Learn to Behave! Rapid Training of Behavior Automata

Sean Luke
Department of Computer Science
George Mason University
4400 University Drive MSN 4A5
Fairfax, VA 22030 USA
sean@cs.gmu.edu

Vittorio Amos Ziparo
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Ariosto 25, I-00185
Rome, ITALY
ziparo@dis.uniroma1.it

ABSTRACT

Programming robot or virtual agent behaviors can be a challenging task, and makes attractive the prospect of automatically learning the behaviors from the actions of a human demonstrator. However, learning complex behaviors rapidly from a demonstrator may be difficult if they demand a large number of training samples. We describe an architecture for rapid learning of recurrent behaviors from demonstration. The architecture is based on deterministic hierarchical finite-state automata (HFAs) with classification algorithms taking the place of the state transition function. This architecture allows for task decomposition, statefulness, parameterized features and behaviors, per-behavior feature set customization, and storage of learned behaviors in libraries to be used later on as elements in more complex behaviors. We describe the system, then illustrate its application in a simple, but nontrivial, foraging task involving multiple behaviors.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Algorithms, Design, Human Factors

Keywords

Learning from Demonstration, Hierarchical Finite-state Automata, Agents, Robotics

1. INTRODUCTION

Our goal is to enable the rapid, real-time training of complex, stateful agent behaviors. Agent behavior training has applications in a variety of fields, including 3D animation, game level design, and autonomous robotics. In these areas, programming custom domain-specific behaviors on-the-fly may not be desirable or possible, and so it is attractive to instead have the agent learn them from a trainer.

One of the challenges facing training, however, is the conflict between the real-time nature of training and the large numbers of samples that may be demanded by a challenging, high-dimensional domain. It may not be feasible to ask a trainer to perform hundreds of trials to satisfy the needs of a learning algorithm. Thus, one of our goals is to develop methods to reduce domain complexity, and ideally reduce the number of necessary samples, while not sacrificing the gamut of learnable behaviors. We do this by taking advantage of domain knowledge in various ways, and thus our method lies somewhere in the middle-ground between explicit programming (that is, specification) and full, unfettered learning.

Our learned agent behaviors take the form of deterministic hierarchical finite-state automata (HFA). Obviously HFA are not as expressive as other models: for example, the parallelism inherent in Petri Nets; or the richer computational capacity afforded stack automata or arbitrary functions. The motivation underlying the choice of HFAs is twofold. First, HFA are a widely adopted tool for modeling agent and robot behaviors, rich enough for a broad range of common behaviors, yet are simple enough to allow the straightforward demonstration of our learning approach. Second, we chose HFAs as they enabled us to do task decomposition easily.

There are many HFA formulations. Ours is straightforward: a learned behavior is a standard Moore Machine finite-state automaton, where each state is associated with a certain behavior, and also with a transition function which stipulates, given the current world situation, which state to transition to in the next time step. There is a start state but no accepting states.

Our approach is to build an HFA iteratively: we allow the user to easily create an HFA based on a current library of behaviors (some of which may themselves be HFAs). When the HFA is complete, it is added to the library to help build a more complex higher-level HFA. One can create of course an HFA by coding it by hand: but of interest to us is the ability to *learn* the HFA by watching a demonstrator manipulate the agent. As the agent moves about in the environment, the demonstrator directs it to perform various behaviors (and thus to transition to various new states). Each time the demonstrator requests such a transition, the system records the transition and the current world situation. At the end of the training period, from these records the system builds, for each state (behavior), a learned transition function indicating under what conditions the agent should transition to new states. This is essentially a supervised learning task and can employ a variety classification algorithms: at present our learned models take the form of decision trees.

The approach also lends itself to both stochastic and deterministic transitions. Decision trees traditionally compute classes deterministically, based on the most common class among the relevant training examples. Our method can be set up to do this; or to choose classes stochastically based on the proportion of examples from a given class. The experiments in this paper apply the latter method.

The learning domain for an HFA behavior can obviously be complex and of high dimensionality, depending on the number of basic behaviors and the dimensionality of the agent's feature vector. This in turn can require a large number of training sessions to adequately describe the domain. It is not reasonable to expect a demonstrator to perform that many training sessions, and so it is important to reduce the domain space complexity or training difficulty. We have done this in three ways:

- An HFA encourages task decomposition. Rather than learn one large behavior, the system may be trained on simpler behaviors, which are then composed into a higher-level learned behavior. This essentially projects the full learning space into multiple lower-dimensional spaces.
- Feature vector reduction. Our system allows the user to specify precisely those features he feels are necessary for a given learned HFA, which in turn dramatically reduces the learning space. Each HFA, including lower-level HFAs, may have its own different reduced feature vector.
- Generalization by parametrization. All behaviors, including HFAs themselves, may be parameterized with *targets*: for example, rather than create a behavior go-to-home-base, we can create a general behavior go-to(A), and allow for higher-level behaviors to specify the meaning of the target A at a future time. This can significantly reduce the number of behaviors which must be trained.

By employing these complexity-reduction measures, our system ideally enables the rapid construction of complex behaviors, with internal state and a variety of sensor features, in real time entirely by training from demonstration.

The remainder of the paper is laid out as follows. We begin with a discussion of related work. We then describe the basic HFA model and our approach to learning the transition functions in the automaton. We follow this with a training example of a nontrivial foraging behavior, then conclude with a discussion of future directions.

2. RELATED WORK

Our approach generally fits under the category of *learning from demonstration* [3], an overall term for training agent actions by having a human demonstrator perform the action on behalf of the agent. Because the proper action to perform in a given situation is directly provided to the agent, this is broadly speaking a supervised learning task, though a significant body of research in the topic actually involves reinforcement learning, whereby the demonstrator's actions are converted into a reinforcement signal from which the agent is expected to derive a policy. The lion's share of learning from demonstration literature comes not from virtual or game agents but from autonomous robotics. For a large survey of the area, see [2].

Learning Plans. One learning from demonstration area, closely related to our own research, involves the learning of largely directed acyclic graphs of behaviors (essentially plans) from sequences of actions [1, 16, 18, 21], possibly augmented with sequence iteration [25]. Like our approach, these plans are often parameterizable.

Such plan networks generally have limited or no recurrence: instead they usually tend to be organized as sequences or simultaneous groups of behaviors which activate further behaviors downstream. This is mostly a feature of the problem being tackled: such plans are largely induced from ordered sequences of actions intended to produce a result. Since we are training goal-less behaviors rather than plans, our model instead assumes a rich level of recurrence: and for the same reason the specific ordering of actions is less helpful.

Learning Policies. Another large body of work in learning from demonstration involves observing a demonstrator perform various actions when in various world situations. From this the system gleans a set of $\langle \textit{situation, action} \rangle$ tuples performed and builds a policy function $\pi(\textit{situation}) \rightarrow \textit{action}$ from these tuples. This can be tackled as a supervised learning task [4, 5, 8, 10, 12, 15]. However, some literature instead transforms the problem into a reinforcement learning task by providing the learner only with a reinforcement signal based on how closely the learned policy matches the tuples provided by the demonstrator [9, 24]. This is curious given that the problem is, in essence, supervised; the reinforcement methods are in some sense working with reduced information.

Our approach differs from these methods in an important way. Instead of learning $\textit{situation} \rightarrow \textit{action}$ rules, our model learns the transition functions of an HFA with predefined internal states, each corresponding to a possible basic behavior. This enables the demonstrator to differentiate transitions to new behaviors not just based on the current world situation but also the current behavior. That is, we learn rules of the form $\langle \textit{previous action, situation} \rangle \rightarrow \textit{action}$. Another, somewhat different use of internal state would be to distinguish between aliased observations of hidden world situations, something which may be accomplished through learning hidden Markov models (for example, [13]).

Hierarchical Models. The use of hierarchies in robot or agent behaviors is very old indeed, going back as early as Brooks's Subsumption Architecture [7]. Hierarchies are a natural way to achieve layered learning [22] via task decomposition. This is a common strategy to simplify the state space: see [11] for an example. While it is possible in these cases to induce the hierarchy itself, usually such methods iteratively compose hierarchies in a bottom-up fashion.

Our HFA model bears some similarity to hierarchical behavior networks such as those for virtual agents [6] or physical robots [17], in which feed-forward plans are developed, then incorporated as subunits in larger and more complex plans. In such literature, the actual application of hierarchy to learning from demonstration has been unexpectedly limited. However, learning from demonstration has been applied more extensively to multi-level reinforcement learning, as in [23], albeit with a fixed hierarchy.

Language Induction. One cannot mention learning finite state automata without noting that they have a long history in language induction and grammatical inference, with a correspondingly massive literature. For recent surveys of techniques using automata for grammar induction, see [19,

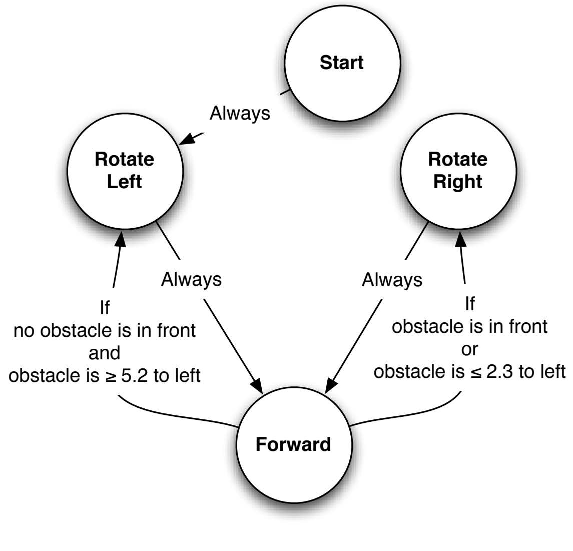

Figure 1: A simple finite-state automaton for wall following (counter-clockwise). All conditions not shown are assumed to indicate that the agent remain in its current state.

26]. However the goal of this literature is fundamentally different from ours in this paper. Specifically, in language induction, the learning algorithm is given a set of positive and negative string examples and generates an automaton which induces an underlying language. Typically these algorithms make no assumptions about the number of states, assume the states are unlabelled, typically assume a small set of transition conditions, and include accepting or rejecting states. In contrast we are not interested in terminating automata, and seek to induce only the edges among a pre-specified set of labeled states, given examples with labelled transitions from state to state.

3. THE HFA MODEL

Using our system, a trainer iteratively develops new finite-state automata, whose states encompass behaviors drawn from a behavior library. An automaton is learned by observing the trainer as he selects various behaviors in various situations. Once learned, the automaton can then be added as a behavior in the library, and then may be itself used as a state in more complex automata. In the following, we first describe the hierarchical finite state automaton model, and in the next section we detail our approach for learning the automaton by demonstration from the trainer.

States and Behaviors. Our HFAs model Moore machines: that is, each state corresponds to a behavior, and when in a state, the HFA performs that behavior. A behavior may be an atomic behavior or may itself be another HFA, leading to the hierarchical definition of the model. Atomic behaviors are hard-coded behaviors provided by the system. For example, the behavior rotate-left might be an atomic behavior: when employing this behavior, the agent will spin counter-clockwise at some rate. The HFA always begins in the *start* state, associated with a special idle behavior, and which always transitions immediately to some other state. Another special state is the optional *done* state, whose behavior simply sets a done flag and immediately transitions to the *start* state. This is used to potentially indicate to higher-level HFAs that the behavior of the current HFA is "done".

Figure 1 shows a simple automaton with four states, corresponding to the behaviors start, rotate-left, rotate-right, and forward. It may appear at first glance that not all HFAs can be built with this model: for example, what if there were *two* states in which the rotate-left behavior needed to be done? This can be handled by creating a simple HFA which does nothing but transition to the rotate-left state and stay there. This automaton is then stored as a behavior called rotate-left2 and used in our HFA as an additional state, but one which performs the identical behavior to rotate-left.

Features. Transitions from state to state are triggered by observable *features* of the environment. One such feature might be distance-to-closest-obstacle-on-my-left. At any time, this feature yields a non-negative value indicating the distance to such an obstacle. In our system features presently take three forms: *categorical features*, which return unordered values like "red" or "blue"; *continuous features*, which return real-valued numbers (like distances); and *toroidal features*, which return real-valued numbers but which are assumed to wrap around in a toroidal fashion (like angles). Boolean features are typically modeled as categorical features. One special boolean feature is the done feature, which is true if the current behavior is a lower-level HFA, and if it has triggered its done flag.

Targets. Importantly, our approach supports parameterized, general-purpose behaviors and transitions. Rather than create a behavior called go-to-obstacle-number-42, we can create a behavior called go-to(A), where A may be specified later. Similarly, rather than the aforementioned feature distance-to-closest-obstacle-on-my-left, we might instead have the more general feature distance-to(B). This separates features and behaviors from the *targets* to which they apply. For example, a feature or behavior may be either specified with regard to one or more *ground targets* ("obstacle 42" or "the closest obstacle on my left") — resulting in a behavior such as go-to(obstacle-42) — or the target may simply be left unspecified (A), to be bound to a ground target at some later time. In the latter case, the unbound target is called a *parameter*.

When an HFA employs features or behaviors with as-of-yet unbound targets (parameters), it must itself present those parameters when used as a behavior by some higher-level HFA. Thus HFAs themselves may be parameterized.

Transitions. In traditional finite-state automata, transitions are represented by directed edges between nodes, each labelled with a condition which may or may not be true about the current features of the environment. Without loss of generality, it's more useful for us to think of a *transition function* which maps the current state and the current feature vector into a new state. The *start* state always transitions to a specific other state; and the *done* state always transitions to the *start* state.

Operating the HFA. Each timestep the HFA is advanced one tick: it performs one step of the behavior associated with its current state, then applies the transition function to determine a new state for next timestep, if any. When a performed behavior is itself an HFA, this operation is recursive: the child HFA likewise performs one step of *its* current behavior, and applies *its* transition function. Additionally, when an HFA transitions to a state whose behavior is an HFA, that HFA is initialized: its initial state is set to the *start* state, and its done flag is cleared.

Formal Model. For the purposes of this work, we define the class of hierarchical finite-state automata models $\mathcal{H}$ as the set of tuples $\langle \mathcal{S}, \mathcal{F}, T, \mathcal{B}, M \rangle$ where:

- $\mathcal{S} = \{S_0, S_1, \ldots, S_n\}$ is a set of *states*, including a distinguished *start* state S_0, and possibly also one *done* state S_*. Exactly one state is active at any time.
- $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ is a set of *observable features* in the environment. The set of features is partitioned in three disjoint subsets representing categorical ($\mathcal{C}$), continuous ($\mathcal{R}$) and toroidal ($\mathcal{A}$) features. Each F_i can assume a value f_i drawn from a finite (in the case of $\mathcal{C}$) or infinite (in the case of $\mathcal{R}$ and $\mathcal{A}$) number of possible values. At any point in time, the present assumed values $\vec{f} = \langle f_1, f_2, \ldots, f_n \rangle$ for each of the $F_1, F_2, \ldots, F_n$ are known as the environment's current *feature vector*.
- $T : F_1 \times F_2 \times \ldots \times F_n \times \mathcal{S} \rightarrow \mathcal{S}$ is a *transition function* which maps a given state S_i, and the current feature vector $\langle f_1, f_2, \ldots, f_n \rangle$, onto a new state S_j. The *done* state S_* is the sole state which transitions to the *start* state S_0, and does so always: $\forall S_k \neq S_* \, \forall \vec{f} \; T(\vec{f}, S_k) \neq S_0$ and $\forall \vec{f} : T(\vec{f}, S_*) = S_0$.
- $\mathcal{B} = \{B_1, B_2, \ldots, B_n\}$ is a set of *atomic behaviors*. By default, the special behavior idle, which corresponds to inactivity, is in $\mathcal{B}$, as may also be the optional behavior done.
- $M : \mathcal{S} \rightarrow \mathcal{H} \cup \mathcal{B}$ is a one-to-one mapping function of states to basic behaviors or hierarchical automata. $M(S_0) =$ idle, and $M(S_*) =$ done. M is constrained by the stipulation that recursion is not permitted, that is, if an HFA $H \in \mathcal{H}$ contains a mapping M which maps to (among other things) a child HFA H', then neither H' nor any of its descendent HFAs may contain mappings which include H.

We further generalize the model by introducing free variables $(G_1, \ldots, G_n)$ for basic behaviors and features: these free variables are known as *targets*. The model remains unaltered, by replacing behaviors B_i with $B_i(G_1, \ldots, G_n)$ and features F_i with $F_i(G_1, \ldots, G_n)$. The main differences are that the evaluation of the transition function and the execution of behaviors will both be based on ground instances of the free variables.

4. LEARNING FROM DEMONSTRATION

The above mechanism is sufficient to hand-code HFA behaviors to do a variety of tasks; but our approach was meant instead to enable the learning of such tasks. Our learning algorithm presumes that the HFA has a fixed set of states, comprising the combined set of atomic behaviors and all previously learned HFAs. Thus, the learning task consists only of learning the transitions among the states: given a state and a feature vector, decide which state (drawn from a finite set) to transition to. This is an ordinary classification task. Specifically, for each state S_i we must learn a classifier $\vec{f} \rightarrow \mathcal{S}$ whose attributes are the environmental features and whose classes are the various states. Once the classifiers have been learned, the HFA can then be added to our library of behaviors and itself be used as a state later on.

Because the potential number of features can be very high, and many unrelated to the task, and because we want to learn based on a very small number of samples, we wish to reduce the dimensionality of the input space to the machine learning algorithm. This is done by allowing the user to specify beforehand which features will matter to train a given behavior. For example, to learn a Figure-8 pattern around two unspecified targets A and B, the user might indicate a desire to use only four parameterized features: distance-to(A), distance-to(B), direction-to(A), and direction-to(B). During training the user temporarily binds A and B to some ground targets in the environment, but after training they are unbound again. The resulting learned behavior will itself have two parameters (A and B), which must ultimately be bound to use it in any meaningful way later on.

The training process works as follows. The HFA starts in the "start" state (idling). The user then directs the agent to perform various behaviors in the environment as time progresses. When the agent is presently performing a behavior associated with a state S_i and the user chooses a new behavior associated with the state S_j, the agent transitions to this new behavior and records an *example*, of the form $\langle S_i, \vec{f}, S_j \rangle$, where $\vec{f}$ is the current feature vector. Immediately after the agent has transitioned to S_j, it turns out to be often helpful to record an additional example of the form $\langle S_j, \vec{f}, S_j \rangle$. This adds at least one "default" (that is, "keep doing state S_j") example, and is nearly always correct since in that current world situation the user, who had just transitioned to S_j, would nearly always want to stay in S_j rather than instantaneously transition away again.

At the completion of the training session, the system then builds transition functions from the recorded examples. For each state S_k, we build a decision tree D_{S_k} based on all examples where S_k is the first element, that is, of the form $\langle S_k, \vec{f}, S_i \rangle$. Here, $\vec{f}$ and S_i form a data sample for the classifier: $\vec{f}$ is the input feature and S_i is the desired output class. If there are no examples at all (because the user never transitioned from S_k), the transition function is simply defined as always transitioning to back to S_k.

At the end of this process, our approach has built some N decision trees, one per state, which collectively form the transition function for the HFA. After training, some states will be unreachable because the user never visited them, and so no learned classification function ever mapped to them. These states may be discarded. The agent can then be left to wander about in the world on its own, using the resulting HFA.

Though in theory many classification algorithms are applicable (such as K-Nearest-Neighbor or Support Vector Machines), in our experiments we chose to use a variant of the C4.5 Decision Tree algorithm [20] for several reasons:

1. Many areas of interest in the feature space of our agent approximately take the form of rectangular regions (angles, distances, etc.).

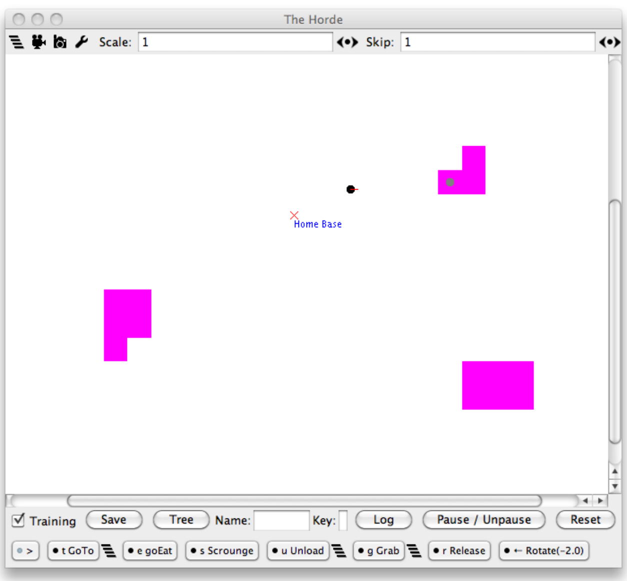

Figure 2: The foraging scenario in our testbed.

2. Decision trees nicely handle various kinds of data: in our case, we used categorical, real-valued, and toroidal data (the latter requiring so-called "pie-slice" decision tree splits).

3. Decision trees are particularly adept at handling unscaled dimensions in the feature space. In our case, we would otherwise be faced with asking how many units of distance were equivalent to a degree of angle, or to a change from "true" to "false".

In decision trees, the class is most commonly computed deterministically: the leaf node in a decision tree is set to the class appearing among the plurality of training examples which wound up at that leaf node. During the implementation and the evaluation of our algorithm, we found out that in many cases we would not want a deterministic classification. For example, when performing a wall-following behavior, we'd need to turn left some *percentage* of time. As a result, our decision tree procedure can also compute classes stochastically, with probability based on the proportion of relevant examples at a given leaf node rather than a plurality vote. In the following example, we solely use this second method.

5. EXAMPLE

We have implemented an experimental research testbed for training agents using this approach (Figure 2), written with the MASON multiagent simulation toolkit [14] (see http://cs.gmu.edu/∼eclab/projects/mason/). In the environment, our agent can sense a variety of things: the relative locations of obstacles, other agents of different classes, certain predefined waypoints, food locations, etc. In this testbed, the experimenter trains an HFA by first selecting features relevant to the behaviors (see Figure 3), then grounding targets for behaviors and features, then directing the agent to

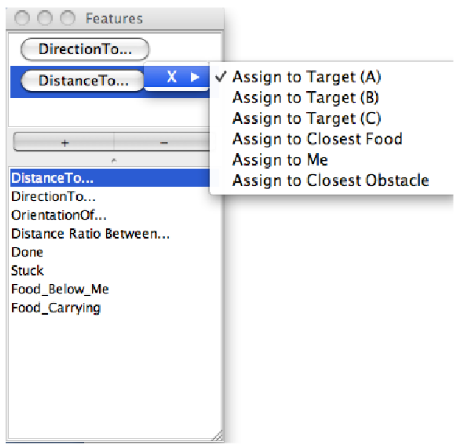

Figure 3: Feature selection and target assignment.

perform behaviors by pressing various buttons or keystrokes, and then finally adding the trained HFA to the system library.

We have successfully trained several simple behaviors, tracking and acquiring a target, wall-following, generic obstacle circumnavigation, and tracing paths (such as a figure eight path between two targets). In this section, we give an example where we have trained the agent to perform a moderately complex foraging task: to harvest food from food sources and bring it back to deposit at the agent's central station. Food can be located anywhere, as can the station. Food at a given location can be in any concentration, and depletes, eventually to zero, as it is harvested by the agent. The agent can only store so much food before it must return to the station to unload. There are various corner cases: for example, if the agent depletes food at a harvest location before it is full, it must continue harvesting at another location rather than return to the station. The scenario is shown in Figure 2: the black circle is the agent, pink areas are food sources, and the red "×" (labelled "Home Base") is the station.

Foraging tasks are of course old hat in robotics, and are not particularly difficult to code by hand. But *training* such a behavior is less trivial. We selected this task as an example because it illustrates a number of features special to our approach: our foraging behavior is in fact a three-layer HFA hierarchy; employs "done" states; involves real-valued, toroidal, and categorical (boolean) inputs; and requires one behavior with an unbound parameter used in two different ways.

The behavior is shown in Figure 4. It requires seven basic behaviors: start and done, forward, rotate-left, rotate-right, load-food (deplete the current location's food by 1, and add 1 to the agent's stored food), and unload-food (remove all the agent's stored food). It also requires several features: distance-to(A), angle-to(A), food-below-me (that is, how much food is located here), food-stored-in-me, and done. Finally, it requires two targets to bind to A: the station and nearest-food.

From this we manually decomposed the foraging task into a hierarchy of four HFA behaviors, and trained each one in turn as described next. All told, we were able to train all four

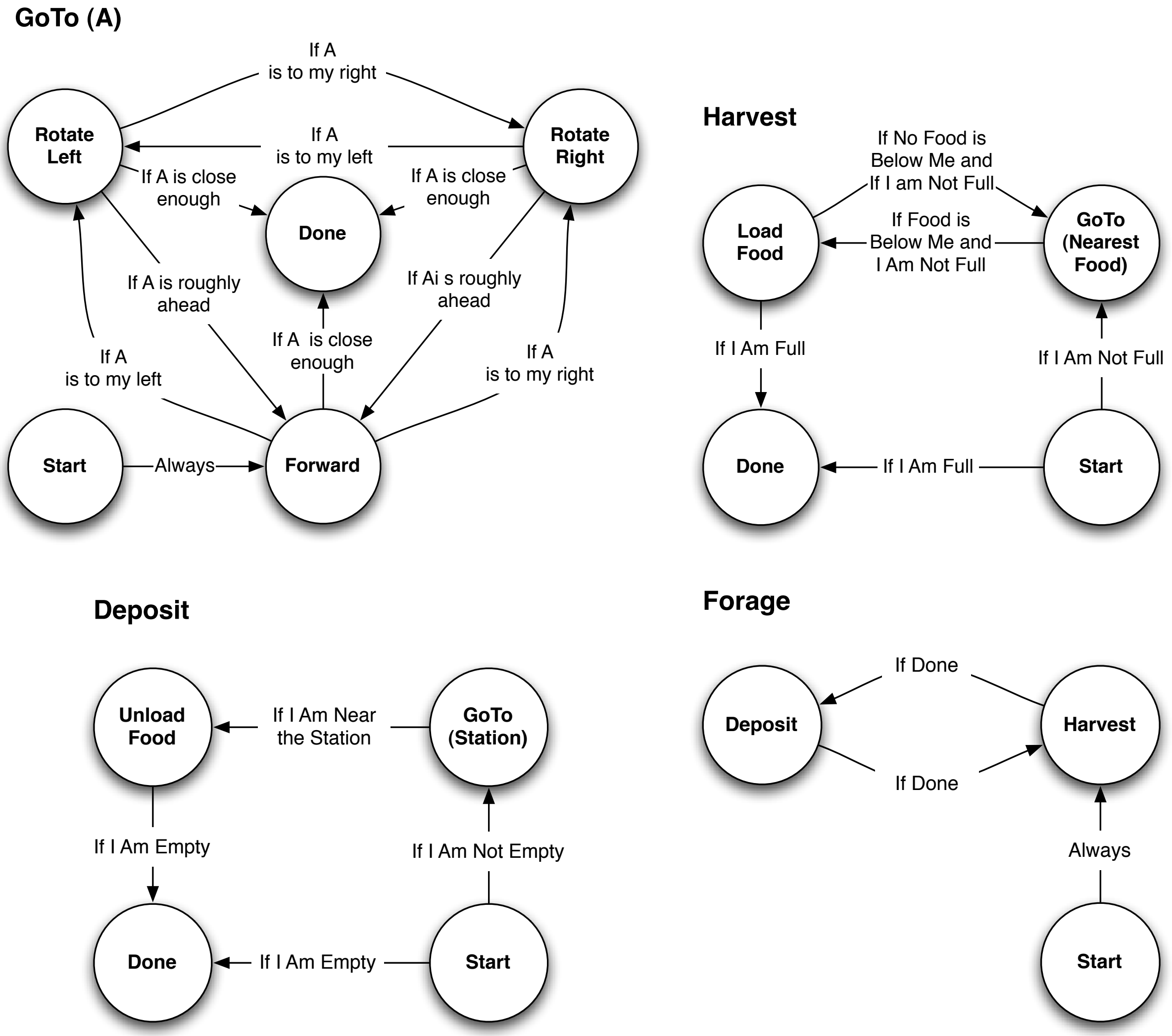

Figure 4: The Forage behavior and its sub-behaviors: Deposit, Harvest, and GoTo(*Parameter* A). All conditions not shown are assumed to indicate that the agent remain in its current state.

behaviors, and demonstrate the agent properly foraging, in a manner of minutes.

The GoTo(A) Behavior. This behavior caused the agent to go to the object marked A. The behavior was a straightforward bang-bang servoing controller: rotate left if A is to the left, else rotate right if A is to the right; else go forward; and when close enough to the target, enter the "done" state.

We trained the GoTo(A) behavior by temporarily declaring a marker in the environment to be Parameter A, and reducing the features to just distance-to(A) and angle-to(A). We then placed the agent in various situations with respect to Parameter A and "drove" it over to A by pressing keys corresponding to the rotate-left, rotate-right, forward, and done behaviors. After a short training session, the system quickly learned the necessary behaviors to accurately go to the target and signal completion. Once completed, it was made available in the library as go-to(A).

The Harvest Behavior. This behavior caused the agent to go to the nearest food, then load it into the agent. When the agent had filled up, it would signal that it was done. If the agent had not filled up yet but the food has been depleted, the agent would search for a new food location and continue harvesting. This behavior employed the previously-learned go-to(A) behavior as a subsidiary behavior, binding its Parameter A to the nearest-food target. This behavior also employed the features food-below-me and food-stored-in-me.

We trained the Harvest Behavior by directing the agent to go to the nearest food, then load it, then (if appropriate) signal "done", else go get more food. We also placed the agent in various corner-case situations (such as if the agent started out already filled up with food). Again, we were able to rapidly train the agent to perform harvesting. Once completed, it was made available in the library as harvest.

The Deposit Behavior. This behavior caused the agent to go to the station, unload its food, and signal that it is done. If the agent was already empty when starting, it would immediately signal done. This behavior also used the previously-learned go-to(A) behavior as a subsidiary state behavior, but instead bound its Parameter A to the station target. It used the features food-stored-in-me and distance-to(station). We trained the Deposit Behavior in a similar manner as the Harvest Behavior, including various corner cases. Once completed, it was made available in the library as deposit.

The Forage Behavior. This simple top-level behavior just cycled between depositing and harvesting. Accordingly, this behavior employed the previously-learned deposit and harvest behaviors. The behavior used only the done feature.

6. CONCLUSION

In this paper, we have presented an approach for training agent behaviors using a hierarchical deterministic finite state automata model and a classification algorithm, implemented as a variant of the C4.5 algorithm. The main goal of our approach is to enable users to train agents rapidly based on a small number of training examples. In order to achieve this goal, we trade off learning complexity with training effort, by enabling trainers to decompose the learning task in a hierarchical manner, to learn general parameterized behaviors, and to explicitly select the most appropriate features to use when learning. This in turn reduces the dimensionality of the learning problem.

We have developed a proof of concept testbed simulator which appears to work well: we can train parameterized, hierarchical behaviors for a variety of tasks in a short period of time. We are presently deploying the platform to robots in our laboratory. In the mean time, there are a number of interesting issues that remain to be dealt with.

Multiple Agents. Our immediate next goal is to move to training multiple agents. In the general case, multiagent learning is a much more complex task than single-agent learning, involving game-theoretic issues which may be well outside the scope of the learning facility. However we believe there are obvious approaches to certain simple multiagent learning scenarios: for example teaching agents to perform actions as *homogeneous behavior* groups (perhaps by training an agent with respect to other agents not under his control, but moving them similarly). Another area of multiple agent training may involve *hierarchies of agents,* with certain agents in control of teams of other agents.

Unlearning. There are two major reasons why an agent may make an error. First, it may have learned poorly due to an insufficient number of examples or unfortunately located examples. Second, it may have been misled due to *bad examples.* This second situation arises due to errors in the training process, something that's surprisingly easy to do! When an agent makes a mistake, the user can jump in and correct it immediately, which causes the system to drop back into training mode and add those new examples to the behavior's collection. However this does *not* cause any errant examples to be removed. Since the agent made an error based not on examples but rather based on the learned function, identifying which examples were improper, and whether to remove them, may prove a challenge.

Programming versus Training. We have sought to train agents rather than explicitly code them. However we also aimed to do so with a minimum of training. These goals are somewhat in conflict. To reduce the training necessary, we typically must reduce the problem space complexity and/or dimensionality. We have so far done so by allowing the user to inject domain knowledge into the problem (via task decomposition, for example, or by explicitly training for certain corner cases). This is essentially a step towards having the user explicitly declare part of the solution rather than have the learner induce it. So is this learning or coding?

We think that training of this sort is somewhere in-between: in some sense the learning algorithm is relieving the trainer from having to "code" everything himself. The question worth studying is: how much learning is useful before the number of samples required to learn outweighs the reduced "coding" load, so to speak, on the trainer?

Other Representations. HFAs cannot straightforwardly do parallelism or planning. We chose HFAs largely because they were simple enough to make training intuitively feasible. Now that we've demonstrated this, we wish to examine how to train with other common representations, such as Petri nets or hierarchical task network plans, to demonstrate the generality of the approach.

7. ACKNOWLEDGMENTS

This work was supported in part by NSF grant 0916870.

8. REFERENCES

[1] R. Angros, W. L. Johnson, J. Rickel, and A. Scholer. Learning domain knowledge for teaching procedural skills. In *The First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1372–1378. ACM, 2002.

[2] B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57:469–483, 2009.

[3] C. G. Atkeson and S. Schaal. Robot learning from demonstration. In D. H. Fisher, editor, *Proceedings of the Fourteenth International Conference on Machine Learning (ICML)*, pages 12–20. Morgan Kaufmann, 1997.

[4] M. Bain and C. Sammut. A framework for behavioural cloning. In *Machine Intelligence 15*, pages 103–129. Oxford University Press, 1996.

[5] D. C. Bentivegna, C. G. Atkeson, and G. Cheng. Learning tasks from observation and practice. *Robotics and Autonomous Systems*, 47(2-3):163–169, 2004.

[6] R. Bindiganavale, W. Schuler, J. M. Allbeck, N. I. Badler, A. K. Joshi, and M. Palmer. Dynamically altering agent behaviors using natural language instructions. In *Autonomous Agents*, pages 293–300. ACM Press, 2000.

[7] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[8] S. Calinon and A. Billard. Incremental learning of gestures by imitation in a humanoid robot. In C. Breazeal, A. C. Schultz, T. Fong, and S. B. Kiesler, editors, *Proceedings of the Second ACM SIGCHI/SIGART Conference on Human-Robot Interaction (HRI)*, pages 255–262. ACM, 2007.

[9] A. Coates, P. Abbeel, and A. Y. Ng. Apprenticeship learning for helicopter control. *Communications of the ACM*, 52(7):97–105, 2009.

[10] J. Dinerstein, P. K. Egbert, and D. Ventura. Learning policies for embodied virtual agents through demonstration. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1257–1252, 2007.

[11] D. Grollman and O. Jenkins. Learning robot soccer skills from demonstration. In *IEEE 6th International Conference on Development and Learning (ICDL)*, pages 276–281, July 2007.

[12] M. Kasper, G. Fricke, K. Steuernagel, and E. von Puttkamer. A behavior-based mobile robot architecture for learning from demonstration. *Robotics and Autonomous Systems*, 34(2-3):153–164, 2001.

[13] D. Kulic, D. Lee, C. Ott, and Y. Nakamura. Incremental learning of full body motion primitives for humanoid robots. In *8th IEEE-RAS International Conference on Humanoid Robots*, pages 326–332, Dec. 2008.

[14] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. Mason: A multi-agent simulation environment. *Simulation*, 81(7):517–527, July 2005.

[15] J. Nakanishi, J. Morimoto, G. Endo, G. Cheng, S. Schaal, and M. Kawato. Learning from demonstration and adaptation of biped locomotion. *Robotics and Autonomous Systems*, 47(2-3):79–91, 2004.

[16] M. N. Nicolescu and M. J. Mataric. Learning and interacting in human-robot domains. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 31(5):419–430, 2001.

[17] M. N. Nicolescu and M. J. Mataric. A hierarchical architecture for behavior-based robots. In *The First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 227–233. ACM, 2002.

[18] M. N. Nicolescu and M. J. Mataric. Natural methods for robot task learning: instructive demonstrations, generalization and practice. In *The Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 241–248. ACM, 2003.

[19] R. Parekh and V. Honavar. Grammar inference, automata induction, and language acquisition. In *Handbook of Natural Language Processing*, pages 727–764. Marcel Dekker, 2000.

[20] J. R. Quinlan. *C4.5: Programs for Machine Learning (Morgan Kaufmann Series in Machine Learning)*. Morgan Kaufmann, 1 edition, January 1993.

[21] P. E. Rybski, K. Yoon, J. Stolarz, and M. M. Veloso. Interactive robot task training through dialog and demonstration. In C. Breazeal, A. C. Schultz, T. Fong, and S. B. Kiesler, editors, *Proceedings of the Second ACM SIGCHI/SIGART Conference on Human-Robot Interaction (HRI)*, pages 49–56. ACM, 2007.

[22] P. Stone and M. M. Veloso. Layered learning. In R. L. de Mántaras and E. Plaza, editors, *11th European Conference on Machine Learning (ECML)*, pages 369–381. Springer, 2000.

[23] Y. Takahashi and M. Asada. Multi-layered learning system for real robot behavior acquisition. In V. Kordic, A. Lazinica, and M. Merdan, editors, *Cutting Edge Robotics*. Pro Literatur, 2005.

[24] Y. Takahashi, Y. Tamura, and M. Asada. Mutual development of behavior acquisition and recognition based on value system. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 386–392. IEEE, 2008.

[25] H. Veeraraghavan and M. M. Veloso. Learning task specific plans through sound and visually interpretable demonstrations. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2599–2604. IEEE, 2008.

[26] E. Vidal. Grammatical inference: An introductory survey. In *Grammatical Inference and Applications*, pages 1–4. Springer, 1994.