{E}_{q_\lambda}[\log \gamma_{\lambda_0}(\tau, \theta) - \log q_\lambda(\tau, \theta)] = \mathbb{E}_{q_\lambda}[\beta R(\tau)] - D_{\mathrm{KL}}(q_\lambda(\tau, \theta) \,||\, p_{\lambda_0}(\tau, \theta)). \tag{9}$$

In other words, the lower bound in sequential decision problems represents a trade-off between the expected reward under the approximate posterior q_λ and the information-theoretic deviation between prior p_{λ_0} and approximate posterior q_λ, in which β controls relative importance of rewards.

The normalized density $\gamma_{\lambda_0}/Z_{\lambda_0}$ is sometimes interpreted as a posterior $p(\tau, \theta \mid y)$ conditioned on a single pseudo-observable with likelihood $p(y \mid \tau) := \exp(\beta R(\tau))$. This pseudo-observable y can be defined as Bernoulli-distributed when $R(\tau) \leq 0$ for all τ, which ensures that $0 \leq \exp(\beta R(\tau)) \leq 1$. In this case $y = 1$ can be seen as a binary indicator of 'optimality.' In the case where $R(\tau)$ can be positive, we may either define $\gamma_{\lambda_0}(\tau, \theta) = p_{\lambda_0}(\tau, \theta) \exp(\beta(R(\tau) - R^*))$, where $R^* = \sup_\tau R(\tau)$ is the finite-horizon supremum of the reward, or interpret $\exp(\beta R(\tau))$ as a factor in a factor graph.

While the interpretation of γ_{λ_0}/Z as a posterior $p(\tau, \theta \mid y)$ is somewhat loose, it is a helpful metaphor when differentiating between EB and VB estimation in the context of policy search. We can intuitively think of EB as learning the prior $p_{\lambda_0}(\tau, \theta)$ that maximizes the marginal probability $p(y \mid \lambda_0)$ of generating a trajectory τ that is 'optimal'. In VB the prior $p_{\lambda_0}(\theta)$ reflects our initial belief about the parameters of the policy $\pi(u \mid \theta)$, which in turn implies a belief about the utility of actions. The marginal of the approximate posterior $q_\lambda(\theta) = \int q_\lambda(\tau, \theta) d\tau$ represents the updated belief about policy parameters, in light of the fact that one 'optimal' trajectory τ was generated by the policy.

2.4 Probabilistic Programs

A probabilistic programming language provides syntax to instantiate random variables, as well as syntax to impose conditions on these random variables. The goal of inference in a probabilistic program is to characterize the distribution on its random variables, subject to the imposed conditions. When conditions define hard constraints (such as $a = 10$, or $a > 5$, for some variable a), execution of a probabilistic program can be interpreted as rejection sampling. Expectation values can, in principle, be calculated by repeatedly running the program and excluding executions that violate one or more conditions. In practice, the language design often implicitly or explicitly guarantees that conditions can be satisfied with a computable non-zero probability in any execution [Goodman et al., 2008, Wood et al., 2014, Mansinghka et al., 2014]. Probabilistic programs in such languages can be seen as importance samplers; repeated execution yields samples from a prior, which are assigned an importance weight according to the probability that its conditions are satisfied.

From a modeling point of view, a probabilistic program $\mathcal{P}$ defines a probability distribution. In a language that provides if-statements, recursion, higher-order functions, and primitive operations such as matrix inversion, this distribution can have an almost arbitrary structure. This means that from an inference point of view, a program $\mathcal{P}$ is a sequence of opaque deterministic operations that is interrupted by (1) *sample* statements, which require generation of a value for a random variable and (2) *conditioning* statements, which change the importance weight of the execution history. In summary, probabilistic programs can be used to define black-box importance samplers that can incorporate arbitrary simulation steps. We will make this notion precise in the next section.

3 Black-box Policy Search

In sequential decision problems, it is natural to think of the distribution $p(x_{t+1} \mid u_t, x_t)$ as a simulator of the state of the world, which generates new contextual information x_{t+1} given the current context x_t and an action u_t. Conversely the policy $\pi(u_t \mid x_t, \theta)$ is a simulator of an agent that decides on an action u_t given the current contextual information x_t and a set of parameters θ that in some way represents generalizable knowledge accumulated from past experiences.

For sufficiently simple problems, both the agent and the world simulator can be adequately described as distributions with a known analytical form, or graphical models. Here we are interested in using probabilistic programs as simulators of both the world and the agent. The trade-off made in this approach is that we can incorporate more detailed assumptions about the structure of the problem into our simulators, at the expense of having to treat them as black boxes from the perspective of the inference algorithm.

To perform policy search in probabilistic programs, we propose an EB variant of BBVI. This method is closely related to the one proposed by [Wingate and Weber, 2013]. We make a single assumption about the structure of the model, which is that the prior on policy parameters factorizes $p(\theta \mid \lambda_0) = \prod_i p(\theta_i \mid \lambda_{0,i})$ into a product of terms $p(\theta_k \mid \lambda_{0,k})$ for which we can calculate a gradient. If we now use a variational distribution $q(\tau, \theta \mid \lambda) = p(\tau \mid \theta) q(\theta \mid \lambda)$ that has the same form as the prior $p(\tau, \theta \mid \lambda_0)$, then at $\lambda = \lambda_0$, the gradient from Equation 5 to simplifies to[1]

$$\nabla_\lambda \mathcal{L}_{\lambda,\lambda_0}\Big|_{\lambda=\lambda_0} = \mathbb{E}_{q_\lambda}\left[\nabla_\lambda \log q_\lambda(\tau,\theta)\left(\beta R(\tau) + \log p_{\lambda_0}(\tau,\theta) - \log q_\lambda(\tau,\theta)\right)\right]\Big|_{\lambda=\lambda_0}, \tag{10}$$

$$= \beta \mathbb{E}_{q_\lambda}\left[\nabla_\lambda \log q_\lambda(\tau,\theta) R(\tau)\right]\Big|_{\lambda=\lambda_0} = \beta\, \nabla_\lambda J_\lambda\Big|_{\lambda=\lambda_0}. \tag{11}$$

In other words, at $\lambda = \lambda_0$ the gradient of the expected reward $\nabla_\lambda J_\lambda$ is proportional to the gradient $\nabla_\lambda \mathcal{L}_{\lambda,\lambda_0}$ of the lower bound on the log desireability. We note that the proportionality constant β can be absorbed into the step size without loss of generality. We therefore from here on assume $\beta = 1$.

The implication of the above identity is that we may perform upper-level policy search by gradient ascent with updates $\lambda_{k+1} = \lambda_k + \rho_k \hat{\nabla}_\lambda \mathcal{L}_{\lambda,\lambda_k}$. The difference between BBVI, which performs VB estimation, and this method, which performs EB estimation, is that the density γ_{λ_k} is conditioned on the previous hyperparameters λ_k, not the prior parameters λ_0. Intuitively, this means that after each gradient step, the parameters λ_k are used to update the prior for the next iteration. The gradient $\hat{\nabla}_\lambda \mathcal{L}_{\lambda,\lambda_k}$ is the equivalent of the sample-based estimator in Equation 6, which simplifies to

$$\hat{\nabla}_{\lambda_i} \mathcal{L}_{\lambda,\lambda_k}\Big|_{\lambda=\lambda_k} = \beta \sum_{n=1}^{N} \nabla_{\lambda_i} \log q_\lambda(\tau^{[n]}, \theta^{[n]})(R(\tau^{[n]}) - \hat{b}_i)\Big|_{\lambda=\lambda_k}. \tag{12}$$

Here $\hat{b}_i$ is a control variate equivalent to the one defined in Equation 7, where the trajectory τ takes the place of the latent variables z.

As a corollary, we may perform maximum likelihood (ML) estimation in a normal policy search setting by using an analogous estimator of the gradient relative to the parameters θ,

$$\hat{\nabla}_\theta \mathcal{L}_{\theta,\theta_k}\Big|_{\theta=\theta_k} = \beta \sum_{n=1}^{N} \nabla_{\theta_i} \log q_\theta(\tau^{[n]})(R(\tau^{[n]}) - \hat{b}_i)\Big|_{\theta=\theta_k}. \tag{13}$$

This is nothing but the likelihood-ratio policy gradient estimator that forms the basis for classic policy search methods such as REINFORCE [Williams, 1992], G(PO)MDP [Baxter and Bartlett, 1999], and the Policy Gradient Theorem algorithm [Sutton et al., 1999].

The difference between the EB and ML gradient estimators is that the first calculates the gradient relative to hyperparameters λ, whereas the other calculates the gradient relative to the parameters θ. Because this difference relates only to the assumed model structure, EB estimation is sometimes referred to as Type II maximum likelihood. We will for this reason refer to gradient optimization with the above sample-based estimators as black-box expectation maximization (BBEM).

[1]We make use of the standard identity $\nabla_\lambda q_\lambda(\tau,\theta) = q_\lambda(\tau,\theta)\nabla_\lambda \log q_\lambda(\tau,\theta)$ in the last equality.

In the context of probabilistic programs, BBEM is a comparatively light-weight technique. It does not require a full automatic differentiation implementation, such as provided by STAN [Stan Development Team, 2014], but instead relies only on derivatives of a limited number of distribution types provided by the language. To show how BBEM can be implemented in probabilistic programs, we abstract away from the details of a specific language syntax, and use an operational definition of a program $\mathcal{P}$. Specifically, we define $\mathcal{P}$ as a stateful deterministic computation in which

- Initially $\mathcal{P}$ is called with no arguments $\mathcal{P}()$
- Each call to $\mathcal{P}$ returns one of three types of tuples:
  1. $(\texttt{sample}, \alpha, f, \phi)$: Here α is a unique address of a random variable z_α distributed according to f_α with parameters ϕ_α. The backend samples $x_\alpha \sim f_\alpha(\cdot \,|\, \phi_\beta)$. Execution continues by calling $\mathcal{P}(x_\alpha)$.
  2. $(\texttt{factor}, \gamma, l)$: Here γ is a unique address for a factor with log probability l_γ and importance weight $w_\gamma = \exp(l_\gamma)$. Execution continues by calling $\mathcal{P}()$.
  3. $(\texttt{return}, v)$: Execution completes, returning a value v.

Because each call to $\mathcal{P}$ is deterministic, an execution history is fully characterized by the values that are associated with its random variables. However the set of random variables that is instantiated may vary from execution to execution. We write A, Γ for the set of addresses of each type visited in a given execution. The program $\mathcal{P}$ now defines an unnormalized density $\gamma_{\mathcal{P}_\lambda}$ on x of the form

$$\gamma_{\mathcal{P}}(x) := p_{\mathcal{P}}(x) \prod_{\gamma \in \Gamma} \exp(l_\gamma), \qquad p_{\mathcal{P}}(x) := \prod_{\alpha \in A} f_\alpha(x_\alpha \,|\, \phi_\alpha)\,. \tag{14}$$

Implicit in this notation is the fact that the distribution types f_α and their parameters ϕ_α, are return values from calls to $\mathcal{P}$. While these are in principle fully determined by x, we assume they are opaque to the inference algorithm, in the sense that no analysis is performed to characterize the conditional dependence of each ϕ_α on other random variables in the program.

In order to implement BBEM, we need to extend the above definition of $\mathcal{P}$ to differentiate between the random variables x that are to be considered parameters θ, and the random variables that take the role of latent variables z. This can be done in any number of ways. For simplicity we here assume that the programmer explicitly declares the role of each variable. This allows us to express $\mathcal{P}$ as density on latent variables z_α with addresses $\alpha \in A$, parameters θ_β with addresses $\beta \in B$, and factors with addresses $\gamma \in \Gamma$

$$\gamma_{\mathcal{P}}(z, \theta) := p_{\mathcal{P}}(z, \theta) \prod_{\gamma \in \Gamma} \exp(l_\gamma), \qquad p_{\mathcal{P}}(z, \theta) := \prod_{\alpha \in A} f_\alpha(z_\alpha \,|\, \phi_\alpha) \prod_{\beta \in B} f_\beta(\theta_\beta \,|\, \eta_\beta)\,. \tag{15}$$

We now define a variational approximation $\mathcal{Q}_\lambda$ of a program $\mathcal{P}$ with parameters λ that will be learned by the inference backed. A variational program $\mathcal{Q}_\lambda$ executes in the same manner as $\mathcal{P}$, except when it encounters a variable that is labeled as variational. In this case it returns the tuple

4. $(\texttt{learn}, \beta, f, \eta)$: The address β identifies a random variable θ_β in the model, distributed according to a distribution f_β with parameters η_β. The backend samples $\theta_\beta \sim f(\cdot \,|\, \lambda_\beta)$ conditioned on a learned variational parameter λ_β which incurs an importance weight $w_\beta = f(\theta_\beta \,|\, \eta_\beta)/f(\theta_\beta \,|\, \lambda_\beta)$. Execution continues by calling $\mathcal{Q}_\lambda(\theta_\beta)$.

Repeated execution of $\mathcal{Q}_\lambda$ results in a sequence of weighted samples $(w^{[n]}, \theta^{[n]}, z^{[n]}, v^{[n]})$, whose importance weight $w^{[n]}$ is defined as

$$w^{[n]} := \frac{\gamma_{\mathcal{P}}(z^{[n]}, \theta^{[n]})}{p_{\mathcal{Q}_\lambda}(z^{[n]}, \theta^{[n]})}, \qquad p_{\mathcal{Q}_\lambda}(z, \theta) := \prod_{\alpha \in A} f_\alpha(z_\alpha \,|\, \phi_\alpha) \prod_{\beta \in B} f_\beta(\theta_\beta \,|\, \lambda_\beta). \tag{16}$$

With this notation in place, it is clear that we can define a lower bound $\mathcal{L}_{\mathcal{Q}_\lambda, \mathcal{Q}_{\lambda_k}}$ analogous to that of Equation 10, and a gradient estimator analogous to that of Equation 13, in which the latent variables z take the role of the trajectory variables τ. In summary, we can describe a sequential decision problem as a probabilistic program $\mathcal{P}$ in which the log probabilities l_γ are interpreted as rewards, variational parameters θ_β define the policy and all other latent variables z_α are trajectory variables. BBEM inference can then be used to learn the hyperparameters λ that maximize the expected reward.

We note that this definition does not explicitly assume conditional independence structure between the terms $f_\beta(\theta_\beta \,|\, \eta_\beta)$. In the following we do assume this independence holds in order to ensure that the approximating program $\mathcal{Q}_\lambda$ has the same graphical structure as the prior $\mathcal{P}$. Intuitively this means that there must be some representation $\mathcal{P}_\eta$ in which the all η_β have the same values for any execution. One way to enforce this is to pass η in the initial call $\mathcal{P}(\eta)$, although we do not formalize such a requirement here.

4 Related Work

4.1 Planning and Control as Inference

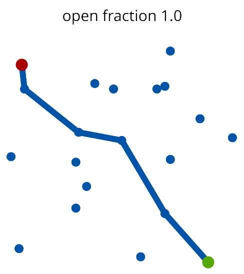
Our formulation of policy search as expectation maximization in probabilistic programs fits into a long history of approaches that frame planning and control problems as inference. Path integral methods [Kappen, 2005, 2007, Broek et al., 2008, Todorov, 2009a,b, Kappen et al., 2012] express the expected cost-to-go as a KL-divergence between a posterior, controlled, distribution on $x_{1:T}$ and a distribution defined as the product of the prior, uncontrolled, dynamics and the exponent of the reward. Minimization of this KL divergence is then equivalent to minimization of the expected cost-to-go. In the context of these types of approaches, minimization of the expected exponentially weighted cost-to-go is known as risk-sensitive optimal control [van den Broek et al., 2010]. Path integral approaches have been applied to reinforcement learning problems in a number of studies [Theodorou et al., 2007, Theodorou and Todorov, 2012, Azar and Kappen, 2012].

Related approaches [Rawlik et al., 2010, 2012] minimize the KL divergence between the distribution on trajectories τ induced by a stochastic policy π relative to that induced by a an exponentially weighted posterior distribution with policy π_0. This is equivalent to a optimal control problem in which the reward is discounted by the log probability of the actions under π_0. Iterative solution, in which each π_i is obtained by KL minimization relative to the posterior induced by π_{i-1}, converges to the optimal policy in this setting, which is consistent with our observation that expectation maximization is equivalent to maximization of the expected reward.

The use of variational inference in reinforcement learning problems has also been explored in a number of ways. When the reward is non-negative, variational Bayes and expectation propagation can be used to approximate a posterior in a (non-exponentiated) reward weighted model [Furmston and Barber, 2010]. The exponentiated reward case has similarly been considered [Neumann, 2011]. In this setting, it has been observed that minimization of the M-projection (i.e. the KL divergence between posterior and variational distribution, as used in expectation propagation) instead of the I-projection (i.e. the KL divergence between the variational distribution distribution and the posterior) leads to an objective that is equivalent to that used in Monte-Carlo EM approaches [Kober and Peters, 2011, Vlassis et al., 2009]. As noted, the variant of BBVI used here is closely related to the one proposed by [Wingate and Weber, 2013] for probabilistic programs.

4.2 Policies as Programs

In the context of probabilistic programming, previous work has applied probabilistic programming to both MDP [Andre and Russell, 2001] and POMDP [Srivastava et al., 2014] settings. In the former, a Lisp dialect is extended with three macros: `choice`, `call` and `action`. This results in a language, ALisp, tailored to the formulation of MDP problems, which may be solved using an adaptation of the MAXQ algorithm [Dietterich, 2000]. The latter considers open world POMDP settings in the language BLOG [Milch et al., 2007], resulting in a variant DTBLOG that adds language decision variables to represent actions and observability, and develops an open universe variant of point-based value iteration [Pineau et al., 2003] known as OUPBVI for solution. Our work differs somewhat from both of these approaches, in that it emphasizes the expressivity of programs for formulation of sequential decision problems in a generic inference setting, rather than solution with problem-specific algorithms in a specialized language. In this sense previous work on compositional policy priors [Wingate et al., 2013], which considers the use of Pitman-Yor adaptor grammars for specification of policies, is related to this work in spirit, even though it is not explicitly formulated in the context of probabilistic programming languages.



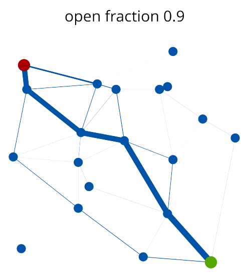
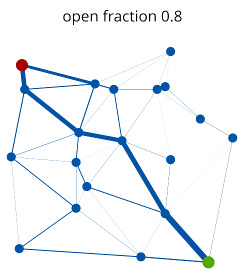
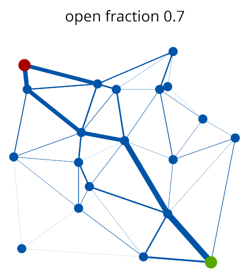




Figure 1: Learned policies for the Canadian traveler problem. Line widths indicate the frequency at which the policy travels each edge, averaged over random combinations of open and closed edges.

5 Case Studies

We demonstrate the proposed policy learning method on three problem domains: (1) the Canadian Traveller Problem, (2) a modified version of the RockSample POMDP, and (3) an optimal diagnosis benchmark inspired by the classic children's game Guess Who. Each of these domains can be formulated as a POMDP. This means that there is some form of unobserved state in the problem instance, and the agent must choose actions based on contextual information x_t that can be described in terms of an information state $x_t = (u_0, o_1, \ldots, u_{t-1}, o_t)$. Even for discrete problems, the cardinality of the set of possible information states x_t grows exponentially with the horizon T.

The aim of these studies is to explore how probabilistic programs can be used to define policies tailored to the structure of each domain. Fundamentally, some information must be discarded when making a decision. Program policies encode our intuition about what information is most relevant in a given context. As such, these studies are not intended to achieve results that are competitive with current state-of-the-art specialized techniques for POMDPs (see Shani et al. [2013] for a recent overview). Rather, we consider probabilistic programs as a concise algorithmic representation of domain-specific probabilistic mappings from information states to actions, in order to describe the search space over policies in terms of a moderate yet not unwieldly number of parameters.

5.1 Evaluation Setup

We use the same experimental setup in each of the three domains. A trial begins with a learning phase, in which BBEM is used to learn the policy hyperparameters, followed by a number of testing episodes in which the agent chooses actions according to a fixed learned policy. At each gradient update step, we use 1000 samples to calculate a gradient estimate. Each testing phase consists of 1000 episodes. All shown results are based on test-phase simulations.

Stochastic gradient methods can be sensitive to the learning rate parameters. Results reported here use a RMSProp style rescaling of the gradient [Hinton et al.], which normalizes the gradient by a discounted rolling decaying average of its magnitude with decay factor 0.9. We use a step size schedule $\rho_k = \rho_0/(\tau + k)^\kappa$ as reported in [Hoffman et al., 2013], with $\tau = 1$, $\kappa = 0.5$ in all experiments. We use a relatively conservative base learning rate $\rho_0 = 0.1$ in all reported experiments. For independent trials performed across a range $1, 2, 5, 10, \ldots, 1000$ of total gradient steps, consistent convergence was observed in all runs using over 100 gradient steps.

5.2 Canadian Traveller Problem

In the Canadian Traveller Problem [Papadimitriou and Yannakakis, 1991], an undirected graph $G = (V, E)$ is given, along with the cost w_e of traversing every edge $e \in E$, and the probability p_e that the edge is open. The agent must traverse the graph from the initial node s to the goal node t at the lowest possible cost. The agent does not know the state of an edge until it reaches one of the edge's vertices. The problem is NP-hard [Fried et al., 2013], and heuristic online and offline approaches [Eyerich et al., 2010] are used to solve problem instances.

Here we learn a policy based on the depth-first search (DFS) — the agent traverses the graph in the depth-first order until the goal node is reached (only connected instances are considered). Depth-first

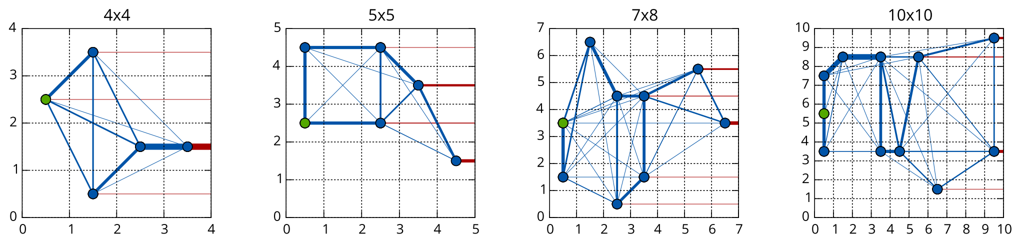


Figure 2: Learned policies for the Rock Sample domain. Edge weights indicate the frequency at which the agent moves between each pair of rocks. Starting points are in green, exit paths in red.

search is an indeterministic algorithm in the sense that the order in which children of every node are considered can be chosen arbitrarily; however, the performance of DFS on a CTP instance is likely to vary depending on the traversal order. We learn a prior on the priority of each child node in every node for a given problem instance, and then evaluate the policy on samples from a distribution of random instantiations of edge states (open or blocked).

Whereas a policy that performs DFS would be difficult if not impossible to describe as a graphical model, its program representation is a straightforward recursive procedure. The number of parameters in this problem, which is two times the number of edges, is manageable and can easily be learned via gradient ascent. The results in Figure 1 show that the learned policy behaves in a reasonable manner. When edges are open with high probability, the policy takes the shortest path from the start node, marked in green, to the target node, marked in red. As the number of closed edges increases, the policy makes use of more alternate routes, whilst avoiding peripheral edges of the graph.

5.3 RockSample POMDP

In the RockSample POMDP [Smith and Simmons, 2004], a square field $N \times N$ with M rocks is given. The robot is initially located in the middle of the left edge of the square. Each of the rocks can be either good or bad; and robot must traverse the field and collect samples of good rocks. The robot can sense the quality of rocks remotely with accuracy decreasing with the distance to the rock. The objective is to cross the field fast, while still sampling as many good rocks as possible. The agent gets a reward of 10 for each good rock, as well as for reaching the right edge. It incurs a penalty -10 for each bad rock, as well as a penalty -1 for each move.

We learn parameters of a heuristic policy in which the robot moves along rocks in a left-to-right order. At each point the agent selects the closest rock and senses it. Based on the reading it then chooses to either move to the rock, or discard it and consider the next closest rock. When the agent gets to a rock, it only samples the rock if the rock is good. The parameters describe the prior over the probability of moving to a rock conditioned on the current location and the sensor reading.

The policy plots in Figure 2 show that this simple policy results in sensible movement preferences. In particular we point out that in the 5×5 instance, the agent always visits the top-left rock when traveling to the top-middle rock, since doing so incurs no additional cost. Similarly, the agent follows an almost deterministic trajectory along the left-most 5 rocks in the 10×10 instance, but does not always make the detour towards the lower rocks afterwards.

5.4 Guess Who

Guess Who is a classic game in which players pick a card depicting a face, belonging to a set that is known to both players. The players then take turns asking questions until they identify the card of the other player [Coster and Coster, 1979]. We here consider a single-player setting where an agent asks a pre-determined number of questions, but the responses are inaccurate with some probability. This is sometimes known as a measurement selection, or optimal diagnosis problem. We make use of a feature set based on the original game, consisting of 24 individuals, characterized by 11 binary attributes and two multi-class attributes, resulting in a total of 19 possible questions. We assume

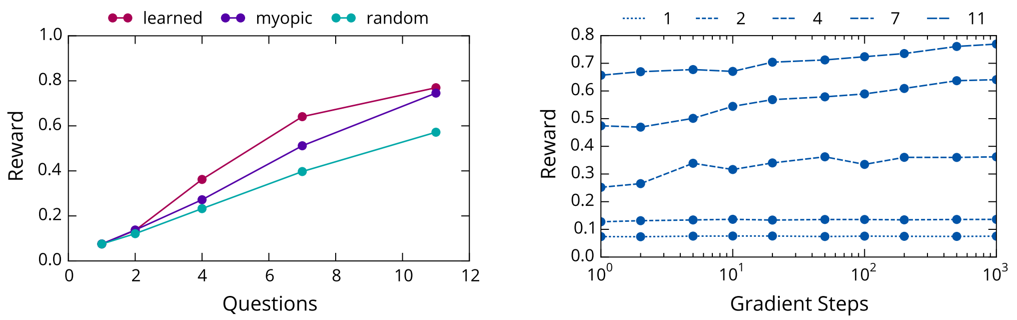


Figure 3: (left) Average reward in Guess Who as a function of number of questions. (right) Convergence of rewards as function number of gradient steps. Each dot marks an independent restart.

a response accuracy of 0.9. By design, the structure of the domain is such that there is no clear winning opening question. However the best question at any point is highly contextual.

We assume that the agent knows the reliability of the response and has an accurate representation of the posterior belief $b_t(s) = p(s \mid x_t)$ for each candidate s in given questions and responses. The agent selects randomly among the highest ranked candidates after the final question. We consider 3 policy variants, two of which are parameter-free baselines. In the first baseline, questions are asked uniformly at random. In the second, questions are asked according to a myopic estimate of the value of information [Hay et al., 2012], i.e. the change in expected reward relative to the current best candidates, which is myopically optimal in this setting. Finally, we consider a policy that empirically samples questions q according to a weight $v_q = \gamma^{c_q}(Ab)_q$, based on the current belief b, a weight matrix A, and a discount factor γ^{n_q} based on the number of times n_q a question was previously asked. Intuitively, this algorithm can be understood as learning a small set of α-vectors, one for each question, similar to those learned in point-based value iteration [Pineau et al., 2003]. The discounting effectively "shrinks" the belief-space volume associated with the α-vector of the current best question, allowing the agent to select the next-best question.

The results in Figure 3 show that the learned policy clearly outperforms both baselines, which is a surprising result given the complexity of the problem and the relatively simplistic form of this heuristic policy. While these results should not be expected to be in any way optimal, they are encouraging in that they illustate how probabilistic programming can be used to implement and test policies that rely on transformations of the belief or information state in a straightforward manner.

6 Discussion

In this paper we put forward the idea that probabilistic programs can be a productive medium for describing both a problem domain and the agent in sequential decision problems. Programs can often incorporate assumptions about the structure of a problem domain to represent the space of policies in a more targeted manner, using a much smaller number of variables than would be needed in a more general formulation. By combining probabilistic programming with black-box variational inference we obtain a generalized variant of well-established policy gradient techniques that allow us to define and learn policies with arbitrary levels of algorithmic sophistication in moderately high-dimensional parameter spaces. Fundamentally, policy programs represent some form of assumptions about what contextual information is most relevant to a decision, whereas the policy parameters represent domain knowledge that generalizes across episodes. This suggests future work to explore how latent variable models may be used to represent past experiences in a manner that can be related to the current information state.

References

David Andre and Stuart J Russell. State Abstraction for Programmable Reinforcement Learning Agents. (October), 2001.

Mohammad Gheshlaghi Azar and Bert Kappen. Dynamic Policy Programming. *Journal of Machine Learning Research*, 13:3207–3245, 2012. ISSN 15324435.

J Baxter and P L Bartlett. Direct gradient-based reinforcement learning: I. gradient estimation algorithms. Technical report, Comp. Sci. Lab., Australian National University, 1999.

Blai Bonet and Hector Geffner. Planning as heuristic search. *Artif. Intell.*, 129(1-2):5–33, 2001.

Bart Broek, Wim Wiegerinck, and Bert Kappen. Graphical model inference in optimal control of stochastic multi-agent systems. *Journal of Artificial Intelligence Research*, 32:95–122, 2008.

Theo Coster and Ora Coster. Guess Who? `http://theoradesign.com`, 1979.

Marc Peter Deisenroth, Gerhard Nuemann, and Jan Peters. A Survey on Policy Search for Robotics. *Foundations and Trends in Robotics*, 2(2011):1–142, 2011.

Thomas G Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomp osition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000. ISSN 10769757. doi: 10.1613/jair.639.

Patrick Eyerich, Thomas Keller, and Malte Helmert. High-quality policies for the Canadian traveler's problem. In *AAAI*, 2010.

Dror Fried, Solomon Eyal Shimony, Amit Benbassat, and Cenny Wenner. Complexity of Canadian traveler problem variants. *Theor. Comput. Sci.*, 487:1–16, 2013.

Thomas Furmston and D Barber. Variational methods for reinforcement learning. *International Conference on Machine Learning*, pages 241–248, 2010.

N. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence*, pages 220–229, 2008.

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *International Conference on Software Engineering (ICSE, FOSE track)*, 2014.

Nicholas Hay, Stuart Russell, David Tolpin, and Solomon Shimony. Selecting Computations: Theory and Applications. In *Uncertainty in Artificial Intelligence: Proceedings*. 2012. ISBN 978-0-9749039-8-9.

Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

Matthew D. Hoffman, David M. Blei, Chong Wang, and John Paisley. Stochastic variational inference. *Journal of Machine Learning Research*, 14:1303–1347, 2013.

H. J. Kappen. Path integrals and symmetry breaking for optimal control theory. 11011:21, 2005.

Hilbert J. Kappen. An introduction to stochastic control theory, path integrals and reinforcement learning. In *American Institute of Physics Conference Series*, volume 887, pages 149–181, 2007. ISBN 0735403902. doi: 10.1063/1.2709596.

Hilbert J. Kappen, Vicenç Gómez, and Manfred Opper. Optimal control as a graphical model inference problem. *Machine Learning*, 87(2):159–182, 2012. ISSN 08856125. doi: 10.1007/s10994-012-5278-7.

Jens Kober and Jan Peters. Policy search for motor primitives in robotics. *Machine Learning*, 84(1-2):171–203, 2011. ISSN 08856125. doi: 10.1007/s10994-010-5223-6.

L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, pages 282–293, 2006.

Sergey Levine and Vladlen Koltun. Guided Policy Search. In *International Conference on Machine Learning*, volume 28, pages 1–9, 2013.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv*, page 78, March 2014.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L Ong, and Andrey Kolobov. Blog: Probabilistic models with unknown objects. *Statistical relational learning*, page 373, 2007.

T Minka, J Winn, J Guiver, and D Knowles. Infer .NET 2.4, Microsoft Research Cambridge, 2010.

Gerhard Neumann. Variational Inference for Policy Search in Changing Situations. In *International Conference on Machine Learning*, 2011.

Christos H. Papadimitriou and Mihalis Yannakakis. Shortest paths without a map. *Theor. Comput. Sci.*, 84(1): 127–150, July 1991.

Joelle Pineau, Geoff Gordon, and Sebastian Thrun. Point-based value iteration: An anytime algorithm for POMDPs. In *International Joint Conference on Artificial Intelligence*, pages 1025–1030, 2003. ISBN 1045-0823.

Rajesh Ranganath, Sean Gerrish, and David M Blei. Black Box Variational Inference. In *Artificial Intelligence and Statistics*, 2014.

Konrad Rawlik, Sethu Vijayakumar, and Marc Toussaint. An Approximate Inference Approach to Temporal Optimization in Optimal Control. In *Neural Information Processing Systems*, pages 1–9, 2010. ISBN 9781617823800.

Konrad Rawlik, Marc Toussaint, and Sethu Vijayakumar. On Stochastic Optimal Control and Reinforcement Learning by Approximate Inference. *On Stochastic Optimal Control and Reinforcement Learning by Approximate Inference*, (2), 2012.

Guy Shani, Joelle Pineau, and Robert Kaplow. A survey of point-based POMDP solvers. *Autonomous Agents and Multi-Agent Systems*, 27(1):1–51, 2013.

Trey Smith and Reid Simmons. Heuristic search value iteration for pomdps. In *Uncertainty in Artificial Intelligence*, pages 520–527, Arlington, Virginia, United States, 2004. AUAI Press.

Siddharth Srivastava, Stuart Russell, Paul Ruan, and Xiang Cheng. First-Order Open-Universe POMDPs. In *Uncertainty in Artificial Intelligence*, 2014.

Stan Development Team. Stan: A C++ Library for Probability and Sampling, Version 2.4. 2014.

R S Sutton, D McAllester, Satinder Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. *NIPS*, page 7, 1999.

Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. Learning policy improvements with path integrals. In *Artificial Intelligence and Statistics*, pages 828–835, 2007.

Evangelos A Theodorou and Emanuel Todorov. Relative entropy and free energy dualities: Connections to Path Integral and KL control. In *Proceedings of the IEEE Conference on Decision and Control*, pages 1466–1473, 2012. ISBN 978-1-4673-2066-5. doi: 10.1109/CDC.2012.6426381.

Emanuel Todorov. Compositionality of optimal control laws. In *Neural Information Processing Systems*, 2009a.

Emanuel Todorov. Efficient computation of optimal actions. *Proc. Nat. Acad. Sci. of America*, 106(28):11478–11483, 2009b.

Bart van den Broek, Wim Wiegerinck, and Bert Kappen. Risk sensitive path integral control. *Uncertainty in Artificial Intelligence*, 2010. ISSN 15687805.

Nikos Vlassis, Marc Toussaint, Georgios Kontes, and Savas Piperidis. Learning Model-free robot control by a Monte Carlo em algorithm. *Autonomous Robots*, 27(2):123–130, 2009. ISSN 09295593. doi: 10.1007/s10514-009-9132-0.

Martin J Wainwright and Michael I Jordan. Graphical Models, Exponential Families, and Variational Inference. *Foundations and Trends in Machine Learning*, 1(12):1–305, 2008.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.

David Wingate and Theo Weber. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299*, pages 1–7, 2013.

David Wingate, Carlos Diuk, Timothy O Donnell, Joshua Tenenbaum, Samuel Gershman, Lyric Labs, and Joshua B Tenenbaum. Compositional Policy Priors. Technical report, Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, 2013.

F Wood, JW van de Meent, and V Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

A Source Code for Case Studies

For illustrative purposes, we include the implementations of each of our case studies in the probabilistic programming system Anglican. The full source code for the case studies is available at

`https://bitbucket.org/probprog/bbps-paper-source-code`

Anglican is a probabilistic programming language that is closely integrated into the Clojure language. The macro `defquery` is used to define a probabilistic model. Anglican programs may make use of user-written Clojure functions (defined with `defn`) as well as user-written Anglican functions (defined with `defm`). The difference between the two is that in Anglican functions may make use of the model special forms `sample`, `observe`, and `predict`, which interrupt execution and require action by the inference backend. In Clojure functions, `sample` is a primitive procedure that generates a random value, `observe` returns a log probability, and `predict` is not available.

We note that the interrupt type $(\texttt{learn}, \beta, f, \eta)$ defined in the main text is implemented in Anglican as a normal sample interrupt $(\texttt{sample}, \alpha, f, \phi)$, where the inference algorithm learns variational parameters for all distributions, unless a type annotation `(with-meta f :ignore true)` is used to indicate that parameters should *not* be learned.

Full documentation for Anglican can be found at

`http://www.robots.ox.ac.uk/˜fwood/anglican`

A.1 Canadian Traveler Problem

```
(ns ctp.model
  (:use [anglican runtime emit]))

(defdist factor [] []
  (sample [this] nil)
  (observe [this value] value))

(def +factor+ (factor))

(defn argmax [thing]
  (first (apply max-key second thing)))

(defm sample-and-ignore
  [id dist]
  (sample id (with-meta dist {:ignore true})))

;;;; Canadian Traveller Problem

;; The graph is attributed by two probabilities for each
;; edge --- that the edge is open, and that the traveller
;; [dis]likes the edge.
;;
;; The objective is to learn a policy of traversal order
;; that maximizes the probability of arriving alive.
;;
;; The stochastic policy is represented by vector of
;; probabilities of selecting each edge.

;;; Default values for parameters.
;;
;; Parameters can be passed via the initial value as
;;   [instance p-open gas-cost]
;;
;; Instances are in ctp.data/instances.

(defm travel
  "Performs a travel through the graph;
  Parameters:
    graph,
    s - starting node,
    t - goal node,
    policy - function from node
             to children selection probabilities;
    open? - function from edge to boolean state
    select-action - function from node and selection probabilities
                    to a child.
  The last parameter to replace normal (sample ...) during inference
  with (embang.runtime/sample ...) during policy evaluation."
  [graph s t policy open? select-action]
  ;; All edges are open or blocked with the same probability.
  (let [;; Used to compute the walk distance.
        edge-weight (fn [u v]
                      (some (fn [child]
                              (if (= (first child) v)
                                (second child)))
                            (nth graph u)))

        ;; Performs depth-first search from u to t.
        ;;  Returns a tuple [connected passed-distance].
        dfs (fn dfs [u t]
              (if (= u t)
                [true 0.]
                (loop [Tu (filter (fn [choice]
                                    (open? u (first choice)))
                                  (policy u))
                       ;; Start with zero passed distance.
                       passed 0.]
                  ;; On every step of the loop, filter visited
                  ;; edges from the transition vector.
                  (let [Tu (filter
                             (fn [choice]
                               (not (contains?
                                      (retrieve ::visited)
                                      #{u (first choice)})))
                            Tu)]

                    (if (empty? Tu)
                      [false passed]
                      ;; Sample an action
                      (let [v (select-action u Tu)]
                        ;; Search for the goal in the subtree.
                        (store ::visited
                               (conj (retrieve ::visited)
                                     #{u v}))
                        (let [res (dfs v t)
                              passed (+ passed
                                        (edge-weight u v)
                                        (second res))]
                          (if (first res)
                            ;; Goal found in the subtree.
                            [true passed]
                            ;; Continue in another subtree.
                            (recur
                              Tu
                              ;; Add the weight of the edge
                              ;; through which we return.
                              (+ passed (edge-weight v u))))))))))]

    (store ::visited #{})
    (dfs s t)))

(defm predict-policy
  "predict the policy"
  [graph policy]
  (loop [nodes (range (count graph))]
    (when (seq nodes)
      (let [[u & nodes] nodes]
        (loop [Tu (policy u)]
          (when (seq Tu)
            (let [[[v p] & Tu] Tu]
              (predict [:T u v] p)
              (recur Tu))))
        (recur nodes)))))

(with-primitive-procedures [argmax]
  (defquery ctp
    "predicting policy for CTP"
    [instance p-open gas-cost]
    (let [graph (get instance :graph)
          s (get instance :s)
          t (get instance :t)

          ;; Fix transition probabilities in every node.
          T (reduce
              (fn [T u]
                (assoc T u
                       (let [children (map first (nth graph u))
                             policy (sample u (dirichlet
                                                (repeat
                                                 (count children)
```

```
                                             1.))))]
                         ;; Build argument for the categorical
                         ;; distribution: selection probability
                         ;; for each child.
                         (map vector children policy))))
              {} (range (count graph)))

          ;; Policy function for the agent.
          policy (fn [u] (get T u))

          ;; Instantiate the graph --- this is the model noise.
          open? (let [open-belief (flip p-open)]
                  (mem (fn [u v] (sample-and-ignore
                                  [u v]
                                  open-belief))))

          ;; Compute the predictions
          [connected distance] (travel graph s t policy open?
                                      (fn [u Tu]
                                        (argmax Tu)))]
                                        ;;(sample u (categorical Tu))))]

      ;; Reject disconnected items.
      (observe (flip 1.0) connected)
      ;; Observe how the agent liked the journey.
      (observe +factor+ (* (- distance) gas-cost))
      (predict-policy graph policy))))
```

A.2 RockSample POMDP

```
(ns rockwalk.model
  "RockSample POMDP solving using Probabilistic Inference"
  (:require [anglican.dist :refer [+factor+]]
            [clojure.tools.cli :as cli])
  (:use [anglican runtime emit
         [state :only [get-predicts]]
         [core :only [doquery]]]
        [rockwalk data]))

(defm sample-and-ignore
  [id dist]
  (sample id (with-meta dist {:ignore true})))

;; We implement an offline algorithm for the RockSample POMDP.
;; At every step, the agent selects the closest stone,
;; senses it, and either goes for the stone or discards it.
;; When there are no stones left, the agent heads straight
;; to the right edge.

;; A problem instance is defined by
;;  - the field size (n n);
;;  - locations and values of rocks (good/bad).
;;  - location of the agent.

(defrecord state [n rocks x y])
;; where 'rocks' is a hash map
;;   [^Integer x ^Integer y] -> ^Boolean good

;; The agent always starts at the middle of the left edge.
;; The goal state is beyond the right edge.

(defn goal?
  "true if the state is a goal state"
  [state]
  (= (:x state) (:n state)))

;; The sensor returns a noisy observation of rock value. The
;; accuracy decreases exponentially with the distance. At zero
;; distance, the sensor always returns the correct value. At the
;; half-efficiency distance (hed), the correct value is returned
;; with probability 0.75. At infinity, the correct and the
;; incorrect values are equally probable.

(defn accuracy
  "computes the probability of returning the correct rock value
  by the sensor"
  [state x y hed]
  (let [d (let [dx (- (:x state) x)
                dy (- (:y state) y)]
            (Math/sqrt (+ (* dx dx) (* dy dy))))
        efficiency (Math/pow 0.5 (/ d hed))]
    (* 0.5 (+ efficiency 1.))))

;; The robots moves rectilinearly, the manhattan distance is
;; chosen to choose a rock and compute the reward.

(defn distance
  "distance between the current and the next location"
  [state x y]
  (+ (Math/abs (- (:x state) x))
     (Math/abs (- (:y state) y))))

;; In the original formulation of RockSample the moves are free,
;; so the optimum policy is to go to every rock, know its value
;; with certainty, and sample if good. This problem formulation
;; still works for assessing value iteration algorithms because
;; of implicitly assumed reward discounting. However, a sound
;; problem formulation would either state the discounting factor
;; explicitly, limit the number of moves, or incur a cost on
;; every move.
;;
;; Here, we modify the problem formulation, so that in addition
;; to the rewards for sampling a good rock and for reaching the
;; right edge, there is a penalty for every move. All moves in
;; our space of policies are legal, and the robot never samples
;; a bad rock.

(def +sample-reward+    10.)
(def +move-reward+      -1.)
(def +goal-reward+      10.)

(defn goto
  "goes to a target location;
  samples the rock in that target location
  if the rock is there and good;
  returns the updated state and the reward"
  [state [x y :as loc]]
  [(assoc state
     :x x :y y
     :rocks (dissoc (:rocks state) loc))
   (+ (* (distance state x y) +move-reward+)
      (if ((:rocks state) loc)
        +sample-reward+
        0.0))])

;; A rock can be removed without going to the new location,
;; just because it is not deemed worth the attention.

(defn discard
  "discards the rock, returns the updated state"
  [state loc]
  (assoc state :rocks (dissoc (:rocks state) loc)))

;; At every step, the next rock is the closest rock
;; in the left-most column containing rocks

(defn next-rock
  "Returns the coordinates of the next rock"
  [state]
  (loop [nloc nil
         locs (keys (:rocks state))]
    (if (seq locs)
      (let [[[x y :as loc] & locs] locs
            [nx ny] nloc]
        (if (or (nil? nloc)
                (< x nx)
                (and (= x nx)
                     (< (distance state x y)
                        (distance state nx ny))))
          (recur loc locs)
          (recur nloc locs)))
      nloc)))

;; Now we can define the query that learns the thesholds.
(with-primitive-procedures [goal? accuracy distance
                            next-rock goto discard]
  (defquery rockwalk
    "rockwalk policy learning"
    [instance hed scale]
    (let [rocks (into
                 {}
```

```
           (map (fn [[loc _]]
                  [loc
                   (sample-and-ignore
                     [:rock loc] (flip 0.5))])
                (:rocks instance)))]
(loop [state (assoc instance
              :rocks rocks)
       visited []
       reward 0]
  (if (goal? state)
    (do
      ;; (observe +factor+ (* reward scale))
      (predict :visited visited)
      (predict :reward reward))
    (let [loc (next-rock state)]
      (if (nil? loc)
        ;; no rocks left, go to the goal
        (let [goal [(:n state) (:y state)]
              [state r] (goto state goal)]
          (observe +factor+ (* r scale))
          (recur state
                 visited
                 (+ reward r)))
        (let [;; sample sensor reading for next rock
              good (sample-and-ignore
                    [:sense [(:x state) (:y state)]]
                    (flip
                     (let [[x y] loc]
                       (if (get (:rocks state) loc)
                         (accuracy state x y hed)
                         (- 1. (accuracy state x y hed))))))
              ;; decide whether to visit the rock
              ;; (this is the policy choice)
              visit (sample
                     [:policy [(:x state) (:y state)] loc good]
                     (flip 0.5))]
          (if visit
            ;; goto to rock, gain reward if rock is good
            (let [[state r] (goto state loc)]
              (observe +factor+ (* r scale))
              (recur state
                     (conj visited loc)
                     (+ reward r)))
            ;; remove rock from list of visitable rocks
            (let [state (discard state loc)]
                  (recur state
                         visited
                         reward))))))))))
```

A.3 Guess Who

A.3.1 Dataset

```
id,beard,ear-rings,eye-color,gender,glasses,hair-color,hair-length,hair-type,hat,moustache,mouth-size,nose-size,red-cheek
alex,false,false,brown,male,false,black,short,straight,false,true,large,small,false
alfred,false,false,blue,male,false,ginger,long,straight,false,true,small,small,false
anita,false,false,blue,female,false,blonde,long,straight,false,false,small,small,true
anne,false,true,brown,female,false,black,short,curly,false,false,small,large,false
bernard,false,false,brown,male,false,brown,short,straight,true,false,small,large,false
bill,true,false,brown,male,false,ginger,bald,straight,false,false,small,small,true
charles,false,false,brown,male,false,blonde,short,straight,false,true,large,small,false
claire,false,false,brown,female,true,ginger,short,straight,true,false,small,small,false
david,true,false,brown,male,false,blonde,short,straight,false,false,large,small,false
eric,false,false,brown,male,false,blonde,short,straight,true,false,large,small,false
frans,false,false,brown,male,false,ginger,short,curly,false,false,small,small,false
george,false,false,brown,male,false,white,short,straight,true,false,large,small,false
herman,false,false,brown,male,false,ginger,bald,curly,false,false,small,large,false
joe,false,false,brown,male,true,blonde,short,curly,false,false,small,small,false
maria,false,true,brown,female,false,brown,long,straight,true,false,small,small,false
max,false,false,brown,male,false,black,short,curly,false,true,large,large,false
paul,false,false,brown,male,true,white,short,straight,false,false,small,small,false
peter,false,false,blue,male,false,white,short,straight,false,false,large,large,false
philip,true,false,brown,male,false,black,short,curly,false,false,large,small,true
richard,true,false,brown,male,false,brown,bald,straight,false,true,small,small,false
robert,false,false,blue,male,false,brown,short,straight,false,false,small,large,true
sam,false,false,brown,male,true,white,bald,straight,false,false,small,small,false
susan,false,false,brown,female,false,white,long,straight,false,false,large,small,true
tom,false,false,blue,male,true,black,bald,straight,false,false,small,small,false
```

A.3.2 Model Auxilliary Functions

```
(ns guess-who.model
  (:require [guess-who
             [data :refer [+entities+ +questions+]]]
            [anglican
             [math :refer [max-entries]]
             [runtime :refer [log exp log-sum-exp]]]))

;; GUESS WHO
;;
;; entities {id {attr value}}
;; questions [[attr value]]
;; info {[attr value] [num-true num-false]}
;; belief [[id prob]]

(def +reliability+ 0.9)

(defn update-info
  "incorporates a question response into the information state"
  [info question response]
  (let [[a b] (get info question [0 0])]
    (assoc info
      question (if response
               [(inc a) b]
               [a (inc b)]))))

(defn log-likelihood
  "returns log p(info | entity), the joint log probability
  of responses given the attribute values of an entity"
  [info entity]
  (reduce
    +
    0.0
    (map (fn [[[attr value] [a b]]]
                   (let [;; get probability of true response
                 pi (if (= (entity attr) value)
                        +reliability+
                        (- 1 +reliability+))]
             ;; binomial probabilities of responses
             (+ (* (log pi) a) (* (log (- 1 pi)) b))))
         info)))

(defn posterior-belief
  "returns p(id | info) the normalized posterior belief"
  [info]
  (let [log-ws (map (partial log-likelihood info)
                    (vals +entities+))
        log-sum-w (reduce log-sum-exp log-ws)]
    (map vector
         (keys +entities+)
         (map #(exp (- % log-sum-w))
              log-ws))))

(defn response-probability
  "returns p(response | belief), the marginal question response
  probability given current belief"
  [belief question]
  (let [;; calculate prior probability that question is true
        ;; (given belief)
        [attr value] question
        [a b] (reduce
                (fn [[a b] [id prob]]
                  (if (= (get-in +entities+ [id attr]) value)
                    [(+ a prob) b]
                    [a (+ b prob)]))
                [0.0 0.0]
                        belief)
        pi (/ a (+ a b))]
    ;; probability of 'true' response given belief
    (+ (* +reliability+ pi)
       (* (- 1 +reliability+) (- 1 pi)))))

(defn relative-utility [initial-belief final-belief]
  "returns the expected change in reward of a final belief state
  relative to an initial belief state"
  (let [initial-candidates (max-entries initial-belief)
        final-belief (into {} final-belief)]
    (- (reduce max (vals final-belief))
       (/ (reduce + (map final-belief initial-candidates))
          (count initial-candidates)))))
```

A.3.3 Value of Information

```
(ns guess-who.heuristics
  "non-sampling based heuristics for the expected value of an action"
  (:use guess-who.model
        [anglican.math :only [max-entries]]
        [guess-who.data :only [+entities+ +questions+]]))

(defn myopic-voi
  "calculates a myopic estimate of the value of information"
  [info question]
  (let [belief (posterior-belief info)
        theta (response-probability belief question)
        u-true (relative-utility belief
                                 (posterior-belief
                                  (update-info info
                                               question
                                               true)))
        u-false (relative-utility belief
                                  (posterior-belief
                                   (update-info info
                                                question
                                                false)))]
    (+ (* theta u-true)
       (* (- 1 theta) u-false))))

(defn recursive-voi
  "calculates a recursive estimate of the value of information"
  [info depth question]
  (let [belief (posterior-belief info)
        candidates (max-entries belief)
        theta (response-probability belief question)
        vois (map (fn [response]
                    (let [belief (posterior-belief info)
                          new-info (update-info info question response)
                          new-belief (posterior-belief new-info)]
                      (+ (relative-utility belief new-belief)
                         (if (<= depth 1)
                           0.
                           (reduce max
                                   (map (fn [next-question]
                                          (recursive-voi new-info
                                                         (dec depth)
                                                         next-question))
                                        +questions+))))))
                  [true false])]
    (reduce +
            (map * [theta (- 1.0 theta)] vois))))
```

A.3.4 Policies

```
(ns guess-who.policy
  "Policy implementations for Guess Who"
  (:require [clojure.core.matrix :as mat
             :refer [add sub div mul mmul array]]
            [guess-who
             [data :refer [+questions+ +entities+]]
             [model :refer [posterior-belief]]
             [heuristics :refer [myopic-voi recursive-voi]]]
            [anglican
             [math :refer [max-entries max-index sigmoid]]])
  (:use [anglican runtime emit
         [state :only [get-predicts]]
         [core :only [doquery]]]))


(mat/set-current-implementation :vectorz)

(defprotocol Policy
  (select [self info]))

;; select questions at random
(defrecord UniformPolicy
  []
  Policy
  (select
   [self _]
   (rand-nth +questions+)))

;; select questions according to highest myopic value of information
```

```
(defrecord MyopicVoiPolicy
  []
  Policy
  (select
   [self info]
   (let [value-estimate (map vector
                             +questions+
                             (map (partial myopic-voi info)
                                  +questions+))
         best-questions (max-entries value-estimate)]
     (rand-nth best-questions))))

;; multiplies the belief vector with a weight matrix and
;; selects the question according to the max entry of the
;; resulting vector
(defrecord LinearBeliefPolicy
  [weights]
  Policy
  (select
   [self info]
  (let [belief (mapv second (posterior-belief info))
        values (mmul weights belief)]
    (get +questions+
         (sample (discrete (mat/to-nested-vectors values)))))))

;; like linear policy but also discounts weights of previously
;; asked questions
(defrecord DiscountedBeliefPolicy
  [weights gamma]
  Policy
  (select
    [self info]
    (let [qcounts (map (fn [q] (apply + (info q))) +questions+)
          discounts (mapv #(Math/pow gamma %) qcounts)
          belief (mapv second (posterior-belief info))
          values (mul discounts (mmul weights belief))]
      (get +questions+
           (max-index values)))))

;; applies logistic regression transform to vectorized
;; information state, and returns the max index of the
;; resulting vector
(defrecord LogisticInfoPolicy
  [weights bias]
  Policy
  (select
   [self info]
   (let [info-vector (mapcat (fn [q]
                               (get info q [0.0 0.0]))
                             +questions+)
         values (sigmoid (add (mmul weights
                                    info-vector)
                             bias))]
     (get +questions+
          (sample (discrete (mat/to-nested-vectors values)))))))

(defn make-policy [policy-type parameters]
  (case policy-type
    :uniform (->UniformPolicy)
    :myopic-voi (->MyopicVoiPolicy)
    :linear-belief (apply ->LinearBeliefPolicy parameters)
    :discounted-belief (apply ->DiscountedBeliefPolicy parameters)
    :logistic-info (apply ->LogisticInfoPolicy parameters)
    (throw (Exception. "policy-type must be one of [:uniform :myopic-voi :linear-belief :discounted-belief :logistic-info
```

A.3.5 Episode Simulation

```
(ns guess-who.trial
  "Guess Who POMDP solving with BBEM policy search"
  (:require [clojure.core.matrix :as mat :refer [array]]
            [guess-who
             [data :refer [+questions+ +entities+]]
             [heuristics :refer [myopic-voi recursive-voi]]
             [model :refer [+reliability+ posterior-belief
                            response-probability relative-utility update-info]]
             [policy :refer :all]]
            [anglican
             [trap :refer [value-cont]]
             [state :refer [initial-state]]
             [math :refer [max-entries max-index]]
             [bbvb :refer [get-raw-proposals]]
             bbem])
  (:use [anglican runtime emit
         [state :only [get-predicts]]
         [core :only [doquery]]]))

(defn value-state-cont
  "returns both value and state"
  [v s]
  [v s])

(with-primitive-procedures
  [update-info posterior-belief response-probability
   relative-utility max-index array make-policy select]

  (defm sample-policy
    "samples policy parameters and returns a policy instance of specified type"
    [policy-type]
    (let [sample-weights (fn [] (array
                               (map (fn [q]
                                      (map (fn [e]
                                             (sample
                                              [q e]
                                              (gamma 100.0 100.0)))
                                           (keys +entities+)))
                                    +questions+)))

          params (case policy-type
                   :linear-belief
                   (let [weights (sample-weights)]
                     [weights])

                   :discounted-belief
                   (let [weights (sample-weights)
                         gamma (sample :gamma (beta 1 1))]
                     [weights gamma])

                   :logistic-info
                   (let [weights (array
                               (map (fn [q1]
                                      (flatten
                                       (map (fn [q2]
                                              [(sample [:weights q1 q2 true]
                                                       (gamma 10.0 100.0))
                                               (sample [:weights q1 q2 false]
                                                       (gamma 10.0 100.0))])
                                            +questions+)))
                                    +questions+))
                         bias (array
                               (map (fn [q]
                                      (sample [:bias q]
                                              (normal 1.0 0.2)))
                                    +questions+))]
                     [weights bias])
                   nil)]
      (make-policy policy-type params)))

  (defquery simulate-episode
    "simulates an episode of guess who, sampling responses
    based on the current belief"
    [policy-type depth inverse-temp initial-info]
    (let [initial-info (or initial-info {})
          initial-belief (posterior-belief initial-info)
          inverse-temp (or inverse-temp 1.0)
          policy (sample-policy policy-type)]
      (loop [questions []
             info initial-info
             belief initial-belief
             reward 0.0]
        (if (>= (count questions) depth)
          (do
            (predict :policy policy)
            (predict :questions questions)
            (predict :reward reward)
            (predict :info info))
          (let [;; select question according to sampled policy
                question (select policy info)
                ;; simulate response according to marginal
                ;; probability given current belief
                response (sample
                          (with-meta
                              (flip (response-probability belief question))
                              {:ignore true}))
                ;; update information and belief state
                new-info (update-info info question response)
                new-belief (posterior-belief new-info)
                ;; update reward
                new-reward (relative-utility initial-belief new-belief)]
            ;; factor according to change in reward
            (observe (flip (exp (* inverse-temp (- new-reward reward)))) true)
            ;; continue to next question
            (recur (conj questions question)
                   new-info
                   new-belief
                   new-reward)))))))

(defn learn-policies
  "learns a policy using BBEM inference. returns empirical
  distribution of policies from last iteration"
  [policy-type depth number-of-steps
   & {initial-proposals :initial-proposals
      number-of-particles :number-of-particles
      base-stepsize :base-stepsize
      adagrad :adagrad
      robbins-monro :robbins-monro
      :or {:number-of-particles 100
           :base-stepsize 1.0
           :adagrad 0.9
           :robbins-monro 0.9}}]
  (let [samples (->> (doquery :bbem
                              simulate-episode
                              [policy-type depth]
                              :initial-proposals initial-proposals
                              :number-of-particles number-of-particles
                              :base-stepsize base-stepsize
                              :adagrad true
                              :initial-proposals initial-proposals)
                     (drop (* number-of-steps number-of-particles))
                     (take number-of-particles))
        proposals (get-raw-proposals (first samples))
        policies (map (comp :policy get-predicts)
                      samples)]
    [policies proposals]))

(defn test-episode
  "plays a guess who episode with fixed policy
  and returns the final reward"
  [true-id policy number-of-questions]
  (loop [number-of-questions number-of-questions
         info {}]
    (if (zero? number-of-questions)
      (let [belief (posterior-belief info)
            guess-id (rand-nth (max-entries belief))]
        (if (= guess-id true-id)
          1.
          0.))
      (let [question (select policy info)
            [attr value] question
            response (if (sample (flip +reliability+))
                       (= (get-in +entities+ [true-id attr]) value)
                       (not= (get-in +entities+ [true-id attr]) value))]
        (recur (dec number-of-questions)
               (update-info info question response))))))

(defn test-sweep
  "runs a test episode for each entity multiple
  times and returns the total reward"
  [policy number-of-questions number-of-sweeps]
  (reduce
   (fn [reward id]
     (+ reward
        (test-episode id
                      policy
                      number-of-questions)))
   0.0
   (reduce concat
           (repeat number-of-sweeps
                   (keys +entities+)))))

(defn trial
  "learns a policy and runs a number of test sweeps. returns
  the rewards for all test episodes and the learned policy."
```

```
[policy-type number-of-questions
 number-of-particles number-of-steps number-of-test-sweeps
 & {base-stepsize :base-stepsize
    adagrad :adagrad
    robbins-monro :robbins-monro
    initial-proposals :initial-proposals
    :or {:base-stepsize 1.0
         :adagrad 0.9
         :robbins-monro 0.9}}]
(let [[policies proposals] (learn-policies policy-type
                                           number-of-questions
                                           number-of-steps
                                           :number-of-particles number-of-particles
                                           :base-stepsize base-stepsize
                                           :adagrad adagrad
                                           :robbins-monro robbins-monro)
      rewards (doall (map #(test-sweep % number-of-questions number-of-test-sweeps)
                          policies))]
  [rewards policies proposals]))
```