# Multiagent Supervised Training with Agent Hierarchies and Manual Behavior Decomposition

**Keith Sullivan** **Sean Luke**
Department of Computer Science
George Mason University, Fairfax, VA, USA
{ksulliv2, sean}@cs.gmu.edu

## Abstract

We present a supervised learning from demonstration system capable of training stateful and recurrent behaviors, both in the single agent and multiagent case. Furthermore, behavior complexity due to statefulness and multiple agents can result in a high dimensional learning space, which can require many samples to learn properly. Our approach, which relies heavily on both per-agent behavior decomposition and structuring agents into a tree hierarchy, can significantly reduce the number of samples and make such training feasible. We demonstrate our system in a simulated collective foraging task where all the agents execute the same behavior set. We also discuss how to extend our approach to a heterogeneous case, where different subgroups of agents perform different behaviors.

## 1 Introduction

We present a system for interactive real-time training of behaviors for potentially large groups of robots or software agents via supervised learning from demonstration. We are interested in relatively "complex" behaviors, by which we mean ones which can involve many internal states, make probabilistic decisions, are heavily recurrent, are parameterizable, and exist at both the agent level and at the group level.

We realize that this is a tall order for three major reasons. First, single-agent *stateful, recurrent behaviors* present a learning problem of potentially high dimensionality: but real-time learning from demonstration yields only a small number of training examples, which may be inadequate to model the space. Second, the design space for *multiagent behaviors* does not scale gently: as the number of agents grows, the emergent macrophenomena which arise from their interactions can become more complex and difficult to predict, again presenting a dimensionality problem for the learner, particularly if the group of agents are heterogeneous. Third, these emergent macrophenomena presents a daunting *inverse problem* for supervised learning methods. Such methods usually require that the agents be given samples of which micro-level behaviors to perform in various situations: but the experimenter does not know this — he only (qualitatively) knows the macro-level behavior he wishes to produce, and may not be clear what micro-level behaviors will produce this (due to emergence).

Still, we believe that we can make some progress. Our approach tackles these issues in two ways: by using manual task decomposition to learn *hierarchies of behaviors* within agents; and second, by organizing the agents into a tree-structured *hierarchy of agents*. Both of these "hierarchical" mechanisms enable a divide-and-conquer approach to simplifying the problem, albeit in different ways. We discuss each next.

**Hierarchical Trained Behaviors** We manually decompose the task into a hierarchy of subtasks, and train a behavior for each subtask. Lower level behaviors are composed into more complex behaviors via scaffolding. Each behavior is a finite-state automaton whose states are previously-learned automata or hard-coded behaviors: the learner develops the transition function between the states. Task decomposition allows us to project the complex learning problem into multiple smaller, often trivial, subproblems. It also allows us to reduce the feature space on a per-subproblem basis. Finally, our behaviors are parameterizable ("go to A" versus "go to the door"), allowing them to be reused in different contexts.

**Agent Hierarchies** We partition the agents into a tree hierarchy of subgroups, with basic agents as leaf nodes and *controller agents* as non-leaf nodes in a forest of trees. We recognize that a tree structure has disadvantages, notably rigidity of organization. However, we feel that the simplicity of a tree offers advantages in our system. Importantly, this hierarchy allows us to run the gamut from fully-distributed (each agent is its own one-node tree) clear to completely centralized (all agents are grouped under a single root controller). For example, we could create a multilayer hierarchy with level-1 controller agents each in charge of small squads of basic agents, then level-2 controller agents each in charge of larger groups of level-1 controller agents, and finally a root-level controller agent. Controller agents may be trained in a fashion essentially the same as basic agents.

We imagine that agent hierarchies and controller agents are most useful when they enable behavior heterogeneity in the group (directing different subgroups to perform different tasks, for example). We are ultimately working towards this end: but for now we have been working solely with homogeneous multiagent hierarchical behaviors. Even this can be useful, however, in breaking the group into coordinating subgroups.

In this paper we give an example of a successfully trained multiagent collective foraging task. We compare separate agents versus a single layer hierarchy versus a two layer hierarchy. We also compare the performance of trained behaviors to that of hard-coded behaviors. While we can (and do) run the system on robots, in this paper we demonstrate in simulation due to the number of agents involved.

## 2 Previous Work

Our approach touches on a variety of subareas in the multiagent learning and learning from demonstration literature. We discuss some of them below.

Much of the robot learning from demonstration literature may be divided into systems which learn plans (for example [Nicolescu and Mataric, 2002; Veeraraghavan and Veloso, 2008]) and those which learn *policies*, that is, stateless mappings from the agent's feature vector to a desired action [Bentivegna *et al.*, 2004; Dinerstein *et al.*, 2007; Kasper *et al.*, 2001; Nakanishi *et al.*, 2004]. Some work involves stateful models related to ours, notably via Hidden Markov Models. For example, [Hovland *et al.*, 1996] treat states not as behaviors but as hidden world conditions which the learner is attempting to discover and optimize for. [Goldberg and Mataric, 2002] learns transitions between states corresponding to behaviors, though it does not label the transitions.

Task decomposition and iterative hierarchical learning is a natural way to achieve layered learning [Stone and Veloso, 2000]. Such approaches can be done to learn policies: perhaps most similar to ourselves is [Saunders *et al.*, 2006], who train sequences and stateless policies iteratively built on earlier ones. Plans may also be decomposed, then iteratively learned in a similar fashion fashion [Nicolescu and Mataric, 2002].

One of the primary challenges addressed by this paper is in applying learning from demonstration — at its heart a supervised task — to the multiagent case. As noted in [Panait and Luke, 2005], supervised learning methods are not very common in multiagent learning: by far the lion's share of literature is based on reward-based methods such as reinforcement learning or stochastic optimization. Of those supervised methods, many fall in the category of *agent modeling*, where agents learn about one another rather than about a task given to them by demonstrator. For example, in the celebrated [Stone and Veloso, 2000], the supervised task ("pass evaluation") is reasonably described as agent modeling, while the full multiagent learning task ("pass selection") uses reinforcement learning. Multiagent learning may also be achieved via confidence estimation rather than reinforcement learning [Chernova, 2009].

## 3 Agent Behaviors

Each agent in our world learns a set of one or more behaviors. Each of these behaviors takes the form of a hierarchy of finite-state automata in the form of Moore machines. An automaton is a tuple $\langle S, F, T, B \rangle \in \mathscr{H}$ defined as follows:

- $S = \{S_1, ..., S_n\}$ is the set of *states* in the automaton. Among other states, there is one *start state* $S_1$ and zero or more *flag states*. Exactly one state is active at a time, designated $S_t$. The states are internal states of the automaton, not world states as we do not model world situations.
- $B = \{B_1, ..., B_k\}$ is the set of *basic* (hard-coded) behaviors. Each state is associated with either a basic behavior or *another automaton* from $\mathscr{H}$, with the stipulation that recursion is not permitted.
- $F = \{F_1, ..., F_m\}$ is the set of observable *features* in the environment. At any given time, each feature has a numerical value. The collective values of $F$ at time $t$ is the environment's *feature vector* $\vec{f}_t = \langle F_1, ..., F_m \rangle$.
- $T = F_1 \times ... \times F_m \times S \rightarrow S$ is the *transition function* which maps the current state $S_t$ and the current feature vector $\vec{f}_t$ to a new state $S_{t+1}$.
- We generalize the model with free variables (parameters) $G_1, \ldots, G_n$ for basic behaviors and features. We replace each behavior $B_i$ with $B_i(G_1, \ldots, G_n)$ and feature $F_i$ with $F_i(G_1, \ldots, G_n)$. The evaluation of the transition function and the execution of behaviors will both be based on ground instances (*targets*) of the free variables.

An automaton starts in its *start* state $S_1$, whose behavior simply idles. Each timestep, while in state $S_t$, the automaton queries the transition function to determine the next state $S_{t+1}$, transitions to this state, and if $S_t \neq S_{t+1}$, stops performing $S_t$'s behavior and starts performing $S_{t+1}$'s behavior. Finally, $S_{t+1}$'s behavior is pulsed. If the associated behavior is itself an automaton, this pulsing process recurses into that automaton.

Features may describe both internal and external (world) conditions, and may be toroidal (such as "angle to goal"), continuous ("distance to goal"), or categorical or boolean ("goal is visible").

The purpose of a flag state is simply to raise a flag in the automaton to indicate that the automaton believes that some condition is now true. Two obvious conditions might be *done* and *failed*, but there could be others. Flags in an automaton appear as optional features in its *parent* automaton. For example, the *done* flag may be used by the parent to transition away from the current automaton because that automaton believes it has completed its task.

Behaviors and features may be optionally assigned one or more parameters: rather than have a behavior called *go to the ball*, we can create a behavior called *goTo(A)*, where *A* is left unspecified. Similarly, a feature might be defined not as *distance to the ball* but as *distanceTo(B)*. If such a behavior or feature is used in an automaton, either its parameter must be bound to a specific *target* (such as "the ball" or "the nearest obstacle"), or it must be bound to some higher-level parent *C* of the automaton itself. Thus finite-state automata may themselves be parameterized.

### 3.1 Training a Single-Agent Behavior

We build a "complex" agent behavior by first manually decomposing it into a hierarchy of $N$ smaller behaviors, then train each of those behaviors bottom-up until we have constructed (via scaffolding) the "complex" one. We begin with a *behavior library* consisting of basic (hard-coded) behaviors available to the agent. Members of the behavior library form the available states of the new behavior presently being trained. Each time a behavior is learned, it is entered into the *behavior library* and made available as a state for upcoming trained behaviors.

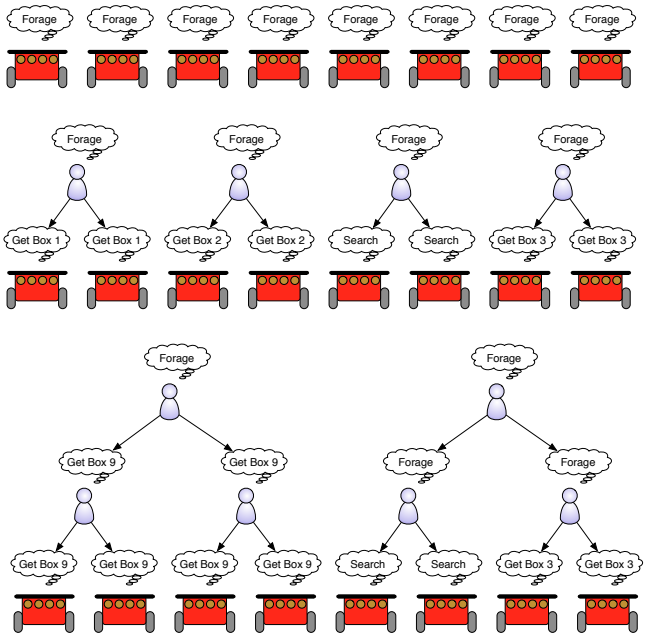


Figure 1: Structures used in the Experiments. At top is the *fully distributed* structure, with each agent running an identical copy of the same behavior (and behavior libraries). Center is a simple *one-level* hierarchical structure, with agents grouped under one level of controller agents. At bottom is a simple *two-level* hierarchical structure, with level-1 controller agents grouped under various level-2 controller agents. All agents under a controller agent run the same behavior, though the current state of that behavior may vary depending on the current situation of the subgroup rooted by that agent.

To train a behavior, we place the agent into *training mode*, where it performs exactly those behaviors directed by the user. Each time the user directs the agent to perform some new behavior, the agent records a training example: a triple $\langle B_t, f_t, B_{t+1} \rangle$ consisting of the current behavior $B_t$, the current feature vector $f_t$, and the desired new behavior $B_{t+1}$. If behavior $B_{t+1}$ should be executed only once, then no additional examples are recorded. Otherwise, another example is recorded of the form $\langle B_{t+1}, f_t, B_{t+1} \rangle$ (saying "if I am doing $B_{t+1}$, and the world looks like $f_t$, continue doing $B_{t+1}$"). The feature vector is specified by the user from a library of predefined but parameterizable features appropriate to the task.

After directing the agent for some time, the user then switches to *testing mode*, which causes the agent to learn the finite-state automaton. Since the automaton's states are fixed (they correspond to the behaviors currently in the behavior library), the training process simply learns the transition functions for the states in the automaton. To learn the transition function for a given state $S_i$ and associated behavior $B_i$, we take all recorded examples of the form $\langle B_i, f_a, B_b \rangle$, then reduce them to simply $\langle f_a, B_b \rangle$, which form points $f_a$ with labels $B_b$ for a classification algorithm. At present our classification algorithm is a C4.5-style decision tree with probabilistic leaf nodes: but most any classification algorithm could be used. The resulting classifiers form the transition functions for the automaton. Unused states in the automaton are trimmed and the automaton is saved to the library. Finally, if a learned automaton uses a parameterized behavior or feature, each such parameter must either be bound to a ground target ("the nearest ball") or to a parameter of the automaton itself.

During testing mode an agent performs the learned behavior interactively with the user. If the user notices a wayward action, he may step in at any time, immediately transferring the agent back to training mode, and correct the action, adding more examples, then reenter testing mode. For a more detailed discussion of the single agent model see [Luke and Ziparo, 2010; Sullivan *et al.*, 2010].

### 3.2 Homogeneous Multiagent Hierarchies

We wish to develop not just single-agent behaviors but collective multiagent behaviors. The obvious (distributed) approach is to simply endow all agents with the same top-level behavior. An alternative centralized approach is to define a single *controller agent* in charge of all subsidiary agents. The subsidiary agents all have the same behaviors in their libraries; but the controller agent has its own separate library of behaviors, both basic behaviors and learned automata. A controller agents' basic behaviors do not manipulate the controller, but instead correspond to a unique behavior in the libraries of the subsidiaries. When a controller agent transitions to a new basic behavior, this directs the subsidiaries to immediately start performing the corresponding behavior in their libraries.

Our framework is in-between: we define a hierarchy of controller agents. The basic agents are grouped into subgroups, each headed by a level-1 controller agent; then various level-1 controller agents are grouped as subsidiaries to level-2 agents, and so on, up to level-$m$ agents forming one or more roots. Just as all basic agents have the same behaviors, all controller agents at a given level have the same behaviors. The actual structure of the hierarchy (number of levels, number of agents per controller, etc.) is at present pre-specified by the user.

After training basic agents in the usual fashion, we may then train a level-1 controller agent, then a level-2 controller agent, and so on. Controller agent training is essentially the same as for basic agents: the user directs the controller agent to perform various behaviors (which in turn cause the controller's subsidiaries to perform behaviors), which adds examples to a database from which transition functions are learned. While the basic behaviors for a controller agent are straightforward, what is a controller agent's set of features? We presume that, unlike a basic agent, a controller agent isn't embodied: his features are derived from statistical results from his subsidiaries: for example "a basic agent in my group is stuck (or isn't)", or "all my immediate subsidiaries are 'done' (or not)", or "the average Y position of basic agents in my group". Obviously, like an agent's basic behaviors and features, the choice of features available to a controller agent are domain-specific.

## 4 Demonstration: Box Foraging

We applied our multiagent homogeneous hierarchies to a simulated box foraging problem: agents hunt for boxes, then pull them towards a known deposit location. The boxes are randomly distributed throughout the environment, and after

collection, a new box is placed randomly in the environment. The environment consists of various circular "boxes" and 50 agents in a 200 by 200 environment. The agents moved 0.1 units per timestep. All experiments used 10 boxes, each 5 units in diameter. Some boxes require 5 agents to pull them, and others require 25 agents. When a box is close enough to the deposit location, it is considered "deposited": it is reset to a new random location and all agents are released from it.

**Agent Behavior Decomposition** We manually decomposed and trained a behavior hierarchy as follows:

- Agents' basic features were *DistanceTo(X), DirectionTo(X), ICanSeeABox, IAmAttachedToABox,* and *Done*. The first two features were parameterizable to either visible boxes or to the deposit location. The last feature was true when the *done* flag had been raised. Boxes could only be seen if they were within 10 units.
- Agents' basic behaviors were *Forward, RotateLeft, RotateRight, GrabBox, ReleaseBox, ReleaseBoxAndFinish,* and *Done*. Both *ReleaseBox* and *Done* would raise the *done* flag and (as normal) immediately transfer to the start state. *ReleaseBoxAndFinish* would as well, except that it would also raise a finished flag in the agent which could be detected by controllers as a feature. Boxes could only be grabbed if they were sufficiently close (5 units).
- Using *Forward, RotateLeft,* and *RotateRight*, we trained *Wander*, which wandered randomly.
- Using *Forward, RotateLeft,* and *RotateRight*, plus the *DistanceTo(X)* and *DirectionTo(X)* features, we trained the behavior *Goto(X)*, which servoed to a given target.
- Using *Goto(X)*, *GrabBox*, *ReleaseBoxAndFinish*, and *DistanceTo(X)*, we trained *ReturnWithBox*, which pulled the box back to the deposit location and released it when the agent was close enough to home.
- Using *Wander* and *ReturnWithBox*, we trained *Forage*, a simple top-level composition which foraged for boxes and brought them to the deposit.

If agents were acting on their own (they had no controller), their top-level behavior would be simply *Forage*. When acting under a 1-level controller, the current behavior of the agent would be determined by the controller.

**1-Level Controller Agent Behavior Decomposition** The 1-level controller's behavior hierarchy was as follows:

- A controller's basic features were *SomeoneIsAttachedToABox* and *SomeoneIsFinished*. The latter feature was true if any subsidiary agent had raised its *finished* flag. A controller also had access to an additional target: *closest-attached-agent*, which pointed to the subsidiary agent which had grabbed the box (if any).
- A controller's basic behaviors corresponded to the full set of behaviors of its subsidiary agents: *Forward, RotateLeft, RotateRight, GrabBox, ReleaseBoxAndDone, Done, Wander, Goto(X), ReturnWithBox,* and *Forage*.

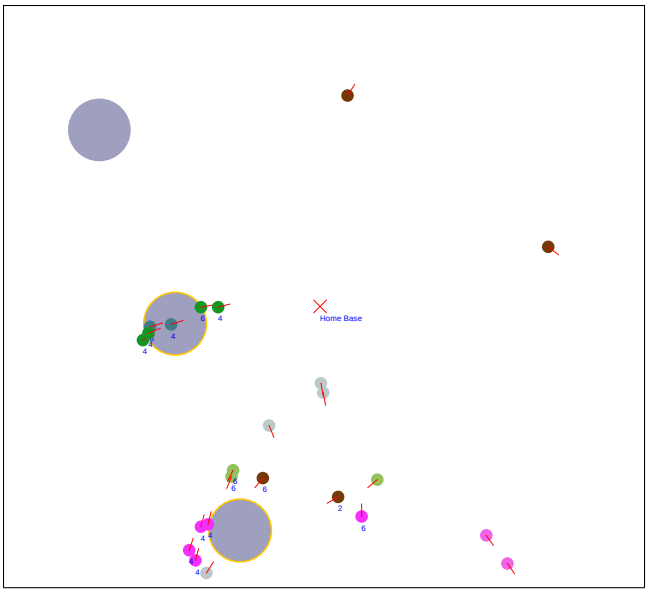


Figure 2: A screenshot of our system in action (showing part of the environment). The large grey circles are the boxes, and the X in the middle is the collection location. Note that while the agents pulling the box on left are all from the same subgroup, the box in the bottom is being pulled by agents from different subgroups.

- Using *ReleaseBox*, *ReturnWithBox*, *Forage*, *SomeoneIsAttachedToABox*, *SomeoneIsFinished*, and *Goto(closest-attached-agent)*, we trained the behavior *ControlForage*, which directed agents to *Forage* until an agent found a box. Then, the controller would direct agents to *Goto(closest-attached-agent)*; once agents where close to the attached agent, they would grab the box and begin pulling it towards the deposit location. Once one agent finished pulling the box, the controller would direct the agents to *ReleaseBox*, and to resume *Forage.*

If the agents were acting on their own (they had no 2-level controller), their top-level behavior would be simply *ControlForage*. When acting under a 2-level controller, the current behavior of the 1-level controllers would be determined by their 2-level controllers.

**2-Level Controller Agent Behavior Decomposition** The 2-level controller's behavior hierarchy was as follows:

- A controller's basic features were *SomeoneNeedsHelp* and *SomeoneIsFinished*. The former feature is true if a subsidiary agent knows of a box which requires more agents to push it than are available to the subsidiary agent. A 2-level controller also had an additional target: *biggest-attached-agent*, which is the agent attached to the largest box (if any) in the controller's subgroup.
- A 2-level controller's basic behaviors corresponded to behaviors from its subsidiary 1-level controllers: *ControlForage*, *ReturnWithBox*, *Goto(X)*.

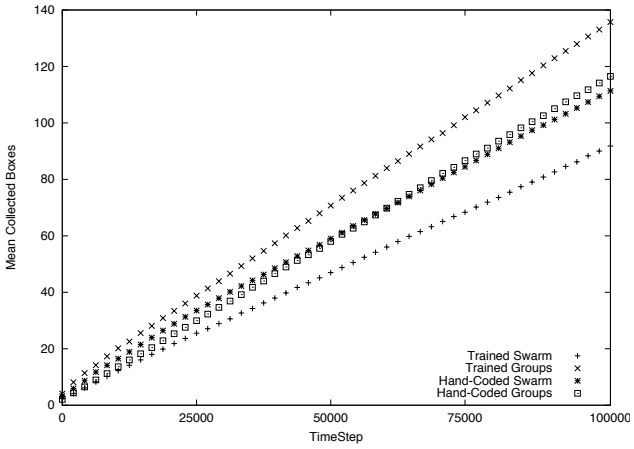


Figure 3: Mean number of boxes collected over time for the first experiment.

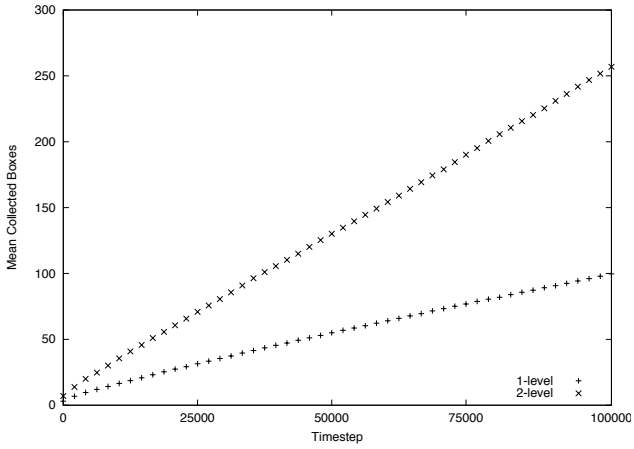


Figure 4: Mean number of boxes collected over time for the second experiment.

- We trained the behavior *ControlForage2*, which is similar to the *ControlForage* behavior. The difference is that the 2-level controller directs agents to *Goto(biggest-attached-agent)* when a 1-level controller requires help.

A 2-level controller's top behavior was *ControlForage2*.

## 4.1 Experiments

We considered three structures: (1) 50 independent agents (2) a *semi-distributed* group of 10 independent 1-level controller agents, each heading a five-agent subgroup (3) two independent 2-level controller agents, each heading five 1-level controller agents, each heading a five-agent subgroup. The idea behind these structures is illustrated in Figure 1.

We performed two experiments. In the first experiment, we first sought to demonstrate that a simple hierarchy can out-perform a group of independent agents; and second, that the behaviors learned in this experiment would perform adequately compared to fine-tuned hand-coded behaviors. In the second experiment, we sought to demonstrate that a two-layer hierarchy can outperform a one-layer hierarchy. Each experimental run lasted 100,000 timesteps, and each treatment consisted of 100 independent runs. Stated differences in results are measured at the 100,000 timestep with a 95% confidence, using two-tailed t-tests with a Bonferroni correction.

**First Experiment: 1-Level Hierarchies** Initially, we compared an entirely distributed group against a *semi-distributed* group, and also compared a trained behavior versus a hand-coded behavior. Figure 3 shows the results of all four experimental runs.

*Independent Agents Versus Hierarchies* We expected the controller agents to improve performance due to the semi-centralized coordination available, because the controller enabled specific groups of agents to work together on a single box. Without controller agents, agents could become stranded at boxes waiting for other agents to help pull. These waiting agents simply relied on random discovery of the box by other agents to gather enough helpers. Figure 3 verifies the expected improvement due to the controller agents, particularly when using trained behaviors rather than hand-coded ones. The improvement was statistically significant in both cases.

*Hand-Coded Versus Trained* We then compared the trained versions of the two previous structures with hand-coded versions of the same. Figure 3 again shows the results. We had expected the hand-coded solutions to perform better, since trained solutions contained significant training error. But in fact, in the 1-level hierarchy case the trained solution actually performed statistically significantly better than the hand-coded solution due to a more random exploration strategy, which allowed agents to disperse throughout the environment better. This exploration strategy didn't fare as well in the independent agents case, however: the hand-coded solution did statistically significantly better because its exploration strategy happened to result in more agents trying to pull the same box, rather than distributing across multiple boxes. While the results do not present a clear advantage to either training or programming, they do suggest that training the agents will crucially not *significantly* impair performance.

**Second Experiment: 2-Level Hierarchies** We then compared the same 1-level hierarchy as before against a two-level hierarchy: two 2-level controllers, each in charge of five 1-layer controllers, each in charge of five agents. We changed the scenario to favor two levels of coordination: the environment had eight boxes which each required five agents to pull, and two boxes which each required twenty-five agents to pull. Just as the first experiment was constructed so as to demonstrate the use of some degree of homogeneous coordination, the second experiment is meant to show the value of homogeneous coordination at two levels (5 agents or 25 agents per box).

As shown in Figure 4, two layers significantly outdo a single layer, and for similar reasons as the first experiment. If in the one-layer case a group discovers a 25-agent box, 4 other groups must randomly discover the box before it can be moved and all the groups freed. But with two layers of coordination, we can train agents to work together not only at in 5-agent groups but also in 25-agent groups. (Indeed we can train agents in any level of hierarchy.)

## 5 Future Work

**Multiagent Heterogeneity** Our ultimate goal is to demonstrate learned heterogeneous multiagent behaviors. To do this, we distinguish between homogeneous and heterogeneous controller agents. Our controller agents described so far are homogeneous. Heterogeneous controller agents might, at their top level, have behaviors which are pairs of automata. The controller would divide its subsidiary agents into two disjoint sets, and each automaton would control one set. Homogeneous controller agents might have heterogeneous subsidiaries and vice versa: though we expect the most common scenario would be a hierarchy of heterogeneous controller agents above a layer of homogeneous agents controlling the basic agents. We also imagine a dynamic agent hierarchy where basic and controller agents can change their parent based on higher level agent decisions.

**Behavior Revision** Our system is interactive in two senses: first, the user directly takes control of the agent, showing it how to perform a given behavior. Second, after training the behavior and observing its performance, the user may again take control and revise the behavior further. However at present once the behavior has been saved and used in a higher-level composed behavior, the user can no longer revise it. We have found that this presents difficulties for the experimenter: if he discovers an error in the behaviors, he must not only revise that behavior but also redo, from scratch, all behaviors which have formed compositions over it. We need to enable low-level behavior revision and examine the cascading effects that such revision has on all learned higher-level behaviors.

Our behavior training is also at present only *additive*: when the experimenter jumps in to revise a behavior after detecting an error, his revision adds new examples to the behavior but does not remove errant examples. We need to examine how to determine which examples caused the incorrect trained model, and whether to remove or modify them.

## 6 Conclusion

We demonstrated a supervised learning from demonstration system capable of learning stateful, recurrent multiagent behaviors in the form of probabilistic hierarchical finite-state automata, with an aim towards full heterogeneity. We do this by manually decomposing the task, learning simple behaviors, and using scaffolding to learn increasingly complex composed automata until we have achieved the behavior needed. This reduces the dimensionality of the learned problem, making possible more complex problems despite limited training examples. We also construct an agent hierarchy, which enables agents to work together as independent small groups as needed.

## References


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

[Chernova, 2009] Sonia Chernova. *Confidence-based Robot Policy Learning from Demonstration*. PhD thesis, Carnegie Mellon University, 2009.

[Dinerstein *et al.*, 2007] Jonathan Dinerstein, Parris K. Egbert, and Dan Ventura. Learning policies for embodied virtual agents through demonstration. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1257–1252, 2007.

[Goldberg and Mataric, 2002] Dani Goldberg and Maja J Mataric. Maximizing reward in a non-stationary mobile robot environment. *Autonomous Agents and Multi-Agent Systems*, 6:2003, 2002.

[Hovland *et al.*, 1996] G. E. Hovland, P. Sikka, and B. J. McCarragher. Skill acquisition from human demonstration using a hidden markov model. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2706–2711, 1996.

[Kasper *et al.*, 2001] Michael Kasper, Gernot Fricke, Katja Steuernagel, and Ewald von Puttkamer. A behavior-based mobile robot architecture for learning from demonstration. *Robotics and Autonomous Systems*, 34(2-3):153–164, 2001.

[Luke and Ziparo, 2010] Sean Luke and Vittorio Ziparo. Learn to behave! rapid training of behavior automata. In Marek Grześ and Matthew Taylor, editors, *Proceedings of Adaptive and Learning Agents Workshop at AAMAS 2010*, pages 61 – 68, 2010.

[Nakanishi *et al.*, 2004] Jun Nakanishi, Jun Morimoto, Gen Endo, Gordon Cheng, Stefan Schaal, and Mitsuo Kawato. Learning from demonstration and adaptation of biped locomotion. *Robotics and Autonomous Systems*, 47(2-3):79–91, 2004.

[Nicolescu and Mataric, 2002] Monica N. Nicolescu and Maja 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.

[Panait and Luke, 2005] Liviu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, 2005.

[Saunders *et al.*, 2006] Joe Saunders, Chrystopher L. Nehaniv, and Kerstin Dautenhahn. Teaching robots by moulding behavior and scaffolding the environment. In *Human-Robot Interaction*, pages 118–125, 2006.

[Stone and Veloso, 2000] Peter Stone and Manuela M. Veloso. Layered learning. In Ramon López de Mántaras and Enric Plaza, editors, *11th European Conference on Machine Learning (ECML)*, pages 369–381. Springer, 2000.

[Sullivan *et al.*, 2010] Keith Sullivan, Sean Luke, and Vittoria Amos Ziparo. Hierarchical learning from demonstration on humanoid robots. In *Proceedings of Humanoid Robots Learning from Human Interaction Workshop*, Nashville, TN, 2010.

[Veeraraghavan and Veloso, 2008] Harini Veeraraghavan and Manuela 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.