Developing navigation behavior through self-organizing distinctive state abstraction

To appear in *Connection Science*, 18:2 (2006).

Jefferson Provost*, Benjamin J. Kuipers, and Risto Miikkulainen
Artificial Intelligence Lab
The University of Texas at Austin
1 University Station C0500
Austin, TX 78712 USA
{jp,kuipers,risto}@cs.utexas.edu

Abstract

A major challenge in reinforcement learning research is to extend methods that have worked well on discrete, short-range, low-dimensional problems to continuous, high-diameter, high-dimensional problems, such as robot navigation using high-resolution sensors. Self-Organizing Distinctive-state Abstraction (SODA) is a new, generic method by which a robot in a continuous world can better learn to navigate by learning a set of high-level features and building temporally-extended actions to carry it between distinctive states based on those features. A SODA agent first uses a self-organizing feature map to develop a set of high-level perceptual features while exploring the environment with primitive, local actions. The agent then builds a set of high-level actions composed of generic trajectory-following and hill-climbing control laws that carry it between the states at local maxima of feature activations. In an experiment on a simulated robot navigation task, the SODA agent learns to perform a task requiring 300 small-scale, local actions using as few as 9 new, temporally-extended actions, significantly improving learning time over navigating with the local actions.

Keywords: Developmental Robotics, Self-organization, Reinforcement learning, Robot navigation.

1 Introduction

Modern robots are endowed with rich, high-dimensional sensory systems, providing measurements of a continuous environment. In addition, many important

*Corresponding author.

real-world robotic tasks have *high diameter*, that is, their solutions require a large number of primitive actions by the robot, for example, navigating to distant locations using primitive motor control commands. Reinforcement learning (RL) methods show promise for automatic learning of robot behavior, but extending these methods to high-dimensional, continuous, high-diameter problems remains a major challenge. Thus, the success of RL on real-world tasks still depends on human analysis of the robot, environment, and task to provide a useful set of perceptual features and an appropriate decomposition of the task into subtasks. For a learning agent to be truly autonomous, however, the 'hard part' of learning ultimately needs to be performed by the agent, rather than the human engineer.

Self-Organizing Distinctive-state Abstraction (SODA) is a new method for automatic discovery of high-level perceptual features and large-scale actions for learning to navigate in continuous environments. This paper presents a first implementation of the algorithm in which the features are learned and the high-level action policies are specified by generic control laws based on those features. The control laws encode no specific information about the robot's sensorimotor system, environment, or task, yet the new actions significantly speed learning on a high-diameter navigation task. The implementation is therefore a significant step towards a generic, autonomous learning agent for robot navigation.

Given high-dimensional, continuous-valued sensory input, and continuous motor output, SODA works as follows:

1. Learn a set of high-level perceptual features that define *distinctive states* in the environment by training a Self-organizing Feature Map Kohonen (1995) while exploring the environment with primitive actions.

2. Construct a set of high-level actions that carry the robot from one distinctive state to another using generic trajectory-following and hill-climbing control laws.

3. Learn policies for high-diameter tasks through reinforcement learning in the abstracted space of high-level features and actions.

In order for the control-laws to work, SODA relies on the assumption that the world is continuous, i.e. small actions generally induce small changes in sensor values. This assumption is usually valid in robotic navigation.

Spatial navigation is a foundational domain of common-sense knowledge, and a specific goal of this work is investigation of how an agent can learn to navigate. As a common-sense knowledge domain, spatial navigation is involved in the solution of many day-to-day problems by humans, and animals, and is essential for mobile robotics, but SODA alone is not intended to learn task decomposition for complex 'meta-navigational' tasks (e.g making a sandwich), although spatial navigation may be essential to completing such tasks.

This paper is organized as follows: Section 2 describes the foundations of SODA in self-organizing maps, robot navigation, reinforcement learning and temporal abstraction. Section 3 describes the SODA method in detail. Section 4 describes a experiment with the method that shows a large reduction in task diameter using the learned high-level actions, and Section 5 discusses the results of the experiment and describes ongoing and future work on SODA.

2 Background and Related Work

SODA is one of a number of efforts in the field of *Developmental Robotics*, several more of which are described in this issue Blank and Meeden (2006); Schlesinger (2006); Kuipers et al. (2006); Schmidhuber (2006); Stronger and Stone (2006); Oudeyer and Kaplan (2006); Gold and Scassellati (2006); Olsson et al. (2006). Developmental robotics comprises a broad class of approaches characterized by self-organization and autonomous development of robot behavior. SODA combines Self-Organizing Feature Maps, the action abstraction from the Spatial Semantic Hierarchy Kuipers (2000), and reinforcement learning with temporal abstraction. These ideas are described in more detail below.

2.1 Self-Organizing Feature Maps

SODA constructs perceptual features by using the sensory input as training data for an unsupervised self-organizing feature map (SOM) Kohonen (1995) that learns a set of *sensory prototypes* to represent its sensory experience.

A standard SOM consists of a set of units arranged in a lattice. The SOM takes a continuous-valued vector $\mathbf{x}$ as input and returns one of its units as the output. Each unit i has a weight vector $\mathbf{w}_i$ of the same dimension as the input. On the presentation of an input, each weight vector is compared with the input using the Euclidean distance and a *winner* is selected as $\arg\min_i \|\mathbf{x} - \mathbf{w}_i\|$. The winner and its lattice neighbors are then adjusted toward $\mathbf{x}$.

The SOM has been used previously for learning perceptual features and state representations in general robotics Martinetz et al. (1990); Duckett and Nehmzow (2000); Nehmzow and Smithers (1991); Provost et al. (2001); Chaput et al. (2003), as well as for learning state abstraction in RL Smith (2002); Toussaint (2004). All these methods use the SOM as a clustering or vector quantization method, ignoring the variation between the inputs that produce the same winner. The SOM, however, can provide both a coarse-grained discretization of its input space, and a set of continuous features (or 'activations') defined in terms of the distance of the input from each unit. These properties divide the continuous state space into a set of neighborhoods defined by their winning unit, each containing a stable fixed point at the local maximum of the winner's activation. SODA uses this feature to construct actions, as described in the next section.

The implementation in this paper uses a variant on the standard SOM algorithm called the Growing Neural Gas (GNG) Fritzke (1995), that begins with a small set of units and inserts new units incrementally to minimize distortion error. The GNG is able to continue learning indefinitely, adapting to changing input distributions. This property makes the GNG especially suitable for robot learning, since a robot experiences its world sequentially, and may experience entirely new regions of the input space after an indeterminate period of exploration. In addition, the GNG is not constrained by the pre-specified topology of the SOM lattice. It learns its own topology in response to experience with the domain. An abbreviated description of the GNG algorithm follows. The reader should refer to the GNG reference Fritzke (1995) for details.

- Begin with two units, randomly placed in the input space.
- Upon presentation of an input vector $\mathbf{x}$:
  - Select the two closest units to $\mathbf{x}$, denoted as $\mathbf{q}_1$ and $\mathbf{q}_2$, if these units are not already connected in the topology, add a connection between them.
  - Move $\mathbf{q}_1$ toward $\mathbf{x}$ by a fraction of the distance between them. Move all the topological neighbors of $\mathbf{q}_1$ toward $\mathbf{x}$ by a smaller fraction of the respective distances.
  - Add the squared error $||\mathbf{x} - \mathbf{q}_1||^2$ to an accumulator associated with $\mathbf{q}_1$.
  - Decay the accumulated error of all nodes by a fraction of their values.
- Periodically, after every λ inputs, add a unit by selecting the existing unit with the greatest accumulated error and the unit among its topological neighbors with the most accumuated error; add new unit whose weight vector is the average of the two selected units. Connect the new unit to the two selected units, and select the original connection between the two.[1]

The original GNG algorithm adds nodes until the network reached some fixed criterion such as a maximum number of units. SODA uses a slightly modified algorithm, *Equilibrium-GNG*, that has no fixed stopping criterion, but rather only adds nodes if the average accumulated error over the network is greater than a given threshold. Given a stationary input distribution, an Equilibrium-GNG will grow until reaching an equilibrium between the rate of accumulation of error (per unit) and the rate of decay. If the distribution changes to cover a new part of input space, the error accumulated in the units nearest the new inputs will increase above the threshold, and the network will grow again.

2.2 Abstract Actions in the Spatial Semantic Hierarchy

To create high-level actions, the agent uses the abstraction from the *control level* to the *causal level* of the *Spatial Semantic Hierarchy* (SSH), a theory of representation of large-scale space Kuipers (2000). At the control level, there are two kinds of control laws: *Trajectory-following* (TF) control laws carry the robot from one distinctive state into the neighborhood of another, while *Hill-climbing*

[1]The algorithm also contains provisions for deleting units and connections. For the details the reader is referred to the original GNG paper.

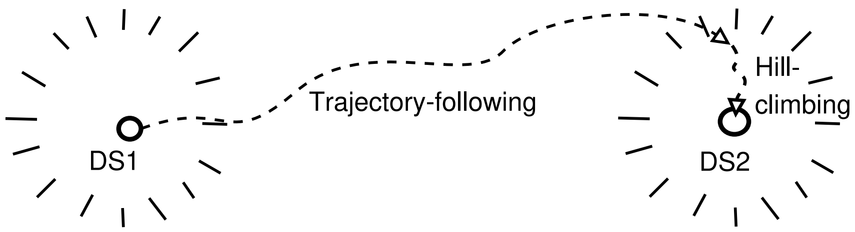

Figure 1: **High-level Actions.** The agent travels from one distinctive state to another using an action with two parts: first a *trajectory-following* (TF) controller drives the robot into the neighborhood of a new sensory prototype, then a *hill-climbing* (HC) controller takes the robot to the local state that best matches that prototype.

(HC) control laws carry the robot to a distinctive state within the neighborhood, by climbing the gradient of some 'distinctiveness measure' on the sensory input (See Figure 1). The causal level defines a set of high-level actions each consisting of a TF/HC control-law pair that carries the robot from one distinctive state to another. The benefit of these actions is twofold: the TF component gives the actions extent, allowing the agent to move through its environment in large steps, while the HC component reduces positional uncertainty by bringing the robot to a fixed state at the end of each action.

2.3 Reinforcement Learning

SODA uses reinforcement learning to learn a policy over high-level, temporally abstract actions. The next sections describe the RL algorithm used in the experiments and the framework SODA uses for temporal abstraction in RL.

2.3.1 The Sarsa(λ) Algorithm

The feature and action construction methods in the proposed work are intended to be agnostic with respect to the specific reinforcement learning algorithm used. The experiment in Section 4 uses Sarsa(λ), described by Sutton and Barto (1998). 'Sarsa' is an acronym for State, Action, Reward, State, Action; each Sarsa update uses the states and actions from time t and the reward, state, and action from time $t+1$, usually denoted by the tuple: $\langle s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}\rangle$. In the simplest form, Sarsa modifies the state-value estimate $Q(s,a)$ as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \tag{1}$$

This rule updates the current Q value by a fraction of the reward after action a_t plus the *temporal difference* of the discounted reward predicted from the next state-action pair, $Q(s_{t+1}, a_{t+1})$ and the current estimate of $Q(s_t, a_t)$. The parameter $0 < \alpha < 1$ is a learning rate that controls how much of this value is used. The parameter $0 < \gamma \leq 1$ is the discount factor. For episodic tasks, that ultimately reach a terminating state, as in Section 4, $\gamma = 1$ is used, allowing $Q(s, a)$ to approach an estimate of the remaining reward for the episode.

The learning rule above only 'backs-up' the reward prediction by one time-step on each step of the agent's policy. With high-diameter tasks, this can mean many episodes of interaction are required to propagate the reward prediction to all states. For faster learning, Sarsa(λ) performs multiple-step backups by keeping track of how long it has been since a state-action pair $\langle s, a\rangle$, has been visited using an *eligibility trace*, $e(s, a)$. The higher the value of $e(s, a)$, the more recently the agent experienced $\langle s, a\rangle$. The eligibility trace is then used to update the value estimate for all recently-visited state-action pairs on each time step. The updates operate as follows:

Each time a step is taken, each eligibility trace is decayed according to a parameter $0 \leq \lambda < 1$:

$$\forall s, a : e(s, a) \leftarrow \lambda e(s, a) \tag{2}$$

Then the eligibility trace for the last state and action is updated:

$$e(s_t, a_t) \leftarrow 1 \tag{3}$$

Finally the Q table is updated according to the eligibility trace:

$$\forall s, a : Q(s, a) \leftarrow Q(s, a) + e(s, a)\, \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \tag{4}$$

Note that when $\lambda = 0$ this update rule reduces to the one step update, in Equation (1).

2.3.2 Temporal Abstraction, SMDPs, and Options

The most widely-used formalism for representing hierarchical task decomposition in reinforcement learning is the semi-Markov decision process (SMDP). SMDPs extend MDPs by allowing actions with variable temporal extent, that may themselves be implemented with RL, executed as 'subroutines.' Such processes are 'semi-Markov' because the choice of primitive actions (at the lowest level of the decomposition) depends not only on the environmental state, but also on the internal state of the agent, as manifest in the choice of higher-level actions.

One formalism for describing SMDPs is *Options* Precup (2000); Sutton et al. (1999). Given an agent operating in an MDP with state set $\mathcal{S}$ and action set $\mathcal{A}$, the Options framework defines temporally-extended actions ('options') as tuples $\langle \mathcal{I}, \pi, \beta \rangle$, where the *input set*, $\mathcal{I} \subseteq \mathcal{S}$, is the set of states where the option may be executed, the policy, $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$, determines the probability of selecting a particular action in a particular state while the option is executing, and the *termination condition*, $\beta : \mathcal{S} \rightarrow [0, 1]$, indicates the probability that the option will terminate in any particular state. For uniformity, primitive actions are formalized as options so each SMDP is defined over options only. Each primitive action $a \in \mathcal{A}$ can be seen as an option whose input set $\mathcal{I}$ is the set of states where the action is applicable, whose policy π chooses a always, and whose termination condition β always returns 1.

There has been considerable research into automatically discovering options or other kinds of high-level actions for reinforcement learning Digney (1998); McGovern and Barto (2001); Ryan (2002); Hengst (2002). These methods all assume that a continuous-to-discrete abstraction already exists, and they search for higher-level temporal abstractions in the (already abstracted) discrete Markov decision process. SODA assumes a continuous state space and discovers a continuous-to-discrete abstraction of both perceptual and action space that results in temporally extended, abstract actions, using the gradients of continuous features to define the high-level actions. The above methods and SODA are potentially complementary. In very large problems it is likely that multiple levels of abstraction will be needed. In principle, one should be able to use the methods above to perform additional temporal abstraction on top of the continuous-to-discrete abstraction SODA provides, but this has not been implemented yet.

3 Learning Method and Representation

The SODA algorithm can be characterized formally as follows. Given

- a robot with a sensory system providing experience as a sequence of N-dimensional, continuous *sensory vectors* $\mathbf{y}_1, \mathbf{y}_2, \ldots$, where every $\mathbf{y}_t \in \mathbb{R}^N$,
- a continuous, M-dimensional motor system that accepts from the agent a sequence of *motor vectors* $\mathbf{u}_1, \mathbf{u}_2, \ldots$ where every $\mathbf{u}_t \in \mathbb{R}^M$,
- a continuous world that induces an almost-everywhere-continuous sensory stream, characterized by one or more large regions in which small actions induce small changes in sensor values, separated by discontinuous 'thresholds', across which sensor values may make large jumps.
- and a scalar *reward signal*, $r_1, r_2, \ldots$, that defines a high-diameter task (that is, maximizing the expected value of r requires a long sequence of motor vectors $\mathbf{u}_t, \ldots \mathbf{u}_{t+k}$).

the algorithm performs these steps:

1. Define a set of discrete, local primitive actions $\mathcal{A}^0$: First, using methods developed by Pierce and Kuipers (1997), learn an *abstract motor interface*, a basis set of orthogonal motor vectors $\mathcal{U} = \{\mathbf{u}^0, \mathbf{u}^1, \ldots \mathbf{u}^{n-1}\}$ spanning the set of motor vectors $\mathbf{u}_t$ possible for the robot. Then define $\mathcal{A}^0$ to be the set of $2n$ discrete actions derived from $\mathcal{U}$ by setting the motor signal $\mathbf{u}_t$ to the value $\mathbf{u}^i$ or $-\mathbf{u}^i$ for a short time period Δt.
2. Learn a set $\mathcal{F}$ of high-level perceptual features: Exploring the environment with a random sequence of $\mathcal{A}^0$ actions, train a SOM with the sensor signal $\mathbf{y}_t$ to converge to a set of high-level features of the environment. For each

Trajectory-follow on a_i^0:
    $f_w \leftarrow \arg\max_{f\in\mathcal{F}} f(\mathbf{y})$
    while $f_w = \arg\max_{f\in\mathcal{F}} f(\mathbf{y})$:
        execute action a_i^0
$f_{ds} \leftarrow \arg\max_{f\in\mathcal{F}} f(\mathbf{y})$
Hill-climb on f_{ds}:
    while $\max_j \Delta^j f_{ds} > 0$:
        $w \leftarrow \arg\max_j \Delta^j f_{ds}$
        execute action a_w^0

Table 1: Trajectory-following/Hill-climbing Pseudo-code. $\Delta^j f_i$ is the gradient of feature f_i with respect to primitive action a_j^0.

weight vector $\mathbf{w}_i$ in the SOM, there is an *activation function* $f_i \in \mathcal{F}$ such that:

$$f_i(\mathbf{y}) = \frac{\hat{f}_i(\mathbf{y})}{\sum_{j=0}^{|\mathcal{F}|} \hat{f}_j(\mathbf{y})} \text{ where } \hat{f}_j(\mathbf{y}) = \frac{1}{\|\mathbf{y} - \mathbf{w}_j\|^z} \tag{5}$$

These equations define an activation for each unit that varies with the inverse of the Euclidean distance of that unit from the input vector. The activation across all units is normalized to sum to unity. The *response exponent* z is a parameter that controls the 'width' or 'focus' of the response, with higher z allocating more activation to the winner.

3. Define a hill-climbing (HC) control law for each $f_i \in \mathcal{F}$: For each f_i in the context where $\arg\max_{f\in\mathcal{F}} = f_i$, estimate its gradient with respect to each primitive action a_j^0 in $\mathcal{A}^0$, denoted $\Delta^j f_i$. This estimate is currently found by applying each action, sampling the feature value, and reversing the action. The HC control-law selects the action with the highest gradient, terminating when all the gradients are negative (See Table 1). The state of the agent when hill-climbing terminates is defined to be a *distinctive state* (dstate).

4. For each distinctive state defined by $f_i \in \mathcal{F}$, define trajectory-following control laws that take the agent to a state where a different feature $f_j \in \mathcal{F}$ is dominant. Currently, the algorithm proposes one such TF control law for each primitive action $a_k^0 \in \mathcal{A}^0$, consisting of simply repeating a_k^0 until the dominant feature f_i is replaced by another feature f_j (See Table 1).

5. Define a set of higher-level actions $\mathcal{A}^1$ where each $a_m^1 \in \mathcal{A}^1$ consists of executing one TF control law, and then hill-climbing on the resulting dominant feature f_j. At this point the agent has abstracted its continuous state and action space into a discrete Semi-Markov Decision Process (SMDP) with one state for each feature in $\mathcal{F}$, and the large-scale actions in $\mathcal{A}^1$.

(a) Abstract Motor Interface		
	$\mathbf{u}^0$	$\mathbf{u}^1$
drive	250 mm/sec	0 mm/sec
turn	$0°$/sec	$90°$/sec

(b) Primitive Actions $\mathcal{A}^0$		
action	$\mathbf{u}$	step
a_0^0	$\mathbf{u}^0$	25mm
a_1^0	$-\mathbf{u}^0$	-25mm
a_2^0	$\mathbf{u}^1$	$9°$
a_3^0	$-\mathbf{u}^1$	$-9°$

Table 2: Abstract Motor Interface and Primitive Actions

3.1 Policy Learning with High-level Actions

The new, feature-action space defined by $\mathcal{F}$ and $\mathcal{A}^1$ forms a semi-Markov decision process (SMDP). As described in Section 2.3.2, high-level actions can be described as *options*. Each action a_i^1 is an option $o_i = \langle \mathcal{I}_i, \pi_i, \beta_i \rangle$, where the input set, $\mathcal{I}_i$, is the set of states where that action's initial trajectory-following control law is applicable[2], π_i embodies the combination of trajectory-following and hill-climbing control-laws described above, and β_i returns true in the states where hill-climbing control-laws terminate, as described in Table 1.

To learn a policy for selecting options to execute, the Sarsa(λ) algorithm presented in Section 2.3.1 must be slightly modified to accommodate the new options Precup (2000). Assume an option o_t is executed at time t, and takes τ steps to complete. Define $\rho_{t+\tau}$ as the cumulative, discounted reward over the duration of the option:

$$\rho_{t+\tau} = \sum_{i=0}^{\tau-1} \gamma^i r_{t+i+1} \tag{6}$$

Using this value, the one-step update rule (Equation 1) is modified as follows:

$$Q(s_t, o_t) \leftarrow Q(s_t, o_t) + \alpha \left[\rho_{t+\tau} + \gamma^\tau Q(s_{t+\tau}, o_{t+\tau}) - Q(s_t, o_t) \right] \tag{7}$$

For Sarsa(λ), the multi-step update rule (Equation 4) is modified analogously. The eligibility-trace now tracks state-option pairs, $e(s, o)$, and is updated upon option selection.

Note that for one-step options, these modifications reduce to the original Sarsa(λ) equations. Also, obviously, when $\gamma = 1$, as is often true in episodic tasks (and is true in the experiment in Section 4), the reward ρ is simply the total reward accumulated over the course of executing the option, and the update rule is again essentially the same as the original, if the reward r in the original rule is taken to mean the reward accumulated since the last action.

4 Experiment

This section presents an experiment demonstrating that SODA significantly reduces the task diameter in a robot navigation task.

[2]In the current case, the trajectory-following control law is open-loop and always applicable, but this formulation admits more sophisticated control-laws with applicability conditions.

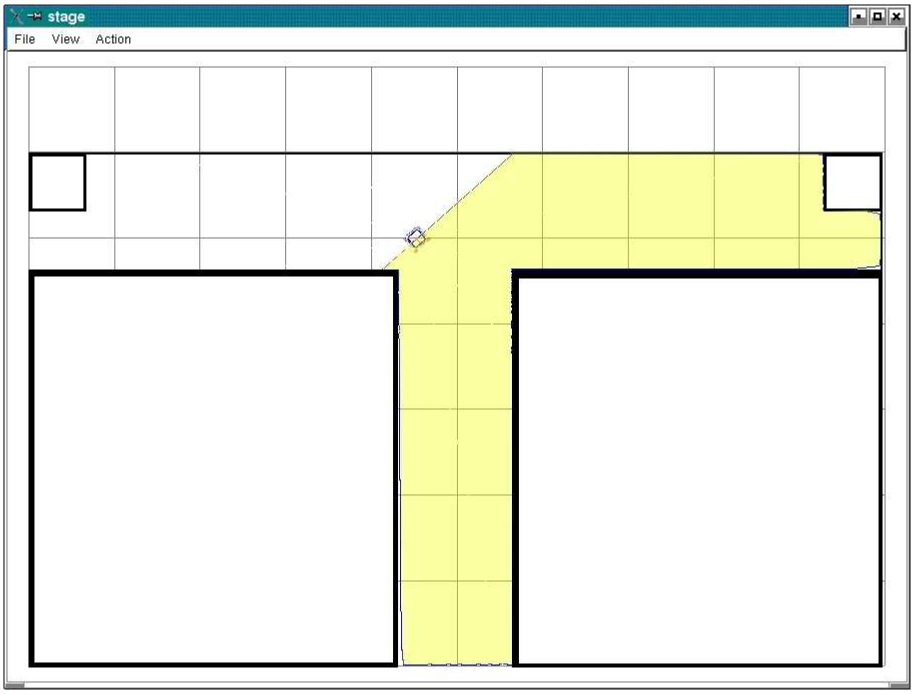

Figure 2: **Simulated Robot Environment**. A screen-shot of the Stage robot simulator with the simulated robot and experimental environment. The robot has a drive-and-turn base, and a laser rangefinder. The agent must learn to travel from the left end of the upper hallway to the end of the lower hallway. The shading indicates the area swept out by the laser rangefinder.

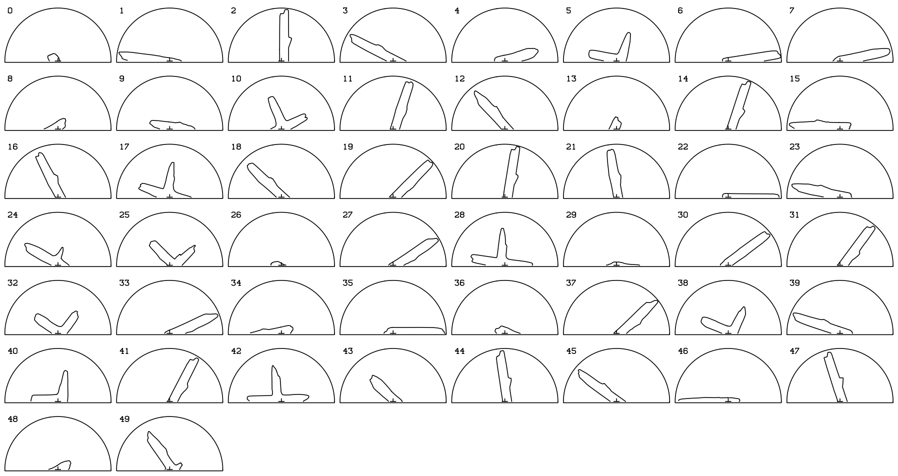

Figure 3: **Example Learned Perceptual Prototypes.** The agent's self-organizing feature map learns a set of perceptual prototypes that are used to define perceptually distinctive states in the environment. This figure shows the set of features learned from one run. Each feature is a prototypical laser rangefinder image plotted radially, with the robot at the origin.

In the experiment, a SODA agent is instantiated in a simulated robot using the Stage simulator Gerkey et al. (2003). The environment is a T-shaped room or maze, shown in Figure 2, measuring 10 000 mm $\times$ 6000 mm. The robot has a simple drive/turn motor system, and the agent is assumed to have already identified the two dimensional abstract motor interface $\mathcal{U}$ shown in Table 2(a). Stage does not simulate acceleration, so velocity changes instantaneously. The agent's input $\mathbf{y}$ consists of the response from a single simulated SICK LMS laser rangefinder, providing 180 range readings over the forward semi-circle, with a maximum range of 8000 mm and a resolution of 10 mm. The simulator accepts motor commands and provides sensations at 10 Hz (simulator time). The abstract motor interface is quantized at this rate to provide four $\mathcal{A}^0$ actions consisting of positive and negative steps along each axis of motor control. These actions are described in Table 2(b).

The agent's task is to drive from the left end of the upper hallway to the bottom of the center hallway. The task terminates when the robot reaches within 500 mm of the goal (a point 500 mm from the end of the lower corridor), or times out after 10 000 simulator steps. The reward on each non-terminal step is -1 unless the robot collides with a wall, in which case it is -5. The reward upon successful termination is 0. This scheme rewards finding the shortest path to the goal while not bumping into walls.

To learn the initial set of perceptual features, the agent trains a Growing Neural Gas network with its sensory input over a 500 000 step random walk through the environment (the equivalent of 50 task learning episodes). The GNG parameters are $\lambda = 2000$, $\alpha = 0.05$, $\beta = 0.0005$, $e_b = 0.05$, $e_n = 0.0006$, $a_{max} = 100$ Fritzke (1995). The GNG was configured to grow only if the average cumulative distortion error across all units was greater than 37 500. A typical set of learned features is shown in Figure 3. This set comprises a wide variety of prototypical views of the environment. Feature activations are computed according to Equation (5), with response exponent $z = 4$.

The agent has four simple, open-loop trajectory-following control laws, one for each $\mathcal{A}^0$ action, as described in step 4 of the algorithm. The agent learns its high-level control policy using episodic, tabular SARSA(λ) reinforcement learning Sutton and Barto (1998), using the SOM winner as the state. SARSA parameters[3] used are $\lambda = 0.9$, $\alpha = 0.2$, $\gamma = 1.0$. All Q values are initialized optimistically to 0.0, and actions are selected greedily with ties broken randomly. These parameter values for both the GNG and SARSA were found to be reasonable after some manual experimentation, but, in keeping with the spirit of autonomous learning, no exhaustive parameter search was performed. Learning trials consisted of 12 runs of 5000 episodes in each of two experimental conditions, the first with only $\mathcal{A}^0$ actions, the second with only $\mathcal{A}^1$ actions. This experiment used 10 different trained SOMs for a total of 240 runs.

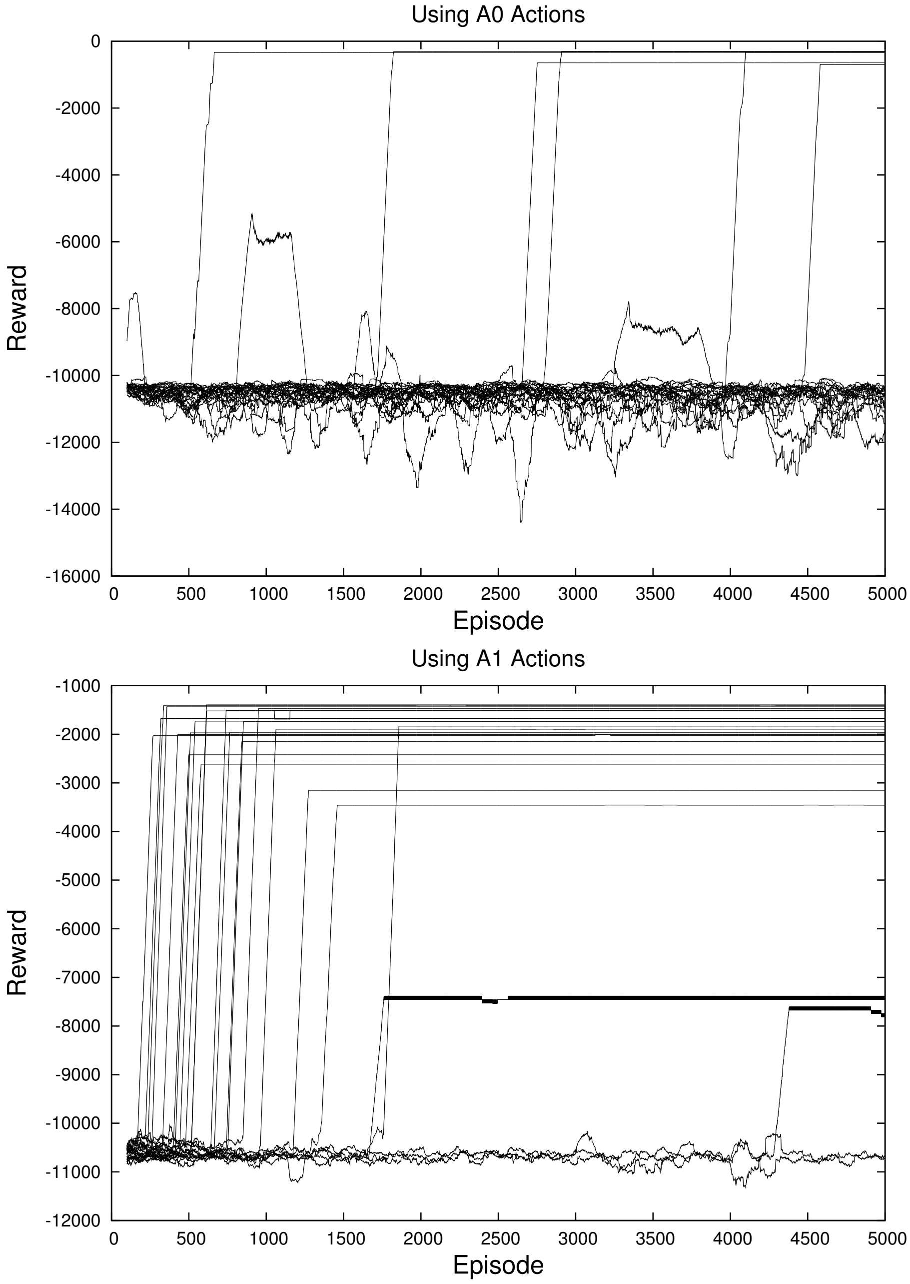

Figure 4: **Learning Performance** These plots show the reward per episode for a selection of individual agents in the navigation experiment. The top plot shows performance for the agents using $\mathcal{A}^0$ actions, the bottom plot for agents using $\mathcal{A}^1$ actions. Every fifth agent is shown for a total of 24 curves per figure. The curves are smoothed with a 100 episode moving window.

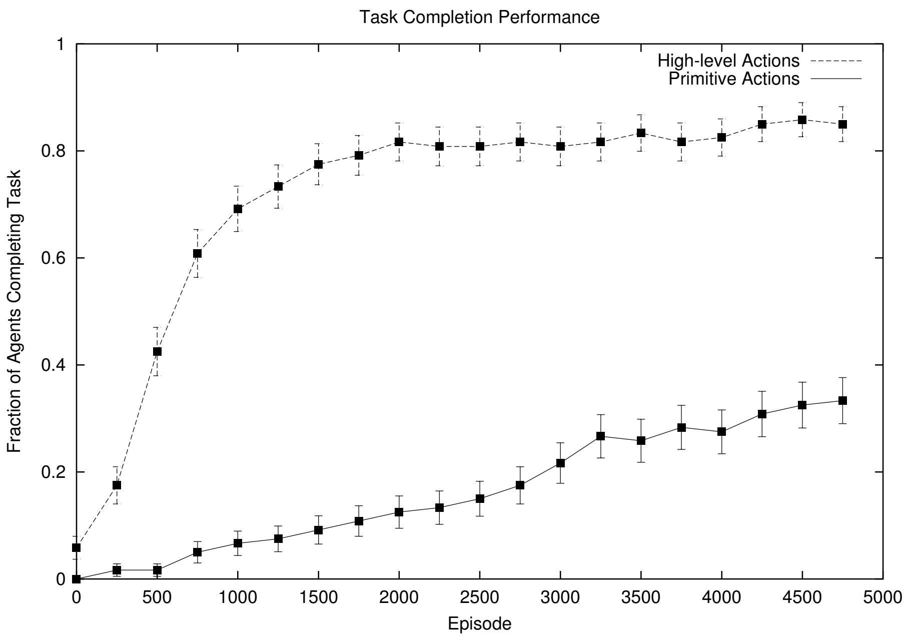

Figure 5: **Average Task Completion** Comparison of the fraction of agents completing the task within each 10 000 time-step episode using primitive ($\mathcal{A}^0$) actions vs. using high-level ($\mathcal{A}^1$) actions. Each curve is an average of 12 runs using each of 10 different learned feature sets. Error bars indicate +/- one standard error. Agents using the $\mathcal{A}^1$ actions learn the task much faster.

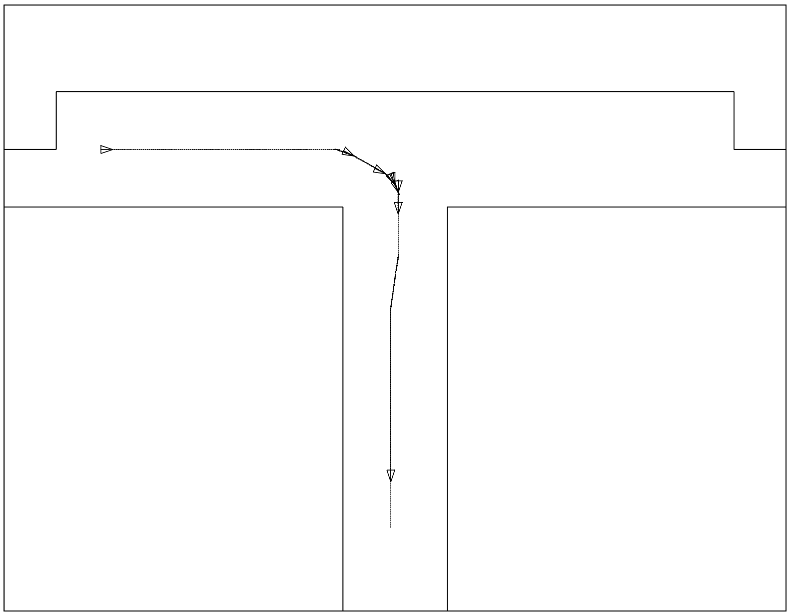

Figure 6: **Navigation using Learned Abstraction**. A schematic view of the environment shown in Figure 2, showing an example episode after the agent has learned the task using the $\mathcal{A}^1$ actions. The triangles indicate the state of the robot at the start of each $\mathcal{A}^1$ action. The sequence of winning features corresponding to these states is [44, 10, 25, 32, 38, 5, 40, 44, 13] (See Figure 7). The narrow line indicates the sequence of $\mathcal{A}^0$ actions used by the $\mathcal{A}^1$ actions. Note that in two cases the $\mathcal{A}^1$ actions essentially abstract the concept, 'drive down the hall to the next decision point.' In the south corridor, the path turns slightly to the west at the transition from trajectory-following to hill-climbing. Navigating to the goal requires only 9 $\mathcal{A}^1$ actions, instead of hundreds of $\mathcal{A}^0$ actions, in other words, task diameter is vastly reduced.

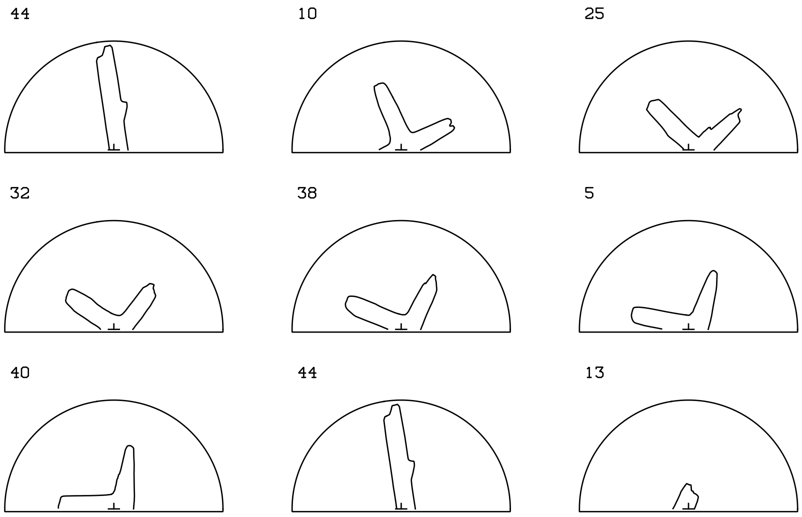

Figure 7: **Features for Distinctive States** The perceptual features for the distinctive states used in the solution shown in Figure 6, in the order they were traversed in the solution. (Read left-to-right, top-to-bottom.)

4.1 Results

Figure 4 demonstrates that agents learn more often to complete the task in the time allotted using $\mathcal{A}^1$ actions than using $\mathcal{A}^0$ actions[4], and those that do finish generally do so sooner. For readability, Figure 4 shows the performance of only a fraction of trials. Figure 5 shows average performance over all trials, and confirms that the trend in Figure 4 holds.

Figure 6 shows a typical task solution learned using high-level actions. Figure 7 shows the sequence of features that define the distinctive states in that solution, in the order that they are encountered in the solution. These figures show that the $\mathcal{A}^1$ actions are highly abstracted compared to the $\mathcal{A}^0$ actions. In particular the first and eighth actions can be interpreted as 'travel down the hall' and comprise large numbers of primitive actions, and even the five smaller actions used to turn in the intersection are considerably larger than single $\mathcal{A}^0$ actions.

[3]SARSA parameters λ and α have different meanings than the GNG parameters with the same names.

[4]Because of the -1 reward per step, any agent achieving reward greater than -10 000 in an episode must have reached the goal.

5 Discussion and Future Work

The results of these experiments demonstrate that SODA can dramatically improve reinforcement learning over using the same coarse-coded state representation and short-range, $\mathcal{A}^0$ actions.

The improved task learning in this experiment follows from two sources: reduced positional uncertainty and reduced task diameter. First, state abstraction methods that partition a continuous state space alias the environment, creating uncertainty about the outcome of actions. For example, if two states in the same partition differ in orientation by a few degrees, moving forward may lead to drastically different states. Hill-climbing to the local feature maximum greatly reduces this uncertainty and thus makes actions more reliable, making it easier to learn the task.

Second, using the short-range actions, the task has a large diameter: the agent must make at least 300 actions to get to the goal. Since a critical choice that must be learned – turning right at the intersection – is approximately halfway through the action sequence, it takes many trials and much exploration to back up the reward to the critical decision point and discover the correct choice. In contrast, traveling between distinctive states using $\mathcal{A}^1$ actions, the agent needs only nine actions to arrive at the goal. This decrease in task diameter makes it much easier to propagate the reward back to the critical choice point, and thereby discover critical decisions in the task.

One potential drawback of using a coarse-grained perceptual representation like a SOM is that it may introduce state aliasing. In the task above there was a small amount of aliasing, but the agent was still able to learn the task because the optimal action was the same in the aliased states. In tasks with more significant amounts of aliasing, plain tabular Sarsa(λ) would need to be replaced with a method that constructs a state memory, like U-Tree McCallum (1995) or Temporal Transition Hierarchies Ring (1994). Such a change should not diminish the usefulness of the underlying abstraction.

Finally it must be noted that the learned solutions using the $\mathcal{A}^1$ actions take significantly more steps than the ≈ 300 step optimal solution with $\mathcal{A}^0$ actions. Most of the difference is the extra cost of the sampling steps needed to discover the gradient direction for hillclimbing. The $\mathcal{A}^1$ agents achieve basic competence at the task so much faster than the $\mathcal{A}^0$ agents that it may be desirable to trade off optimality of the final solution against quick learning of some reasonable solution. Nevertheless, it should be possible to reduce or eliminate the need for these sampling steps as the agent learns. The authors' current work investigates allowing the agent to learn a hillclimbing policy for each feature as a nested reinforcement learning problem. The agent uses a separate subordinate, local GNG for the feature representation for each distinctive state neighborhood, and the hillclimbing learner has an action set consisting of the $\mathcal{A}^0$ actions plus a special 'quit' action, indicating when to finish hillclimbing. It is rewarded for achieving the highest possible activation quickly.

In addition to the cost of sampling actions for hillclimbing, the shortest path to the goal via distinctive states may be longer than the absolute optimal path.

If truly optimal behavior is desired, it should be possible to use trajectories generated using the policy of $\mathcal{A}^1$ actions to bootstrap a policy over $\mathcal{A}^0$ actions and continue learning to optimize that policy, using methods similar to those used by Smart and Kaelbling (2000). This is an area for future research.

6 Conclusion

Self-Organizing Distinctive-state Abstraction (SODA) is a method by which a robot in a continuous world builds complementary perceptual and temporal abstractions by first self-organizing a feature map into a set of higher-level perceptual features, and then using those features to build a set of high-level actions that carry it between perceptually distinctive states in the environment. Experiments on a simulated robot navigation task showed that, starting with a robot with uninterpreted sensors and effectors, SODA was able to learn sets of high-level perceptual features, distinctive states, and actions that greatly abstracted and simplified the robot's model of its environment and its own sensorimotor access to it. Applying reinforcement learning to this abstracted model, the robot was able to learn much more quickly to navigate to its goal.

Acknowledgements

This work was conducted in the Intelligent Robotics Lab and the Neural Networks Research Group in the Artificial Intelligence Lab at The University of Texas at Austin. It was supported in part by the National Science Foundation (grant IIS-0413257), by NIMH Human Brain Project (grant IR01-MH66991), and by an IBM Faculty Research Award.

References

Blank, D. and Meeden, L. (2006). Introduction to the special issue on developmental robotics. *Connection Science*, 18(2).

Chaput, H. H., Kuipers, B., and Miikkulainen, R. (2003). Constructivist learning: A neural implementation of the schema mechanism. In *Proceedings of the Workshop on Self-Organizing Maps (WSOM03)*.

Digney, B. (1998). Learning hierarchical control structure for multiple tasks and changing environments. In *Proceedings of the Fifth Conference on the Simulation of Adaptive Behavior: SAB 98*.

Duckett, T. and Nehmzow, U. (2000). Performance comparison of landmark recognition systems for navigating mobile robots. In *Proc. 17th National Conf. on Artificial Intelligence (AAAI-2000)*, pages 826–831. AAAI Press/The MIT Press.

Fritzke, B. (1995). A growing neural gas network learns topologies. In *Advances in Neural Information Processing Systems 7*.

Gerkey, B., Vaughan, R. T., and Howard, A. (2003). The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, Coimbra, Portugal.

Gold, K. and Scassellati, B. (2006). Learning acceptable windows of contingency. *Connection Science*, 18(2).

Hengst, B. (2002). Discovering hierarchy in reinforcement learning with hexq. In Sammut, C. and Hoffmann, A., editors, *Machine Learning: Proceedings of the 19th Annual Conference*, pages 243–250.

Kohonen, T. (1995). *Self-Organizing Maps*. Springer, Berlin.

Kuipers, B. (2000). The Spatial Semantic Hierarchy. *Artificial Intelligence*, 119:191–233.

Kuipers, B., Beeson, P., Modayil, J., and Provost, J. (2006). Bootstrap learning of foundational representations. *Connection Science*, 18(2).

Martinetz, T. M., Ritter, H., and Schulten, K. J. (1990). Three-dimensional neural net for learning visuomotor coordination of a robot arm. *IEEE Transactions on Neural Networks*, 1:131–136.

McCallum, A. K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, Rochester, New York.

McGovern, A. and Barto, A. G. (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. In *Machine Learning: Proceedings of the 18th Annual Conference*, pages 361–368.

Nehmzow, U. and Smithers, T. (1991). Mapbuilding using self-organizing networks in really useful robots. In *Proceedings SAB '91*.

Olsson, L., Nehaniv, C. L., and Polani, D. (2006). From unknown sensors and actuators to actions grounded in sensorimotor perceptions. *Connection Science*, 18(2).

Oudeyer, P.-Y. and Kaplan, F. (2006). The discovery of communication. *Connection Science*, 18(2).

Pierce, D. M. and Kuipers, B. J. (1997). Map learning with uninterpreted sensors and effectors. *Artificial Intelligence*, 92:169–227.

Precup, D. (2000). *Temporal abstraction in reinforcement learning*. PhD thesis, The University of Massachusetts at Amherst.

Provost, J., Beeson, P., and Kuipers, B. J. (2001). Toward learning the causal layer of the spatial semantic hierarchy using SOMs. AAAI Spring Symposium Workshop on Learning Grounded Representations.

Ring, M. B. (1994). *Continual Learning in Reinforcement Environments*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712.

Ryan, M. R. K. (2002). Using abstract models of behaviours to automatically generate reinforcement learning hierarchies. In *Proceedings of The 19th International Conference on Machine Learning*, Sydney, Australia.

Schlesinger, M. (2006). Decomposing infants' object representations: A dual-route processing account. *Connection Science*, 18(2).

Schmidhuber, J. (2006). Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts. *Connection Science*, 18(2).

Smart, W. D. and Kaelbling, L. P. (2000). Practical reinforcement learning in continuous spaces. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000)*, pages 903–910.

Smith, A. J. (2002). Applications of the self-organizing map to reinforcement learning. *Neural Networks*, 15:1107–1124.

Stronger, D. and Stone, P. (2006). Towards autonomous sensor and actuator model induction on a mobile robot. *Connection Science*, 18(2).

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA.

Sutton, R. S., Precup, D., and Singh, S. (1999). Between MDPs and SMDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211.

Toussaint, M. (2004). Learning a world model and planning with a self-organizing, dynamic neural system. In *Advances in Neural Information Processing Systems 16.*