Learning to Explore and Build Maps*

David Pierce and Benjamin Kuipers
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
dmpierce@cs.utexas.edu, kuipers@cs.utexas.edu

Abstract

Using the methods demonstrated in this paper, a robot with an unknown sensorimotor system can learn sets of features and behaviors adequate to explore a continuous environment and abstract it to a finite-state automaton. The structure of this automaton can then be learned from experience, and constitutes a *cognitive map* of the environment. A generate-and-test method is used to define a hierarchy of features defined on the raw sense vector culminating in a set of continuously differentiable *local state variables*. Control laws based on these local state variables are defined for robustly following paths that implement repeatable state transitions. These state transitions are the basis for a finite-state automaton, a discrete abstraction of the robot's continuous world. A variety of existing methods can learn the structure of the automaton defined by the resulting states and transitions. A simple example of the performance of our implemented system is presented.

Introduction

Imagine that you find yourself in front of the control console for a teleoperated robot (Figure 1). The display at the left of the console is your only sensory input from the robot. The joystick on the right can be used to move the robot through its environment. You do not know how the joystick affects the robot's motion, but you do know that the robot does not move when the joystick is in its zero position. The robot could be a mobile robot in an office building or laboratory or it could be a submarine in the ocean. Your mission is to develop a model of the robot's environment as well as its sensorimotor interface to that environment.

We present a solution to this learning problem by showing how an autonomous agent can learn a discrete model of its continuous world with no *a priori* knowledge of the structure of its world or of its sensorimotor apparatus. The solution is composed of two steps: 1) learning an abstraction of the continuous world to a finite-state automaton and 2) inferring the structure of the finite-state automaton. The second step has been studied extensively (e.g., Angluin 1978, Gold 1978, Kuipers 1978, Angluin 1987, Rivest & Schapire 1993). In this paper, we focus on the first step. The rest of the paper gives the details of the abstraction-learning method illustrated with an example from the world of mobile robotics.

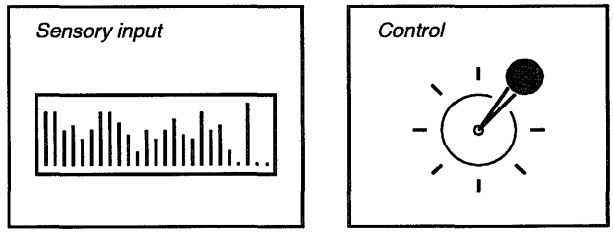


Figure 1: The *tabula rasa* learning problem is illustrated by this interface to a teleoperated robot with an uninterpreted sensorimotor apparatus in an unknown environment. The problem is to develop a practical understanding of the robot and its environment with no initial knowledge of the meanings of the sensors or the effects of the control signals.

Overview: From continuous world to finite-state automaton

The details of the method for learning a discrete abstraction of a continuous world are given below and are illustrated in Figure 2.

Given: a robot with an uninterpreted, well-behaved sensorimotor apparatus, in a continuous, static world. The definition of "well-behaved" is given in a later section.

Learn: a discrete abstraction of the continuous world, specifically, a finite-state automaton.

The solution method involves three stages.

*This work has taken place in the Qualitative Reasoning Group at the Artificial Intelligence Laboratory, The University of Texas at Austin. Research of the Qualitative Reasoning Group is supported in part by NSF grants IRI-8904454, IRI-9017047, and IRI-9216584, and by NASA contracts NCC 2-760 and NAG 9-665.

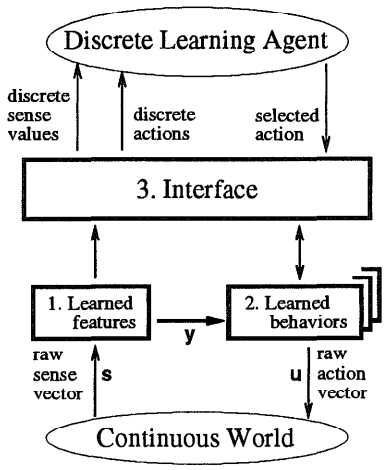


Figure 2: The continuous world is abstracted to a finite-state automaton thereby reducing the task of modeling a continuous world to the well-understood task of modeling a finite-state world. The abstraction is created by 1) learning a set of smooth features y defined as functions of the raw sense vector s, 2) learning reliable path-following behaviors based on those features, and 3) defining an interface that provides the discrete sense values and actions of the finite-state automaton. The *discrete learning agent* is any learning mechanism that can infer the structure of a finite-state automaton.

1. Feature learning

- Learn a set of almost-everywhere smooth (continuously differentiable) *features*, defined as functions on the raw sensory input. A generate-and-test method is used.

2. Behavior learning

- Learn a model of the motor apparatus, specifically, a set of *primitive actions*, one per degree of freedom. A typical mobile robot has two degrees of freedom: translation and rotation.
- Learn a model (called the *static action model*) for predicting the context-dependent effects of primitive actions on features.
- Use the static action model to define a set of behaviors for finding paths using hill-climbing, and an initial set of open-loop behaviors for following paths.
- Learn a model (called the *dynamic action model*) for predicting the effects of actions while path-following.
- Use the dynamic action model to define closed-loop behaviors for following paths more reliably.

3. Defining the abstraction

- Define an interface to the robot's sensorimotor apparatus that abstracts the continuous environment to a finite-state automaton. The learned path-following behaviors produce state transitions; their terminations define states.

The abstraction-learning method has been applied to a simulated mobile robot. Its sensory system consists of a ring of 16 distance sensors giving distances to nearby objects up to a maximum distance of two meters away. It has a tank-style motor apparatus with two real-valued control signals in the range [-1,1] specifying the speeds of the left and right treads respectively. The robot can turn in place, move forward, move backward, or do a combination of both. The robot is placed in a simple environment consisting of a single room roughly ten meters square.

Feature learning

A *feature*, as defined in this paper, is any function over time whose current value is completely determined by the history of current and past values of the robot's raw sense vector. Examples of features are: the raw sense vector itself (an example of a *vector* feature), the components of that vector (each an example of a *scalar* feature), the average value of those components, and the derivatives with respect to time of those components.

Local state variables

The state of a robot in a continuous world is represented by a state vector $\mathbf{x}$ whose components are called *state variables*. For example, a mobile robot's state can be described by the state vector $\mathbf{x} = (x_1, x_2, \theta)$ where x_1 and x_2 give the robot's position and θ its orientation. The *tabula rasa* robot does not have direct access to its state variables, but it can use learned scalar features as *local state variables*. A local state variable is a scalar feature whose gradient (its vector of derivatives with respect to the robot's set of state variables) is approximately constant and nonzero over an open region in state space. When the robot is in that region, a local state variable provides one coordinate of information about the state of the robot: constraining that feature to have a specific value reduces the dimensionality of the world by one. With enough independent features (i.e., features whose gradients are not collinear), the state of the robot can be completely determined, and hence the values of *all* of its features can be determined.

The catch is that the robot cannot directly measure feature gradients since it does not have direct access to its state. However, if the world is static and the effects of the actions can be approximated as linear for a local open region R, then the robot can recognize features suitable as local state variables by analyzing the effects of the actions on the features as is shown below. In the following, $\mathbf{x}$ is the unknown state vector for the robot, y is a scalar feature, $\dot{y}$ is the derivative of y with respect to time, and $\mathbf{u}$ is the robot's vector of motor control signals. The first equation is the general

formulation for a dynamical system.

$$
\begin{array}{lrcll}
(1) & \dot{\mathbf{x}} & = & f(\mathbf{x},\mathbf{u}) & \\
(2) & f(\mathbf{x},\mathbf{0}) & = & \mathbf{0} & \{\text{static world}\} \\
(3) & f(\mathbf{x},\mathbf{u}) & \approx & F_R\,\mathbf{u} & \{2, \text{linearity assumption}\} \\
(4) & \dot{y} & = & \frac{\partial y}{\partial \mathbf{x}}\dot{\mathbf{x}} & \{\text{chain rule}\} \\
(5) & \dot{y} & \approx & \frac{\partial y}{\partial \mathbf{x}} F_R\,\mathbf{u} & \{1, 3\}
\end{array}
$$

Here F_R is a matrix used to approximate function f as a linear function of $\mathbf{u}$ for states in open region R containing $\mathbf{x}$. The conclusion from equation (5) is that $\dot{y}$ is a nonzero, linear function of the action vector $\mathbf{u}$ if and only if the gradient $\frac{\partial y}{\partial \mathbf{x}}$ is nonzero and approximately constant. Therefore in order to recognize a local state variable (a feature with a nonzero, approximately constant gradient), the robot need only demonstrate that the feature's derivative can be estimated as a linear function of the action vector. This is the basis of the test portion of the generate-and-test feature-learning algorithm. A feature y is a *candidate* local state variable if its derivative is small for small actions. It is a *bona fide* local state variable, for a region R, if its derivative can be estimated by $\dot{y} = W_R\,\mathbf{u}$ where W_R is a matrix. Learning and representing this matrix is the job of the static action model. A sensorimotor apparatus is *well-behaved* if the following is almost-everywhere true: Given state $\mathbf{x}$ there exists an open region R containing $\mathbf{x}$ such that the dynamics of the sensorimotor apparatus can be approximated by $\dot{\mathbf{s}} = W_R\mathbf{u}$ where $\mathbf{s}$ is the raw sense vector and W_R is a matrix that depends on the region. This is not as restrictive as it first appears, since nonlinear and discontinuous functions can often be approximated by piecewise linear functions.

A generate-and-test approach

A robot's raw sensory features may not be very suitable as local state variables. For example, sonar sensors have discontinuities due to specular reflection. Their values are smooth for small regions in state space, but it is preferable to have features usable over larger regions. Fortunately, by applying the *min* operator to a set of sonar values, a new feature is obtained that is smooth over a much larger region and thus more suitable as a local state variable. This example illustrates the principle that, if a sensory system does not directly provide useful features, it may be possible to define features that are useful. This principle is the foundation for a generate-and-test approach to feature learning, implemented with a set of *generators* for producing new features and *testers* for recognizing useful features.

The following set of generators, when applied to a sensory system with a ring of distance sensors, are instrumental in discovering features corresponding to minimum distances to nearby objects. The first two, the *group* and *image* generators, are useful for analyzing the structure of a sensory system and defining features that are the basis for higher-level features such as motion detectors.

- The *group* generator splits a vector feature into subvectors, called group features, of highly correlated components. When applied to a sensory system with distance sensors and compass, the distance sensors are collected into a single group.
- The *image* generator takes a group feature and associates a position with each component thus producing an image feature whose structure reflects the structure of the group of sensors. A relaxation algorithm (Figure 3) is used to assign positions to components so that the distance between two components s_i and s_j in the image is proportional to the measured dissimilarity $d(i,j) = \sum_t |s_i(t) - s_j(t)|$. The image generator is based on the principle that sensors that are physically close together will, on average, produce similar values (Pierce 1991).

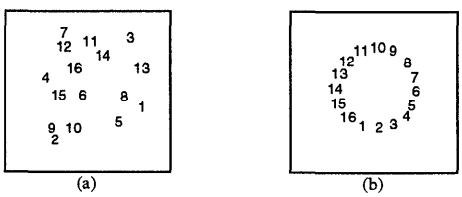


Figure 3: A relaxation algorithm is used to define an image feature for the group feature containing the 16 distance sensors. a) The components are randomly assigned to positions in a two-dimensional plane. b) The components self-organize into a ring in which components with similar values appear close together in the image.

- The *local-minimum* generator takes an image feature and produces a "blob" image feature. A blob image feature provides a mechanism for focus of attention by associating a binary strength with each component. Each component in the image whose value is less than that of its neighbors is given strength 1. Figure 4a illustrates the application of this generator to the image feature of distance values.
- The *tracker* generator takes a blob image feature and monitors the blobs (components with strength equal to 1) within it. The output is a list of blob features giving the value and image-relative position of each input blob. Figure 4b illustrates the application of this generator.

The generators are typed objects: the type of a generator is given by the type of feature to which it applies and the type of feature that it creates. For example, the image generator applies to group features and creates image features.

The only tester needed for finding potential local state variables is the *smoothness* tester. A feature is considered smooth if the standard deviation of its derivative with respect to time is less than a certain value. The generate-and-test approach was applied to the robot with 16 distance sensors. None of the distance sensors were identified as smooth, but the

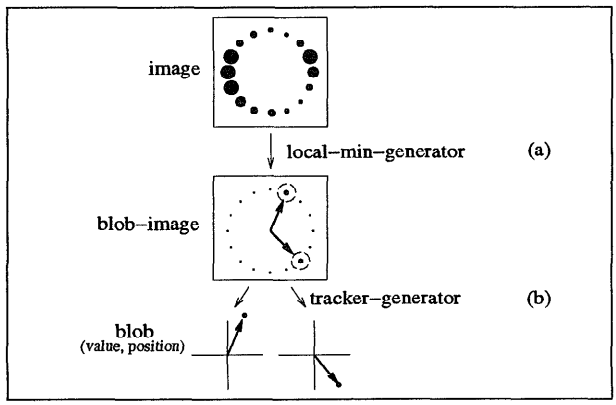


Figure 4: The *local-min* and *tracker* generators are applied to the learned image feature to produce new scalar features that will serve as local state variables. (In this diagram, sense values are represented by the sizes of the disks.) From the robot's perspective, these features are local minima of a sensory image formed by organizing an unstructured, uninterpreted sense vector into a structured ring whose organization reflects intersensor correlations. From our perspective, the features are minimum distances to nearby objects.

learned blob value features were. The size of the search space is kept reasonable by the typing of the feature generators which makes the search space deep but not wide.

Behavior learning

With a good set of features that can serve as local state variables, the problem of abstracting from a continuous world to a finite-state automaton is half solved. The features can be used to express constraints of the form $\mathbf{y} = \mathbf{y}^*$ that define paths. By following such a path until it ends, the robot executes a behavior that can be viewed as an atomic state transition by the discrete learning agent. What remains is to learn behaviors for following paths reliably.

A *behavior* has four components. The *out* component is an action vector used to directly control the motor apparatus. The *app* signal tells whether the behavior is applicable in the current context. The *done* signal tells when the behavior has finished. The *init* input signal is used to begin execution of a behavior. The *out* and *app* components will be defined using the *static* and *dynamic* action models that predict the effects of the robot's actions on the local state variables used to define the path's constraints. A path-following behavior is done when its constraint is no longer satisfied or when a new path-following behavior becomes applicable indicating that the discrete learning agent has a new action to consider.

Primitive actions

The first step toward learning the action models is to analyze the robot's motor apparatus to discover how many degrees of freedom it has and to learn a set of action vectors for producing motion for each degree of freedom. Since the effects of actions are grounded in the sensory features (the only source of information the robot has), we can analyze the motor apparatus by defining a space of action effects and then applying principal component analysis (PCA) to that space in order to define the dimensions of that space and a basis set of effects. See Mardia et al. (1979) for an introduction to PCA.

For example (see Figure 5), characterizing action effects as motion vectors and applying PCA can be used to diagnose the set of primitive actions for a mobile robot with a ring of distance sensors and a motor apparatus that affords rotation and translation actions. For the robot of our running example, this method discovers the primitive action vectors $\mathbf{u}^1 = (-1, 1)$ for one degree of freedom (rotating) and $\mathbf{u}^2 = (1, 1)$ for the second degree of freedom (advancing). The action vector used to control the motor apparatus is a linear combination of the primitive actions, written $\mathbf{u} = \sum_i u_i \mathbf{u}^i$. This method has been successfully applied to both a ring of distance sensors and a small retina (Pierce 1991).

The static action model

The purpose of the static action model is to tell how the primitive actions affect the features that serve as local state variables. Knowing how to take a feature to a target value is necessary for satisfying a constraint, i.e., moving to a path. Knowing how to keep a feature at its target value while moving is necessary for following a path.

In the simplest case, the effect of an action is constant. In general, however, the effect of an action on a feature is context-dependent, meaning that the static action model will have to take context information into account when making its predictions. For example, the advance action will decrease a minimum-distance feature's value when the robot is facing toward a wall; it will increase the value if the robot is facing away from the wall; and it will leave the value invariant if the robot is facing parallel to the wall. The orientation of the robot with respect to the wall is the context that determines the action's effect on the feature.

A brute-force way to define contexts is to break sensory space up into a large set of boxes (cf. Michie & Chambers 1968). In the current example, a more elegant solution is possible. The learned blob features have associated context information, namely the positions of the features in the image from which they are produced (see Figure 4). This information, which encodes the angle of the object whose distance is given by the feature's value, turns out to be sufficient to discriminate the different regions.

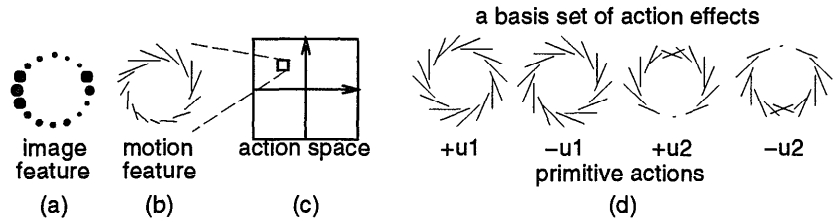


Figure 5: The diagnosis of primitive actions. A *motion* generator is applied to the learned image feature (a). The resulting motion feature (b) is used to characterize actions in terms of average motion vectors (c). Principal component analysis is used to characterize this space of motion vectors to determine a set of actions for motion in each degree of freedom (d).

for each primitive action u_i
  for each feature y_j
    for each context c
      find the best value of w_{ji} in
        $\dot{y}_j \approx w_{ji}\, u_i$
        and the associated correlation, r.

Figure 6: Learning the static action model, i.e., the elements of the context-dependent matrix W_c in $\dot{\mathbf{y}} \approx W_c\, \mathbf{u}$. Linear regression is used to find the values of w_{ji} and r. The correlation between u_i and $\dot{y}_j$ determines the goodness of fit, i.e., the validity of the hypothesis that, for a given context, the relationship is approximately linear.

The learning of the static action model is summarized in Figure 6. While the robot explores by randomly executing one primitive action at a time, linear regression is used to find the best values for w_{ji} in the equation $\dot{y}_j = w_{ji}\, u_i$ as well as a correlation that tells how good the fit is. With this information, it is possible to tell when an action has an effect on a feature (namely, when the correlation associated with the feature's current context is large) and how large that affect is. This is necessary for defining hill-climbing behaviors. It is also possible to tell when (i.e., in what context) an action leaves a feature's value invariant. This is necessary for defining path-following behaviors.

Hill-climbing behaviors. The purpose of hill-climbing behaviors is to move the robot to a state where a path-following behavior is applicable. For each primitive action $\mathbf{u}^i$ and feature y_j, a behavior for hill-climbing to the goal $y_j = y_j^*$ is defined as shown in Figure 7. It is applicable when the static action model predicts that the action is capable of moving the feature toward its target value. It is done when it has succeeded in doing so. Its output is given by a simple control law.

Open-loop path-following behaviors. The static action model does not have enough information to define closed-loop path-following behaviors with error correction to keep the robot on the path, but by using the static model, it is possible to define *open-loop* path-following behaviors. For each primitive action and feature, an open-loop behavior is defined (Figure 8) that is applicable in contexts where, according to the static action model, the action leaves the value of the feature invariant. A signal is superimposed on top of this action that will be used in learning the dynamic action model (next section). After the dynamic action model is learned, a small error-correcting component will be added to keep the robot on the path (i.e., to keep the feature y at the desired value y^*).

Given: $\dot{y}_j \approx w_{ji}\, u_i$ with correlation r for context c.

$$
\begin{aligned}
app(c) &\equiv |r| \gg 0 \\
out(c) &= u_i\, \mathbf{u}^i \\
done(c) &\equiv e_j \approx 0 \text{ where} \\
u_i &= \frac{2\zeta\omega}{w_{ji}}\, e_j + \frac{\omega^2}{w_{ji}} \int e_j\, dt \\
e_j &= y_j^* - y_j .
\end{aligned}
$$

Figure 7: A hill-climbing behavior is defined for each primitive action $\mathbf{u}^i$ and feature y_j to achieve the goal $y_j = y_j^*$. A simple proportional-integral (PI) control law is used with parameters $\zeta = 1.0$, $\omega = 0.05$ (see Kuo 1982).

$$
\begin{aligned}
app(c) &\equiv (y_j \approx y_j^*) \wedge (\dot{y}_j(\mathbf{u}^\beta, c) \approx 0) \\
out &= u_\beta\, \mathbf{u}^\beta + \sum_{\delta \neq \beta} u_\delta \mathbf{u}^\delta
\end{aligned}
$$

Figure 8: An open-loop path-following behavior is defined for each primitive action $\mathbf{u}^\beta$ and feature y_j. The output has two components: a base action and a small orthogonal component used in learning the dynamic action model. Only one of the u_δ's is nonzero at a time. The behavior is applicable when the constraint is satisfied and the action $\mathbf{u}^\beta$ does not change the feature's value. It is done when the constraint is no longer satisfied or a new behavior becomes applicable.

The dynamic action model

The dynamic action model is used to define *closed-loop* (i.e., error-correcting) versions of the path-following behaviors. Specifically, this model will tell, for each path-following behavior, the effect of each orthogonal action (each primitive action other than the path-following behavior's base action), on every feature that is used in the definition of the path-following behavior's constraint.

To learn the dynamic action model, an exploration behavior is used that randomly chooses applicable hill-climbing and open-loop path-following behaviors. An open-loop path-following behavior works by producing the action $\mathbf{u} = u_\beta\,\mathbf{u}^\beta + \sum_\delta u_\delta\,\mathbf{u}^\delta$ where u_β is constant, u_δ is much smaller than u_β, and only one u_δ is nonzero at a time. The behavior runs until it is no longer applicable, it no longer satisfies its constraint, or a new path-following behavior becomes applicable. While it is running, linear regression is used to learn the relationships between the orthogonal actions and the features in the context of running the open-loop path-following behavior. The learning of the dynamic action model is described in Figure 9[1].

for each primitive action $\mathbf{u}^\beta$
  for each feature y_j
    for each orthogonal action u_δ
      for each context c
        find the best values of $n \in \{1, 2\}$, k in
          $y_j^{(n)} \approx k\,u_\delta$
        when $\mathbf{u} = \mathbf{u}^\beta + u_\delta\,\mathbf{u}^\delta$.

Figure 9: Learning the dynamic action model. Here, $y^{(n)}$ is the n^{th} derivative of y. Linear regression is used to find the best value of k. The correlation between u_δ and $y_j^{(n)}$ is used to discover which derivative of the feature is influenced by the action. It is assumed that for a given context, the relationship will be approximately linear.

Closed-loop path-following behaviors. For each primitive action $\mathbf{u}^\beta$ and each feature vector $\mathbf{y}$ (i.e., each possible combination of features), a closed-loop path-following behavior is defined. It is applicable when $\mathbf{u}^\beta$ leaves $\mathbf{y}$ invariant according to the static action model. It is analogous to an open-loop path-following behavior except that it uses the orthogonal actions for error correction. Since each orthogonal action may affect each feature, there will be one term in the control law for each feature-action pair. The details are given in Figure 10.

[1]For the dynamic action model, it is necessary to consider both first and second derivatives of the features. Informally, this is because $\mathbf{u}^\delta$ may affect the derivative of k_β in the equation $\dot{y} = k_\beta\,u_\beta$, that is, $\dot{k}_\beta = k_\delta\,u_\delta$. Together, these give $\ddot{y} = \dot{k}_\beta\,u_\beta = k_\delta\,u_\delta\,u_\beta = k\,u_\delta$, using the fact that u_β is constant for a path-following behavior.

$$
\begin{aligned}
app(c) &\equiv (\mathbf{y} \approx \mathbf{y}^*) \wedge (\dot{\mathbf{y}}(\mathbf{u}^\beta, c) \approx 0) \\
out(c) &= \mathbf{u}^\beta + \sum_\delta u_\delta\,\mathbf{u}^\delta
\end{aligned}
$$

where

$$
\begin{aligned}
u_\delta &= \sum_j u_{\delta j} \\
u_{\delta j} &= \frac{2\zeta\omega}{k}\,e + \frac{\omega^2}{k}\int e\,dt \qquad \text{if } \dot{y}_j \approx k\,u_\delta, \\
u_{\delta j} &= \frac{\omega^2}{k}\,e + \frac{2\zeta\omega}{k}\,\dot{e} \qquad \text{if } \ddot{y}_j \approx k\,u_\delta, \\
e_j &= y_j^* - y_j.
\end{aligned}
$$

Figure 10: Definition of a closed-loop path-following behavior for constraint $\mathbf{y} = \mathbf{y}^*$. Here, $\mathbf{u}^\beta$ is the base action for the behavior and j is the index ranging over the features that comprise vector $\mathbf{y}$. Simple PI and PD (proportional-derivative) controllers are used. Again, ζ=1.0, ω=0.05.

Defining the discrete abstraction

Once the features are learned and the corresponding behaviors are defined, all that remains is to define an interface that presents the continuous world to the discrete learning agent as if the world were a finite-state automaton. A finite-state automaton (FSA) is defined as a tuple $(Q, B, \delta, q_0, \gamma)$ where Q is a finite set of states, B is a set of actions, δ is the next-state function mapping state-action pairs to next states, q_0 is the start state, and γ is the output function determining the sense value associated with each state.

The abstraction we propose involves using a set of learned path-following behaviors that constrain the motion of the robot to a connected network of one-dimensional loci in state space. State transitions correspond to motions resulting from the execution of the path-following behaviors. A state, $q \in Q$, is identified with the point where a path-following behavior ends. At a state, the set of actions $B_q \subset B$ is the set of applicable path-following behaviors. The output $\gamma(q)$ is a symbol that identifies the robot's sensory input at state q. If two states have the same sense vectors, they will have the same output symbol. The next-state function δ is defined by the set of all possible (q, b, q') triples where action b takes the robot from state q to q'. The start state q_0 is taken to be the first state that the robot encounters.

For a path-following behavior to implement a state transition, it must be sufficiently constrained so that the start state determines the termination state — in other words, the path's constraints must limit the robot's motion to one degree of freedom. A constraint $\mathbf{y} = \mathbf{y}^*$ and action $\mathbf{u}^\beta$ define a 1-dof path-following behavior if $\mathbf{u}^\beta$ maintains the constraint (according to the static action model) but does not maintain the con-

straint when combined with any other action (according to the dynamic action model). Moreover, since the other actions all have an effect on the feature vector $\mathbf{y}$ while action $\mathbf{u}^\beta$ is being taken, they can all be used as error-correcting actions in the closed-loop path-following behavior. For example, suppose the robot is facing parallel to a wall at a distance $y = y^*$. The constraint $y = y^*$ and action $\mathbf{u}^1$ (advance) define a 1-dof path-following behavior: Advancing maintains the constraint, but turning and advancing together do not. Turning can be used to provide error correction.

The interface between the continuous world and the discrete learning agent works as follows. The discrete learning agent is given the list B_q of currently applicable 1-dof path-following behaviors. It selects one which then executes until it ends. At this point, the process repeats. If, at a given state, no 1-dof path-following behaviors are applicable, then the discrete learning agent is given a set of underconstrained path-following behaviors. If no path-following behaviors are applicable, it is given a set of hill-climbing behaviors. If no hill-climbing behaviors are applicable, then it is given a single behavior that randomly wanders until another behavior becomes applicable. With this definition of the interface, the discrete learning agent's actions are kept as deterministic as possible, simplifying the task of inferring the structure of the finite-state automaton that the interface defines.

We are currently working to implement a discrete learning agent based on the map-learning approach used by the NX robot (Kuipers & Byun 1988, 1991; Kuipers et al. 1993) which provided the original motivation for this work. For now, a stochastic exploration behavior is used to demonstrate the interface. It operates by randomly choosing an applicable hill-climbing or path-following behavior and executing it until it is done or no longer applicable, or a behavior becomes applicable that previously was not. An example interaction with the interface is demonstrated in Figure 11 for a simple environment.

Related Work

Inferring the structure of finite-state worlds

The task of inferring the structure of a finite-state environment is the task of finding a finite-state automaton that accurately captures the input-output behavior of the environment. In the case that the learning agent is passively given examples of the environment's input/output behavior, it has been shown that finding the smallest automaton consistent with the behavior is NP-complete (Angluin 1978, Gold, 1978). With active learning, in which the agent actively chooses its actions, the problem becomes tractable. Kuipers (1978) describes the TOUR model, a method for understanding discrete spatial worlds based on a theory of cognitive maps. Angluin (1987) gives a polynomial-time algorithm using active experimentation and passively received counterexamples. Rivest & Schapire (1993) improve on Angluin's algorithm and give a version that does not require the reset operation (returning to the start state after each experiment).

Dean et al. (1992) have extended Rivest and Schapire's theory to handle stochastic FSA's. They assume that actions are deterministic but that the output function mapping states to senses is probabilistic. Their trick is "going in circles" until the uncertainty washes out. Dean, Basye, and Kaelbling (1993) give a good review of learning techniques for a variety of stochastic automata. Drescher's schema mechanism (1991) employs a statistical learning method called marginal attribution. The set of learned schemas fills the role that δ and γ play in the FSA model. Schemas emphasize sensory effects of actions rather than state transitions and are ideal for representing partial knowledge in stochastic worlds.

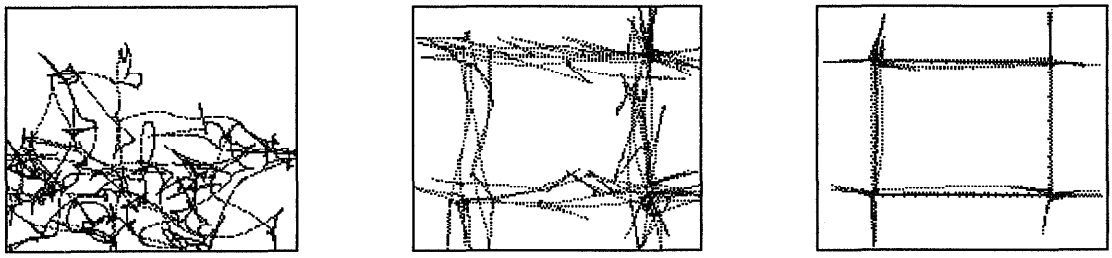
Inferring the structure of continuous worlds

Applying the previous learning methods to the real world or a continuous simulation of it requires an abstraction from a continuous environment to a discrete representation. Kuipers and Byun (1988, 1991) demonstrate an engineered solution to the continuous-to-discrete abstraction problem for the NX robot. NX's *distinctive places* correspond to discrete states and its *local control strategies* correspond to state transitions. These constructs have to be manually redesigned in order to apply to a robot with a different sensorimotor apparatus. Kortenkamp & Weymouth (1994) have engineered a similar solution on a physical robot that exploits visual as well as sonar information. Lin and Hanson (1993) are using reinforcement learning to teach a robot a predefined set of local control strategies such as hall following. A difference between their approach and ours is that our robot must discover and then learn path-following behaviors on its own. It has no concept of "hall" or "hall-following."

To summarize our position, we are developing methods for *learning* from *tabula rasa* the interface that is *engineered* by Kuipers & Byun (1988) and Kortenkamp & Weymouth (1993), and is being *taught* by Lin and Hanson (1993).

Results and Conclusions

The method for diagnosing primitive actions has been successfully applied to a variety of sensory systems: distance sensors, a 5x5 grid of photoreceptors; and to a variety of motor apparatuses: turn-and-advance, tank, translate, turn-advance-slide (having 3 degrees of freedom: rotation, forward-backward, and left-right). The method for discovering local state variables has been successfully applied to the ring of 16 distance sensors. The learning of action models and path-following behaviors has been demonstrated on the simulated robot



(a) (b) (c)

Figure 11: Exploring a simple world at three levels of competence. (a) The robot wanders randomly. (b) The robot explores by randomly choosing applicable hill-climbing and open-loop path-following behaviors based on the static action model. (c) The robot explores by randomly choosing applicable hill-climbing and closed-loop path-following behaviors based on the dynamic action model.

with distance sensors and turn-and-advance motor apparatus.

We have presented a method for learning a cognitive map of a continuous world in the absence of *a priori* knowledge of the learning agent's sensorimotor apparatus or of the structure of its world. By choosing the finite-state automaton as the target abstraction, we inherit a powerful set of methods for inferring the structure of a world. In the process of developing this abstraction, we have contributed methods for modeling a motor apparatus, for learning useful features, and for characterizing the effects of actions on features in two ways: The static action model captures first-order effects useful for defining hill-climbing behaviors and for deciding when an action leaves a feature invariant. The dynamic action model captures second-order effects useful for error correction in robust path-following control laws.

References

Angluin, D. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75:87–106.

Dean, T.; Basye, K.; and Kaelbling, L. 1993. Uncertainty in graph-based map learning. In Connell, J. H., and Mahadevan, S., eds., *Robot Learning.* Boston: Kluwer Academic Publishers. 171–192.

Dean, T.; Angluin, D.; Basye, K.; Engelson, S.; Kaelbling, L.; Kokkevis, E.; and Maron, O. 1992. Inferring finite automata with stochastic output functions and an application to map learning. In *Proceedings, Tenth National Conference on Artificial Intelligence,* 208–214. San Jose, CA: AAAI Press/MIT Press.

Drescher, G. L. 1991. *Made-Up Minds: A Constructivist Approach to Artificial Intelligence.* Cambridge, MA: MIT Press.

Kortenkamp, D., and Weymouth, T. 1994. Topological mapping for mobile robots using a combination of sonar and vision sensing. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94).*

Kuipers, B. J., and Byun, Y.-T. 1988. A robust, qualitative method for robot spatial learning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-88),* 774–779.

Kuipers, B. J., and Byun, Y.-T. 1991. A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations. *Journal of Robotics and Autonomous Systems* 8:47–63.

Kuipers, B.; Froom, R.; Lee, W.-Y.; and Pierce, D. 1993. The semantic hierarchy in robot learning. In Connell, J. H., and Mahadevan, S., eds., *Robot Learning.* Boston: Kluwer Academic Publishers. 141–170.

Kuipers, B. J. 1978. Modeling spatial knowledge. *Cognitive Science* 2:129–153.

Kuo, B. C. 1982. *Automatic Control Systems.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 4 edition.

Lin, L.-J., and Hanson, S. J. 1993. On-line learning for indoor navigation: Preliminary results with RatBot. In *NIPS93 Robot Learning Workshop.*

Mardia, K. V.; Kent, J. T.; and Bibby, J. M. 1979. *Multivariate Analysis.* New York: Academic Press.

Michie, D., and Chambers, R. A. 1968. BOXES: An experiment in adaptive control. In Dale, E., and Michie, D., eds., *Machine Intelligence 2.* Edinburgh: Oliver and Boyd. 137–152.

Pierce, D. M. 1991. Learning a set of primitive actions with an uninterpreted sensorimotor apparatus. In Birnbaum, L. A., and Collins, G. C., eds., *Machine Learning: Proceedings of the Eighth International Workshop (ML91),* 338–342. San Mateo, CA: Morgan Kaufmann Publishers, Inc.

Rivest, R. L., and Schapire, R. E. 1993. Inference of finite automata using homing sequences. *Information and Computation* 103(2):299–347.