Basis Function Construction For Hierarchical Reinforcement Learning

Sarah Osentoski
Department of Computer Science
Brown University
115 Waterman St., 4th Flr
Providence, RI 02912
sosentos@cs.brown.edu

Sridhar Mahadevan
Department of Computer Science
University of Massachusetts, Amherst
140 Governor's Drive
Amherst, MA 01003
mahadeva@cs.umass.edu

ABSTRACT

Much past work on solving Markov decision processes (MDPs) using reinforcement learning (RL) has relied on combining parameter estimation methods with hand-designed function approximation architectures for representing value functions. Recently, there has been growing interest in a broader framework that combines representation discovery and control learning, where value functions are approximated using a linear combination of task-dependent basis functions learned during the course of solving a particular MDP. This paper introduces an approach to automatic basis function construction for hierarchical reinforcement learning (HRL). Our approach generalizes past work on basis construction to multi-level action hierarchies by forming a compressed representation of a semi-Markov decision process (SMDP) at multiple levels of temporal abstraction. The specific approach is based on hierarchical spectral analysis of graphs induced on an SMDP's state space from sample trajectories. We present experimental results on benchmark SMDPs, showing significant speedups when compared to hand-designed approximation architectures.

Categories and Subject Descriptors

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

General Terms

Algorithms, Experimentation

Keywords

Representation Discovery, Hierarchical Reinforcement Learning, Semi-Markov Decision Processes.

1. INTRODUCTION

Most successful applications of autonomous decision-making and learning models, such as Markov decision processes (MDPs) and reinforcement learning (RL), rely on hand-designed approximation architectures to compress the solution space to a low-dimensional subspace. Recently there has been growing interest in a broader framework combining representation discovery and control learning [7, 8, 10, 12], where parameter estimation methods are combined with feature construction techniques that automatically construct a function approximator during the course of solving a specific MDP. While promising results have been obtained in some benchmark MDPs, the larger question of how to scale these combined representation discovery and control learning methods remains open. Hierarchical reinforcement learning (HRL) approaches such as HAMs [11], options [13], and MAXQ [5] have been proposed to scale RL to large domains. HRL techniques are based on the semi-Markov decision process (SMDP) model, where an agent does not need to make decisions at each time step but instead can execute temporally-extended actions. This paper addresses the problem of scaling a unified representation discovery and control learning framework to large SMDPs, by exploiting the hierarchical structure of tasks. Past work on HRL has invariably assumed that value functions for each subtask are stored using table lookup. By combining methods for automatic basis construction with HRL methods, such as MAXQ, we show how a hierarchical function approximator can be automatically formed based on a specified task hierarchy.

Our approach constructs basis functions at multiple levels of abstraction. We focus on a spectral approach, in which basis functions are constructed by analyzing graphs induced on an MDP's state space from sample trajectories. In particular, basis functions correspond to the low-order eigenvectors of the graph Laplacian [8]. The graph Laplacian has been used extensively in machine learning, for problems ranging from dimensionality reduction to spectral clustering. The graph Laplacian approach to basis function construction uses a graph to reflect the topology of the state space of the underlying MDP. Spectral graph analysis is then used to construct compact representations of smooth functions on the underlying state space. We specifically address the problem of *automatic basis construction for SMDPs specified using multi-level task hierarchies*. Given an SMDP M and a task hierarchy H, the agent must automatically construct a low-dimensional representation Φ of M. The construction method should leverage H to create a compact representation Φ that respects the hierarchy. Φ should be constructed such that the solution to M calculated using Φ closely approximates the solution of the original SMDP M consistent with H.

2. HIERARCHICAL REINFORCEMENT LEARNING

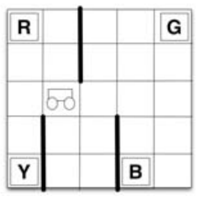
Hierarchical reinforcement learning is a sample-based framework for solving SMDPs that finds the "best" policy consistent with a given hierarchy [1]. These algorithms enable agents to construct hierarchical policies that allow using multi-step actions as "subroutines." HRL algorithms can be described using the SMDP framework. SMDPs are a generalization of MDPs in which actions are no longer assumed to take a single time step and may have varied

Cite as: Basis Function Construction For Hierarchical Reinforcement Learning, Sarah Osentoski and Sridhar Mahadevan, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. 747-754
Copyright © 2010, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

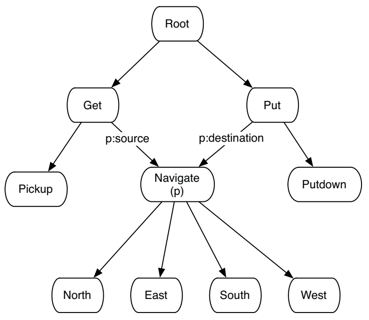
durations. An SMDP can be seen as representing the system at decision points, while an MDP represents the system at all times. An SMDP is defined as a tuple $M = (S, A, P, R)$. S is the set of states, and A is the set of actions the agent may take at a decision point. P is a transition probability function, where $P(s', N|s, a)$ denotes the probability that action a taken in state s will cause a transition to state s' in N time steps. Rewards can accumulate over the entire duration of an action. The reward function $R(s', N|s, a)$ is the expected reward received from selecting action a in state s and transitioning to state s' with a duration of N time steps.

2.1 An Example SMDP

The Taxi task [4], which we use as an example SMDP throughout this paper, is pictured in Figure 1a. The task is defined as a grid of 25 states with four colored locations, red (R), green (G), yellow (Y), and blue (B). The agent must pick up the passenger located on one of the colored locations and drop the passenger at the desired destination. Each state can be written as a vector of variables containing the location of the taxi, the passenger location, and the passenger destination. There are six primitive actions: *north*, *east*, *south*, *west*, *pickup*, and *putdown*. Each action receives a reward of -1. If the passenger is *putdown* at the intended destination, a reward of $+20$ is given. If the taxi executes *pickup* or *putdown* incorrectly, a reward of -10 is received. If the taxi runs into the wall, it remains in the same state.



(a) Taxi Domain.

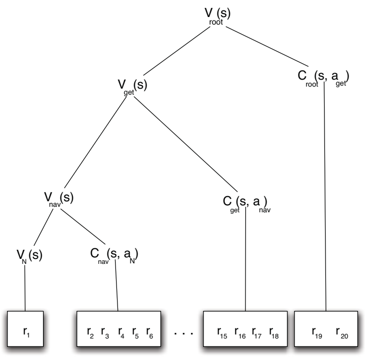


(b) Hierarchy for the Taxi Domain

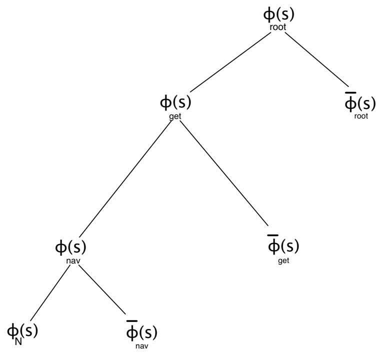
Figure 1: Taxi domain and the task hierarchy for this domain.

The task hierarchy is pictured in Figure 1b. The root node is defined over all states and decomposes into one of two subtasks, *get* and *put*. The *get* action can only be selected when the passenger is not located in the taxi and the *put* action can only be selected when the passenger is located in the taxi. No learning occurs at the root subtask because each state has only one abstract action available to it at any given time. The *get* action only considers the taxi location and the passenger location. It has access to two actions, *navigate(p)*, and *pickup*. The *put* action considers only the taxi location and the passenger destination, and has access to two actions: *navigate(p)*, and *putdown*. The *navigate* action takes as input a parameter p that indicates which of the 4 locations it can navigate to and has access to the 4 navigation actions.

One reason HRL is effective is that value functions can be decomposed using the task hierarchy. The intuition behind our approach to representation discovery for HRL problems is to construct hierarchical basis functions that decompose in an analogous manner. Figure 2a shows how the Q-value function decomposes in MAXQ into two parts: $V_a(s)$, the expected sum of rewards obtained for executing action a and the completion function $C_i(s, a)$, the expected cumulative reward for completing subtask i following the current policy π_i after action a is taken in state s. In order



(a) Value function decomposition using the task hierarchy for the Taxi task.



(b) Representation decomposition based on the task hierarchy for the Taxi task.

Figure 2: We explore an approach to basis function construction that exploits the value function decomposition defined by a fixed task hierarchy.

to scale, the representations created for HRL problems should decompose recursively in a similar manner. Lower level representations can be reused when constructing basis functions at a higher level. Figure 2b illustrates how the basis functions decompose for the Taxi task. For a subtask i, the basis functions for state s decompose into two parts: $\bar{\phi}_i(s)$, the "local" basis functions constructed at subtask i and $\phi_a(s)$, the basis functions from child subtasks, where a is one of the child subtasks. In this work, $\bar{\phi}_i(s)$ is automatically constructed using spectral analysis of a graph G_i that is built from the agent's experience for subtask i. However, other automatic basis function construction approaches, such as Bellman error basis functions [10], could be used.

2.2 Task Hierarchies for HRL

A task hierarchy decomposes an SMDP M into a set of subtasks $\{M_0, M_1, ..., M_n\}$, which can be modeled as "simpler" SMDPs. M_0 is the root subtask that solves M. A subtask is defined to be a tuple $M_i = (\beta_i, A_i, \tilde{R}_i)$. $\beta_i(s)$ is the termination predicate that partitions S into a set of active states S_i and a set of terminal states β_i. The policy π_i for M_i can only be executed if the current state $s \in S_i$. A_i is a set of actions that can be performed to achieve subtask M_i. Actions can either be a primitive action from A or another subtask. A subtask invoked from M_i is called a child of subtask i. No subtask can call itself either directly or indirectly. $\tilde{R}_i(s)$ is a deterministic "pseudo-reward" function specific to M_i.

Task hierarchies may also have parameterized subtasks. If M_j is a parameterized subtask, it is as if this task occurs many times in the action space A_i of the parent task M_i. Each parameter of M_j specifies a distinct task. β_i and $\tilde{R}_i$ are redefined as $\beta_i(s,p)$ and $\tilde{R}_i(s',p)$, where p is the parameter value. If a subtask's parameter has many values, it is the same as creating a large number of subtasks, which must all be learned. It also increases the size of A_i.

A hierarchical policy $\pi = \{\pi_0, \dots, \pi_m\}$ is a set containing a policy for each subtask in the task hierarchy. In each subtask, π_i takes a state and returns a primitive action or subtask to be executed. $P_i^{\pi}(s', N|s,a)$ is the probability transition function for a hierarchical policy at level i, where $s, s' \in S_i$ and $a \in A_i$.

2.3 Hierarchical State Abstraction

Task hierarchies allow state abstractions to occur through an abstraction function χ. Each state s can be written as a vector of variables X. X_i is the subset of state variables that are relevant to subtask i. $X_{i,j}$ is the jth variable for subtask i. A state $\boldsymbol{x}_i$ defines a value $x_{i,j} \in Dom(X_{i,j})$ for each variable $X_{i,j}$. χ_i maps a state s onto only the variables in X_i.

Abstractions allow the agent to use a *state-abstracted task hierarchy*. Given an MDP M and a task hierarchy H, the state variables for each subtask i can be partitioned into two sets X_i and Y_i, where Y_i is the set of state variables irrelevant to the task. χ_i projects s onto only the values of the variables in X_i. When combined with χ, H is called a state-abstracted task hierarchy.

A state-abstracted task hierarchy reduces the size of the learning problem because an *abstract hierarchical policy* can be defined over the reduced space. For an MDP M with a state-abstracted task hierarchy H and χ, an abstract hierarchical policy is a hierarchical policy in which each subtask i has a policy π_i that satisfies the following condition: for any two states s_1 and s_2 such that $\chi_i(s_1) = \chi_i(s_2)$ then $\pi_i(s_1) = \pi_i(s_2)$.

2.4 Solving HRL tasks

Each subtask M_i has a value function $Q_i(s,a)$ that defines the value of taking an action a in state s according to the real reward function R. $Q_i(s,a)$ is used to derive a policy π_i, typically by selecting the action with the maximum Q value for s.

In the MAXQ framework [4] the value function is decomposed based upon the hierarchy. MAXQ defines Q_i recursively as
$Q_i(s,a) = V_a(s) + C_i(s,a)$ where

$$V_i(s) = \begin{cases} \max_a Q_i(s,a) & \text{if } i \text{ is composite} \\ V_i(s) & \text{if } i \text{ is primitive.} \end{cases}$$

$V_a(s)$ is the expected sum of rewards obtained while executing action a. The completion function $C_i(s,a)$ is the expected discounted cumulative reward for subtask i following the current policy π_i after action a is taken in state s. $\tilde{C}$ is the completion function that incorporates both $\tilde{R}_i$ and R, and is used only inside the subtask to calculate the optimal policy of subtask i. $\tilde{Q}_i$ is used to select the action and defined as $\tilde{Q}_i(s,a) = V_a(s) + \tilde{C}_i(s,a)$. If $\tilde{R}_i$ is zero, then C and $\tilde{C}$ will be identical.

2.4.1 Function Approximation for HRL

When performing function approximation in HRL, each subtask has a set of basis functions Φ_i and a set of weights $\boldsymbol{\theta}_i$ that are used to calculate the value function Q. $\phi_i(s,a)$ is a k length feature vector for state s and action a. In MAXQ, the completion function for subtask i at time t, $C_{i,t}(s,a)$, is approximated by $\hat{C}_{i,t}(s,a) = \sum_{j=1}^{k} \phi_{i,j}(s,a)\boldsymbol{\theta}_{i,j,t}$. The update rule for the parameters is given as $\boldsymbol{\theta}_{i,(t+N)} = \boldsymbol{\theta}_{i,t} + \alpha_i[\gamma^N(\max_{a' \in A(s')} \hat{C}_{i,t}(s', a'|\boldsymbol{\theta}_{i,t}) + V_{a',t}(s')) - \hat{C}_{i,t}(s,a|\boldsymbol{\theta}_{i,t})] \cdot \phi_i(s,a)$.

In our experiments we use Q(λ) learning with replacing traces. The update rules for this are: $\boldsymbol{\theta}_{i,(t+N)} = \boldsymbol{\theta}_{i,t} + \alpha\delta_{i,t}\boldsymbol{e}_{i,t}$where $\boldsymbol{e}_{i,t} = \gamma^N\lambda\boldsymbol{e}_{i,t-N} + \phi_i(s,a), \boldsymbol{e}_0 = \mathbf{0}$ and

$$\delta_{i,t} = \gamma^N(\max_{a' \in A(s')} \hat{C}_{i,t}(s',a'|\boldsymbol{\theta}_{i,t}) + V_{a',t}(s')) - \hat{C}_{i,t}(s,a|\boldsymbol{\theta}_{i,t}) \quad (1)$$

$$\tilde{\delta}_{i,t} = \gamma^N(\tilde{R}(s,a) + \max_{a' \in A(s')} \hat{\tilde{C}}_{i,t}(s',a'|\tilde{\boldsymbol{\theta}}_{i,t}) + V_{a',t}(s')) - \hat{\tilde{C}}_{i,t}(s,a|\tilde{\boldsymbol{\theta}}_{i,t}).$$

3. AUTOMATIC BASIS FUNCTION CONSTRUCTION FOR HRL

Our approach to automatic basis function construction for multilevel task hierarchies uses the graph Laplacian approach [8]. In this approach the agent constructs basis functions by first exploring the environment and collecting a set of samples. The samples are used to create a graph where the vertices are states and edges are actions. Basis functions are created by calculating the eigenvectors of the Laplacian of the graph. We extend this approach to multi-level task hierarchies.

Our approach is divided into three steps. The first step constructs a graph for each subtask from the agent's experience. We show how graph construction can leverage the abstractions provided with the task hierarchy. The second step constructs a reduced graph based upon the graph structure. The third step recursively constructs basis functions using basis functions from child subtasks as well as using the eigenvectors of the graph Laplacian from the reduced graph. Algorithm 1 describes the overall algorithm for combining HRL with representation discovery.

Algorithm 1 HRL-RD (Subtask i, State s, Initial Samples $\mathcal{D}$, Number of basis functions k_i, Initial Policy $\pi_{0,i}, \gamma, \lambda, \epsilon$)

```
if $i$ is a primitive subtask then
    execute $i$, receive $r$, and observe the next state $s'$
    $V_{t+1,i}(s) := (1 - \alpha_t(i)) \cdot V_t(i,s) + \alpha_i(t) \cdot r_t$
    RETURN $s'$, 1
else
    if first time executing $i$ then
        Call BasisConstruction$(i, \mathcal{D}, k_i, \pi_{0,i})$ in Algorithm 2
    end if
    $\vec{e} = 0, \bar{N} = 0$
    while $\beta_i(s)$ is false do
        $a^* = \text{argmax}_{a'}[\tilde{Q}_{i,t}(s', a'|\theta_{i,t})]$
        choose an action $a$ according to the current policy $\pi_i$
        if $a = a^*$ then
            $\boldsymbol{e}_i = \gamma^N\lambda\boldsymbol{e}_i$
        else
            $\boldsymbol{e}_i = \mathbf{0}$
        end if
        $\boldsymbol{e}_i = \boldsymbol{e}_i + \phi_i(s,a)$
        $(s', N) =$ HRL-RD$(a, s)$
        Use update rules from Equation 1
        $\theta_{i,(t+N)} = \theta_{i,t} + \alpha_i\delta\boldsymbol{e}_i$
        $\tilde{\theta}_{i,(t+N)} = \tilde{\theta}_{i,t} + \alpha\tilde{\delta}\boldsymbol{e}_i$
        $s = s'$
        $\bar{N} = \bar{N} + N$
        return $s', \bar{N}$
    end while
end if
```

3.1 Graph Creation for Task Hierarchies

The first step in our approach to representation discovery for multi-level task hierarchies is to perform sample collection, such that each subtask i has a set of samples $\mathcal{D}_i$. Each sample in $\mathcal{D}_i$ consists of a state, action, reward, and next state, (s, a, r, s'). The agent constructs a graph from $\mathcal{D}_i$. The agent can leverage a state-abstracted task hierarchy by building the graph in the abstract space defined by χ_i. The graph is built such that $\chi_i(s_1)$ is connected to $\chi_i(s_2)$, if the agent experienced a transition from $\chi_i(s_1)$ to $\chi_i(s_2)$ in $\mathcal{D}_i$. We call a graph constructed over the abstract state space a state-abstracted graph. Figure 2 describes how a graph can be created; this approach is similar to the approach in Osentoski and Mahadevan [9] but uses the abstraction function χ.

For an MDP M with a state-abstracted task hierarchy, a *state-abstracted graph* G_i can be constructed for subtask i over the abstract state space defined by χ_i. The vertices V correspond to the set, or subset, of abstract states $\chi_i(S)$. An edge exists between v_1 and v_2 if there is an action $a \in A_i$ that causes a transition between the corresponding abstract states.

Algorithm 2 BasisConstruction (Subtask i, Samples $\mathcal{D}$, Number of local basis functions k_i, Initial policy π_0)

```
**Sample Collection:**
    1. **Exploration:** Generate a set of samples $\mathcal{D}_i$ which consists of a state, action, reward, and nextstate, $(s, a, r, s', N)$ for subtask, $i$ according to $\pi_0$. $N$ is the number of steps $a$ took to complete.
    2. **Subsampling Step (optional):** Form a subset of samples $\mathcal{D}_i \in \mathcal{D}$ by some subsampling method.
**Representation Learning:**
    1. Build an graph $\mathcal{G}_i = (\mathcal{V}, \mathcal{E}, \mathcal{W})$ from $\mathcal{D}_i$ where state $i$ is connected to state $j$ if $i$ and $j$ are linked temporally in $\mathcal{D}$ and $\mathcal{W}(i, j) = a_i$ where $a_i$ is the action that caused the transition from $i$ to $j$ in $\mathcal{D}$.
    2. $G_i$=GraphReduction($\mathcal{G}_i$) in Algorithm 3.
```

3.1.1 State-abstracted graph for the Get Task

Figure 3 shows the state-abstracted graph for the *get* task. $\chi_{get}(s)$ maps each state s to an abstract state $\boldsymbol{x}_{get} \in X_{get}$ where $X_{get} = \{$*passenger position, taxi position*$\}$. Each vertex in Figure 3 is an abstract state $\boldsymbol{x}_{get}$. The four clusters of vertices correspond to a clustering of the states according to their values for the passenger location. Within each cluster, the darker vertices correspond to states where the taxi is located on one of the colored grid states. The light vertices in the graph are not connected to one another but only to the dark colored vertices. This is because the *get* subtask can only execute the *navigate* and *pickup* actions, and the *navigate* action leads to one of the four colored grid states. Light edges are caused by the *navigate* subtask. Dark edges are caused by primitive actions, in this case the *pickup* action. Each cluster has only one vertex with a dark edge. This vertex represents the state where the taxi is located in the same state as the passenger location. The center vertex represents the terminal state where the passenger is no longer in the taxi.

3.1.2 Building a Reduced Graph

We describe how state abstractions can be created using a graph reduction algorithm. The approach uses only properties of the graph to construct the abstraction. Our approach to graph reduction requires that the original graph $\mathcal{G}_i$ be an edge labeled graph. We define an edge labeled graph to be $G = (V, E, Z, W)$, where V is the set of vertices, E is the edge set, Z is a set of labels over E, and W is the weight matrix. $\mathcal{G}_i$ must be constructed such that $Z(v_1, v_2)$ corresponds to the action a that caused the transition between v_1 and v_2. $\mathcal{G}_i$ may be a state or state-abstracted graph.

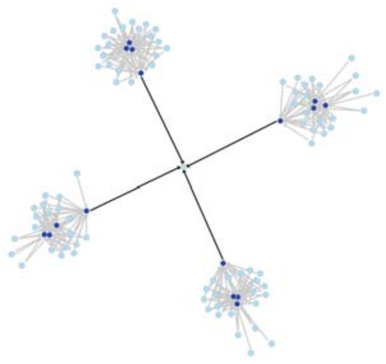


Figure 3: State-abstracted graph of the *get* subtask.

A *reduced graph* can be constructed for subtask i from a graph $\mathcal{G}_i$. Two vertices v_1 and v_2 correspond to states, or abstract states, s_1 and s_2. v_1 and v_2 can be represented as the same abstract vertex $\tilde{v}$, if the state variables for M_i can be divided into two groups X_i and Y_i such that:

- s_1 and s_2 differ only in their values of Y_i
- v_1 and v_2 are connected to the same set of vertices in the graph and the labels $z \in Z$ for the respective edges are the same.

v_1 and v_2 are merged into an abstract vertex $\tilde{v}$ corresponding to the subset of state variables X_i. The graph reduction algorithm creates a reduced graph if M does not have an abstraction function χ associated with H or if χ exists but the state-abstracted graph G_i can be further compressed. If no nodes are merged, the resulting graph will be the original graph. Algorithm 3 contains the algorithm used to transform the state graph into the reduced graph and create basis functions from the reduced graph.

Algorithm 3 GraphReduction ($\mathcal{G}_o = (\mathcal{V}_o, \mathcal{E}_o, \mathcal{Z}_o, \mathcal{W}_o)$)

```
$V_i = \mathcal{V}_o$
**for** all $v_1 \in V_i$ **do**
    **for** all $v_2 \in V_i$ **do**
        $V_1$ is the set of vertices such that $v' \in V_1 \implies v_1 \rightarrow v'$
        $V_2$ is the set of vertices such that $v' \in V_2 \implies v_2 \rightarrow v'$
        **if** $V_1 = V_2$ **and** $s_1 = (\mathbf{x}_i, \mathbf{y}_1)$ **and** $s_2 = (\mathbf{x}_i, \mathbf{y}_2)$ **and** $\forall v_1 \in V_1$ and $v_2 \in V_2$ $\mathcal{Z}_o(v_1, v') = \mathcal{Z}_o(v_2, v')$ **then**
            Merge $v_1$ and $v_2$ into an abstract node $\tilde{v}$ corresponding to the state variables $X_i$
            $\forall v' \in V_1$ $W_i(\tilde{v}, v') = 1$
        **end if**
    **end for**
**end for**
return $G_i$
```

3.1.3 Reduced graph for the Get Task

Figure 4 shows the reduced graphs for each subtask of the Taxi task. The reduced graph for the *Get* task has nine vertices. The outer four vertices are abstract nodes corresponding to states where the taxi is not in one of the colored grid locations. The four inner states correspond to the bottleneck states when the agent is in the same location as the passenger. The center state represents when the passenger has been picked up and is in the taxi.

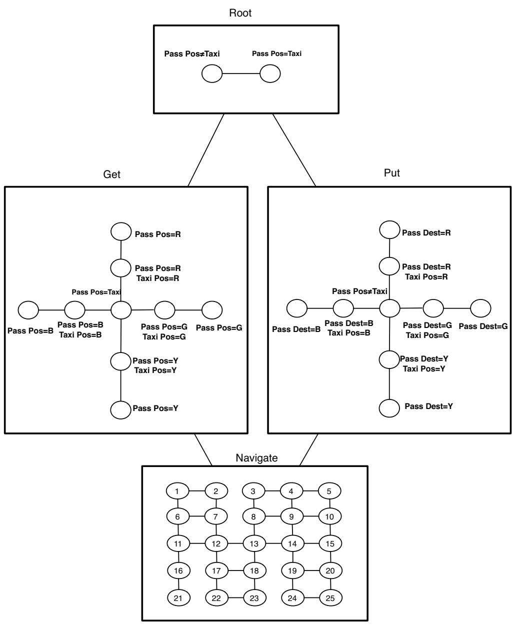


Figure 4: The reduced graphs for the Taxi task.

3.1.4 Generating Hierarchical Basis Functions

The basis functions for a subtask i are automatically constructed by first generating the *local basis functions* $\bar{\Phi}_i$. $\bar{\Phi}_i$ is constructed from the eigenvectors of the graph Laplacian of G_i. A general overview of spectral decomposition of the Laplacian on undirected graphs can be found in [2]. A weighted undirected graph is defined as $G_u = (V, E_u, W)$ where V is a set of vertices, E_u is a set of edges, and W is the set of weights w_{ij} for each edge $(i,j) \in E_u$. If an edge does not exist between two vertices it is given a weight of 0. The valency matrix, D, is a diagonal matrix whose values are the row sums of W. The combinatorial Laplacian is defined as $L_u = D - W$ and the normalized Laplacian is defined as $\mathscr{L}_u = D^{-\frac{1}{2}}(D-W)D^{-\frac{1}{2}}$. The set of basis functions, Φ, are the lowest-order k eigenvectors associated with the smallest eigenvalues of either L_u or $\mathscr{L}_u$. $\phi(s)$ is the *embedding* of state s defined by the eigenvectors. These basis functions are concatenated with basis functions recursively gathered from the child subtasks. This means that the basis functions are no longer guaranteed to be linearly independent. If necessary, the bases can be reorthogonalized using Gram-Schmidt or QR decomposition. Figure 5 shows the second eigenvector of the combinatorial graph Laplacian for the reduced graph of each subtask. The black horizontal lines display the reduced graph. The blue vertical lines show the value of the eigenvector for the particular vertex in the graph.

We define φ to be a compression of the state space. Compressions can be abstractions defined by χ, such as those proposed by Dietterich [5] as well as those from the reduced graph. Compressions can also be defined through spectral graph analysis. For a given subtask i, we define φ_{χ_i} as the compression given by the abstraction function χ_i, φ_G is the compression created by the reduced graph, and φ_e are the eigenvectors of the graph Laplacian. The basis functions $\phi_i(s)$ for subtask i and a state s can be written as the concatenation of the local basis functions with the basis functions from the child subtasks:

$$\phi_i(s) = [\varphi_e(\varphi_G(\varphi_\chi(s))) \mid \phi_a(s) \, \forall a \in A_i(s)],$$

where $a \in A_i(s)$ is not a primitive action. The embedding of state s for the *get* subtask has two parts: the local embedding from the *get* subtask and the embedding from the *navigate* subtask. The local embedding of a state s for a subtask is the row of the eigenvector matrix that corresponds to the abstract state of s.

Since our approach uses reward independent basis functions, the basis functions from parameterized tasks are only used once. This approach allows methods, such as graph Laplacian basis functions, to scale to larger domains. The reduced graph can greatly reduce the size of the eigen problem that must be solved to create these basis functions.

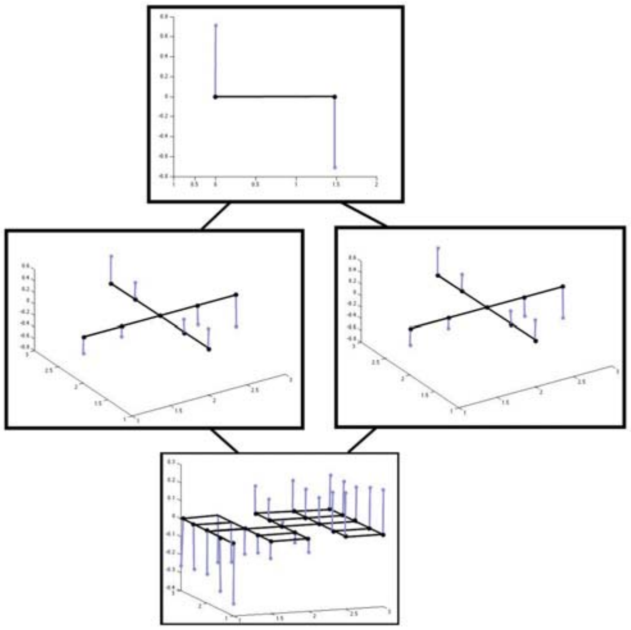


Figure 5: Examples of the basis functions for each level of the task hierarchy. The basis functions pictured are the 2nd eigenvector of the combinatorial graph Laplacian.

Earlier we gave a generic description of how basis function decomposition might occur for the Taxi task. Figure 6 shows the actual decomposition of our recursive basis function construction approach. For a subtask i the basis functions $\phi_i(s)$ are composed of two parts: $\bar{\phi}_{G_i}(s)$ are the basis functions constructed from the reduced graph G_i and $\phi_a(s)$ the basis functions from all of the unique child subtasks $a \in A_i(s)$. For example, the basis functions for the *get* task are constructed from the basis functions from the reduced graph G_{get} and the basis functions from the *navigate* subtask. For the *root* subtask, the basis functions are constructed from the reduced graph G_{root}, the basis functions from the *get* subtask, and the basis functions from the *put* subtask. The basis functions from the *navigate* subtask could potentially be used twice by the *navigate* subtask. Both the *get* and *put* subtasks construct their basis functions using the basis functions from the *navigate* subtask. However, the "extra" set of *navigate* basis functions provide no additional information and can be omitted.

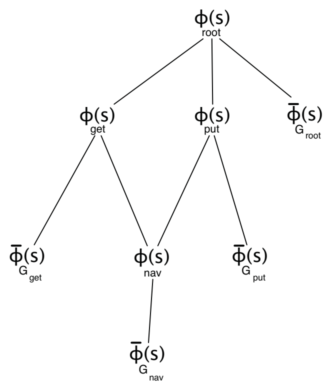


Figure 6: The recursive basis function decomposition from our proposed approach.

Osentoski and Mahadevan [9] demonstrated that constructing basis functions directly in state-action space can significantly speed up learning. Since actions at a lower level are not available at a higher level, recursively generating state-action basis functions is not necessarily straightforward. Thus, our recursive basis function approach constructs basis functions over the state space.

4. THREE TYPES OF ABSTRACTION

In this section, we analyze our approach to basis function construction for HRL. We start by examining the abstractions created using the reduced graph approach and demonstrate that our approach is capable of finding abstractions similar to three types of abstraction outlined by Dietterich [5].

The first type of abstraction, *subtask irrelevance*, involves eliminating state variables that are irrelevant to a subtask and thus play no role in the transition probability function or the reward function for that subtask. The reduced graph construction algorithm constructs a reduced graph containing this abstraction. If the state variables in Y_i have no bearing on the probability transition function, they will be irrelevant in terms of connectivity on the graph and only X_i will be used to represent the state variables.

The second type of abstraction, *shielding*, results from the structure of the hierarchy. The value of s does not need to be represented for a subtask M_i, if for all paths from the root of the hierarchy H to subtask i there is some subtask j whose termination predicate $\beta_j(s)$ is true. Our approach automatically finds this representation because the graph is constructed over states in the set of samples $\mathcal{D}_i$ collected during the agent's initial exploratory period. $\beta_j(s)$ causes j to terminate and j lies on all paths between subtask i and the root. Thus, $\mathcal{D}_i$ cannot contain s, because the agent cannot transition to s during the execution of this subtask. Therefore, the graph will not include s, and s will not be represented in the basis functions.

The third type of abstraction results from "funnel actions", specifically the *result distribution irrelevance* condition [5]. For a given subtask i, result distribution irrelevance constructs an abstraction for all pairs of states s_1 and s_2, where the state variables can be partitioned into two sets $\{X_j, Y_j\}$, such that s_1 and s_2 only differ in their values of Y_j. The completion function for subtask i can be represented as an abstract completion function $C_i^\pi(\mathbf{x}_j, j)$, if the subset of state variables Y_j are irrelevant for the result distribution of child subtask j. Y_j is irrelevant for the result distribution of subtask j, if $P^\pi(s', N|s_1, j) = P^\pi(s', N|s_2, j), \forall s'$ and N.

Result distribution irrelevance is an abstraction over state-action pairs. The graph reduction algorithm creates an abstract state for states s_1 and s_2 when $A_i(s_1) = A_i(s_2)$ and the state variables Y_i are irrelevant to connectivity of s to next state vertices s' for all $a \in A_i(s_1)$. The reduced graph abstraction differs from result distribution irrelevance because it requires the constraint to be true for all available actions. Additionally, it does not require the probabilities to be identical, just the connectivity within the graph.

In general, abstractions formed by the graph reduction algorithm are no longer "safe" state abstractions. The graph reduction algorithm uses connectivity within the graph rather than probabilities. This may lead to abstractions that "overgeneralize". For example, if $P^\pi(s', N|s_1, j)$ is slightly different than $P^\pi(s', N|s_2, j)$ but both values are greater than zero, then both s_1 and s_2 could potentially be collapsed into the same abstract vertex. Additionally, the reductions created by graph reduction algorithm construct "funnel action" abstractions, which are unsafe in the discounted reward setting [5]. However, information is regained when basis functions from child subtasks are used in constructing basis functions. This information regains some of the "lost" information and allows the agent to learn appropriate policies.

5. EXPERIMENTAL ANALYSIS

In this section, we experimentally evaluate the approach and compare it to other techniques.

5.1 Taxi

We evaluated four different techniques on the Taxi task: hierarchical Laplacian basis functions (HLBFs), graph Laplacian basis functions using the more traditional approach, table-lookup and radial basis functions (RBFs) where the j-th basis function for level i is defined as $\phi_{i,j}(s) = \exp(-(s - \mu_{ij})^T \Sigma_{i,j}^{-1}(s - \mu_{i,j}))$. μ_{ij} is the center of the j-th Gaussian kernel of level i and $\Sigma_{i,j}$ is the width of that Gaussian. The results can be seen in Figure 7. The results of each experiment are averaged over 30 trials. The results plot the cumulative reward received by the agent.

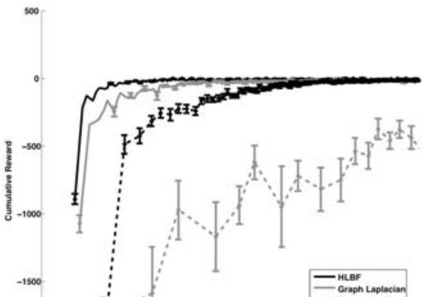


Figure 7: Results for the Taxi domain

The function approximation techniques all use a similar number of basis functions. HLBFs were created using the approach described in this paper. The results use the normalized graph Laplacian. The HLBF approach used ten local basis functions for the *navigate* subtask, nine basis functions for *get*, and seven basis functions for *put*.

The basis functions of the graph Laplacian approach were created by using the eigenvectors of the *directed* graph Laplacian [3] of the state-abstracted graph. This corresponds to the approach to constructing state-space basis functions for SMDPs described in [9] except we allowed the approach to leverage the abstractions provided by the task hierarchy. Ten basis functions were used for all of the subtasks. It is important to note that while a similar number of basis functions were used for both of the graph based approaches the smaller size of the graph in the HLBF approach significantly reduces the amount of effort required to calculate the eigenvectors. The recursive approach also uses basis functions from lower levels in order to obtain a better approximation.

The *navigate* subtask had a total of 17 basis functions created by uniformly placing the RBFs with two states between each RBF. The *get* and *put* subtasks had 21 basis functions created by placing the RBFs uniformly with five states between each RBF. We experimented with different numbers of RBFs but even doubling the number of basis functions did not greatly improve performance. Table 1 lists the number of basis functions used in the experiments for the taxi experiments. As can be seen in Figure 7, the HLBF approach and graph Laplacian approach perform significantly better than the traditional approaches. Additionally, the HLBF approach performed better than the graph Laplacian approach and required solving a smaller eigenproblem.

	Navigate	Get	Pickup
HLBF Construction	10 (local)	9 (local) 19 total	7 (local) 17 total
Graph Laplacian	10	10	10
Table Look Up	25	101	101
Radial Basis Functions	17	21	21

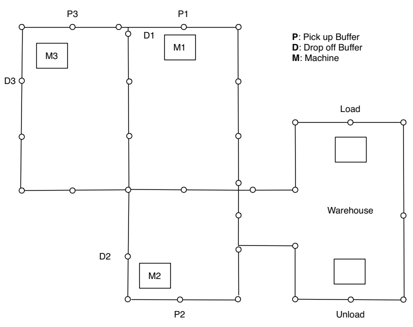
Table 1: Number of basis functions used in the taxi experiments

5.2 Manufacturing Domain

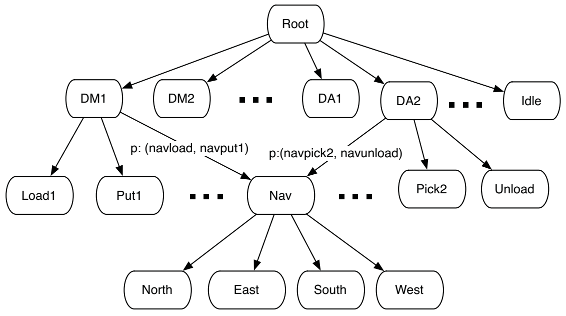
We also evaluated our approach on a simulated manufacturing shown in Figure 8a. This domain models a manufacturing environment and is a modified version of the domain in [6]. The agent travels between 33 locations. $M1 - M3$ are workstations. The agent carries one part at a time to workstation drop off buffers $D1 - D3$ and the assembled parts are brought from the workstation pick up buffers, $P1 - P3$, to the warehouse. A reward of -5 is given when the actions Put1-Put3, Pick1-Pick3, Load1-Load3, and Unload actions are executed illegally. All other actions receive a reward of -1. The task is complete when the agent drops off one of each type of assembled part at the warehouse and a reward of 100 is given.

The factored state consists of the number of parts in the pickup and drop off buffers, if the warehouse contains the three types of parts, the agent's location, the agent's status, and if each assembled part has been delivered. The flat representation of the state space consists of 33 locations, 6 buffers of size 2, 7 possible states of the agent, 2 values for each part in the loading area of the warehouse, and 2 values for each assembled part in the unloading area of the warehouse. This gives a total of $33 \times 3^6 \times 7 \times 2^3 \times 2^3 = 10,777,536$ states. There are 14 primitive actions: North, South, East, West, Put1-Put3, Pick1-Pick3, Load1-Load3, Unload, and Idle. The total number of parameters that must be learned in the flat case is $10,777,536 \times 14 = 161,663,040$.

Figure 8b defines a task hierarchy. The Nav subtask moves the



(a) Manufacturing Domain



(b) Hierarchy for the Manufacturing Domain

Figure 8: Manufacturing Domain and Task Hierarchy

agent throughout the grid. DM1-DM3 tasks require picking up a part from the warehouse and delivering it to the respective machine. DA1-DA3 tasks involve picking up an assembled part from the correct machine and delivering it to the warehouse.

We evaluated the recursive basis function approach and compared it to table look up on this task. We terminated learning after 3000 primitive steps were taken in the domain. Our results use the normalized graph Laplacian. The recursive basis function approach created 15 local basis functions for the Navigate subtask, 10 basis functions for subtasks: DM1-DM3, and DA1 - DA3. The root subtask has 400 local basis functions. Learning results are shown in Figure 9. The results of each experiment were averaged over 30 trials. Experiments using the HLBF approach begin to converge significantly faster than the table lookup approach. The HLBF approach begins to converge at about 50,000 time steps while the table look up approach begins to converge at about 100,000 time steps. Additionally, the learning results for experiments using the HLBF approach are smoother because the basis functions allow the agent to generalize when it encounters previously unseen states. RBF and graph Laplacian approaches are not visualized because they did not preform well in this large domain.

5.3 Discussion of Results

This paper showed how task hierarchies can enable scaling automatic basis function construction methods to large SMDPs. In addition, automatically constructing basis functions significantly improves the speed of convergence of HRL methods such as MAXQ. Basis functions provide generalization over the state space of each subtask allowing the agent to transfer learning across similar states. Our HLBF approach has several advantages that help its performance. The reduced graph has significantly fewer states and thus the agent needs to learn fewer values. This is also beneficial for basis function construction since the size of each basis function

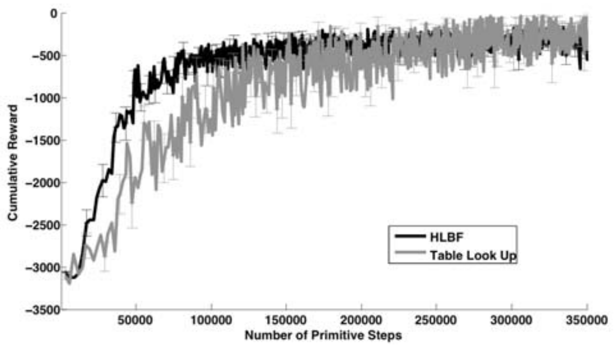


Figure 9: Results for the manufacturing domain

(eigenvector) as well as the time required to compute eigenvectors is significantly reduced.

The HLBF approach is also helpful when the state space is not fully sampled during basis function construction and out of sample extension [14] must be used. Out of sample extension techniques perform best when there are states in the graph that are similar to the new previously unseen state. They also require an accurate distance metric to link previously unseen states to states in the graph. One of the common properties of task hierarchies is that lower level subtasks are defined over a subset of the state variables. This means that while the agent may not have observed the state at a higher level subtask, lower level subtasks are likely to have a representation for the state.

Another benefit of using the reduced graph is that basis functions for higher level subtasks are smoother. Due to the properties of the graphs at more abstract levels and the symmetrization approach, the eigenvectors are often highly localized to a few states even for low order eigenvectors. Thus a significant number of eigenvectors were required for learning. Eigenvectors created from the reduced graph are often significantly smoother since many of the vertices are merged, which makes them more useful for approximating the value function.

6. CONCLUSION AND FUTURE WORK

This paper introduced a new hybrid framework that combines hierarchical learning of representation and control in SMDPs. Specifically, an integrated framework for combining automatic basis function construction from sample trajectories with a hierarchical reinforcement learning method was presented. Basis functions are recursively constructed by hierarchical spectral analysis of a multilevel graph induced from an SMDP's state space, where the task hierarchy is used to form successively more abstract graphs at higher levels of temporal abstraction. Basis functions are then constructed by combining eigenvectors of the reduced graph for the subtask as well as basis functions for lower level child subtasks. We evaluated this approach experimentally and demonstrated that automatic basis function construction can significantly improve the speed of learning when compared to traditional function approximation techniques as well as over exact methods.

There are several avenues for future work. One avenue is to extend this approach to state-action space. A second avenue is to examine skill transfer especially when the agent has access to automatically constructed basis functions. Additionally we plan to extend this work to the multiagent HRL, where the agents could benefit from sharing learned representations. Another is to examine multi-scale representations such as diffusion wavelets to automatically construct a hierarchy of basis functions. This approach may also lead to new insights and new types of skill learning.

7. ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation under grants NSF IIS-0534999 and NSF IIS-0803288. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We thank the members of the Autonomous Learning Lab for their comments and assistance.

8. REFERENCES

[1] A. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Special Issue on Reinforcement Learning, Discrete Event Systems Jouranl*, 13:41–77, 2003.

[2] F. Chung. *Spectral Graph Theory*. Number 92 in CBMS Regional Conference Series in Mathematics. American Mathematical Society, 1997.

[3] F. Chung. Laplacians and the Cheeger inequality for directed graphs. *Annals of Combinatorics*, 9:1–19, 2005.

[4] T. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 118–126. Morgan Kaufman, 1998.

[5] T. Dietterich. Hierarchical reinforcemnt learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:277–303, 2000.

[6] M. Ghavamzadeh and S. Mahadevan. Hierarchical average-reward reinforcement learning. *Journal of Machine Learning Research*, 8:2629–2669, 2007.

[7] P. W. Keller, S. Mannor, and D. Precup. Automatic basis function construction for approximate dynamic programming and reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning*, New York, NY, 2006. ACM Press.

[8] S. Mahadevan. Proto-Value Functions: Developmental Reinforcement Learning. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 553–560, New York, NY, 2005. ACM Press.

[9] S. Osentoski and S. Mahadevan. Learning state-action basis functions for hierarchical MDPs. In *Proceedings of the 24th International Conference on Machine Learning*, 2007.

[10] R. Parr, C. Painter-Wakefield, L. Li, and M. Littman. Analyzing feature generation for value-function approximation. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2007.

[11] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10*, pages 1043–1049. MIT Press, 1998.

[12] M. Petrik. An analysis of Laplacian methods for value function approximation in MDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.

[13] R. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

[14] C. K. I. Williams and M. Seeger. Using the Nyström method to speed up kernel machines. In *Advances in Neural Information Processing Systems 13*, 2001.