Coarticulation: An Approach for Generating Concurrent Plans in Markov Decision Processes

Khashayar Rohanimanesh KHASH@CS.UMASS.EDU
Sridhar Mahadevan MAHADEVA@CS.UMASS.EDU
Department of Computer Science, University of Massachusetts, Amherst, MA 01003

Abstract

We study an approach for performing concurrent activities in Markov decision processes (MDPs) based on the coarticulation framework. We assume that the agent has multiple degrees of freedom (DOF) in the action space which enables it to perform activities simultaneously. We demonstrate that one natural way for generating concurrency in the system is by coarticulating among the set of learned activities available to the agent. In general due to the multiple DOF in the system, often there exists a redundant set of admissible sub-optimal policies associated with each learned activity. Such flexibility enables the agent to concurrently commit to several subgoals according to their priority levels, given a new task defined in terms of a set of prioritized subgoals . We present efficient approximate algorithms for computing such policies and for generating concurrent plans. We also evaluate our approach in a simulated domain.

1. Introduction

Every day in our life we constantly perform concurrent activities. By exploiting many degrees of freedom (DOF) in our body (e.g., arms, legs, eyes, etc), we are able to simultaneously commit to several tasks and as a result generate concurrent plans. As an example consider a driving task which may involve subgoals such as safely navigating the car, talking on the cell phone, and drinking coffee, with the first subgoal taking precedence over the others. Having the benefit of extra DOF in our body, we are able to simultaneously commit to multiple subgoals. For example we can control the wheels by the left arm and use the right arm to answer the cell phone, or drink coffee. In general concurrent decision making is a challenging problem, since subgoals often have conflicting objectives and compete for the limited DOF in the system. However, the main challenge with this problem is the combinatorial space of possible concurrent actions that includes every possible combination of control signals (e.g., primitive actions in MDPs) for controlling the DOF in the system.

In this paper we study an approach for generating concurrent plans based on the coarticulation framework introduced in (Rohanimanesh et al., 2004). We demonstrate how this approach can cope with the curse of dimensionality incurred in systems with excess degrees of freedom, and that it can be viewed as one natural way for generating concurrent plans. The key idea is that for many goal-oriented activities – in addition to the optimal policy – often there exists a set of *ascending* (and possibly sub-optimal) policies that guarantee achieving the goal with a cost of a slight deviation from optimality. Such flexibility enables the agent to simultaneously commit to multiple subgoals.

We argue that coarticulation is a natural way for generating parallel execution plans for several reasons. First, many concurrent decision making problems can be actually viewed as concurrent optimization of a set of prioritized subgoals, in which the agent manages its DOF to simultaneously commit to them. Second, because of the multiple DOF in the system, learned activities offer more flexibility in terms of the range of ascending policies associated with them. For example in the driving task, while the best policy for driving the car would be to control the wheel using both arms, by exploiting the extra DOF in our body we can perform the same task sub-optimally by engaging one arm for turning the wheel and releasing the other arm for committing to the other subgoals of lower pri-

Appearing in *Proceedings of the 22^{nd} International Conference on Machine Learning*, Bonn, Germany, 2005.

ority such as drinking coffee, for example. However, the key advantage of coarticulation in concurrent decision making lies in its efficient search in the exponential space of concurrent actions. The action selection mechanism in this approach is restricted to those that ascend the value functions associated with each activity. This interactive search enables the agent to perform the search in a much smaller set of concurrent actions with a controllable cost in optimality.

In this work, we also present approximate algorithms for scaling the coarticulation framework to large domains, such as concurrent decision making in MDPs, where the concurrent action space is exponentially large. Unfortunately the algorithms presented in (Rohanimanesh et al., 2004) do not scale to such large domains. Thus efficient algorithms for computing the set of ascending policies for activities, and also scalable algorithms for the action selection problem is required.

Most of the related work in the context of Markov decision processes assume that the subprocesses modeling the activities are additive utility independent (Boutilier et al., 1997; Singh & Cohn, 1998; Guestrin & Gordon, 2002) and do not address concurrent planning with a set of learned activities modeled as temporally extended actions. In contrast we focus on problems where the overall utility function may be expressed as a non-linear function of sub-utility functions that have different priorities.

The rest of this paper is organized as follows: In section 2 we briefly overview the coarticulation framework. In section 3 we present approximate algorithms for scaling the coarticulation framework to large problems such as the concurrent decision making problem. We present our empirical results in a simulated domain in section 4. Finally, we summarize the paper in section 5 and describe some future directions.

2. Coarticulation in MDPs

Coarticulation in MDPs (Rohanimanesh et al., 2004) can be viewed as the problem of simultaneously committing to multiple subgoals, placing more weight on subgoals of higher priority. More formally, this approach assumes that the agent has access to a set of learned activities modeled by a set of minimum cost-to-goal ϵ-redundant controllers $\zeta = \{\mathcal{C}_i\}_{i=1}^n$. Each controller is designed to achieve a subgoal ω_i from a set of subgoals $\Omega = \{\omega_i\}_{i=1}^n$, and is modeled as a subgoal option (Precup, 2000) defined over an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$. A controller is ϵ-redundant if it admits multiple optimal or ϵ-ascending policies. A policy π is ϵ-ascending if it satisfies the following conditions:

1. **Ascendancy**:

$$\forall s \in \mathcal{S}, \quad E_{s' \sim \mathcal{P}_s^{\pi(s)}}\{\mathcal{V}^*(s')\} - \mathcal{V}^*(s) > 0 \qquad (1)$$

2. **ϵ-optimality**:

$$\forall s \in \mathcal{S}, \quad \mathcal{Q}^*(s, \pi(s)) \geq \frac{1}{\epsilon}\mathcal{V}^*(s) \qquad (2)$$

The first condition (*ascendancy*) guarantees that the policy arrives in a goal state in a bounded number of steps. The second condition (*ϵ-optimality*) assures the policy deviates from the optimal policy by a factor inversely proportional to ϵ (note that in minimum cost-to-goal problems all rewards are negative, except in a goal state). Given a minimum cost-to-goal controller $\mathcal{C}$, we can compute the set of policies that are ϵ-ascending on $\mathcal{V}_\mathcal{C}^*$ in every state $s \in \mathcal{S}$ by verifying the conditions in Equations 1 and 2 for every action $a \in A(s)$. We define the *ϵ-redundant set* for every state $s \in \mathcal{S}$ as:

$$\mathcal{A}_\mathcal{C}^\epsilon(s) = \{a \in \mathcal{A}(s) \,|\, a \text{ is either optimal, or } \epsilon\text{-ascending}\}$$

Now, given a subset of prioritized subgoals $\omega = \{\omega_i\}_{i=1}^k$, the agent has to devise a policy that simultaneously commits to the most possible number of subgoals according to their degree of significance. We can think of this problem as a multi-criterion reinforcement learning problem (Gabor et al., 1998), in which the reward signal is a vector whose elements are the rewards associated with the controllers, and a lexicographical ordering of such reward vectors are defined according to the priority order of the controllers. For specifying the order of priority relation among the controllers we use the expression $\mathcal{C}_j \triangleleft \mathcal{C}_i$, where the relation "$\triangleleft$" expresses the *subject-to* relation (following (Huber, 2000)). This equation should read: controller $\mathcal{C}_j$ subject-to subtask $\mathcal{C}_i$. A priority ranking system is then specified by a set of relations $\{\mathcal{C}_j \triangleleft \mathcal{C}_i\}$. Without loss of generality we assume that the controllers are prioritized based on the following ranking system: $\{\mathcal{C}_j \triangleleft \mathcal{C}_i | i < j\}$.

The action selection mechanism in the coarticulation framework takes the *ordered intersection* of the ϵ-redundant sets computed for every controller in progress. The ordered intersection operator works as follows: First, we set $U_1 = \mathcal{A}_{\mathcal{C}_1}^{\epsilon_1}$, and then for $i = 2, 3, \ldots, k$ we perform: $U_i = U_{i-1} \cap \mathcal{A}_{\mathcal{C}_i}^{\epsilon_i}$. If $U_i = \emptyset$, we chose $U_i = U_{i-1}$, and continue to the next iteration. After the algorithm terminates, U_k returns a set of actions that achieve the subgoals according to their order of priority. The computational cost of computing the redundant-sets in every state s $\mathcal{A}_\mathcal{C}^\epsilon(s)$ is linear in the number of states and actions:

$O(|S|\,|A|)$. Also the computational cost of performing the action selection mechanism in every state s is $O((k-1)\,(\max_{\mathcal{C}}|\mathcal{A}^{\epsilon}_{\mathcal{C}}(s)|)^2)$.

3. Coarticulation and Concurrent Decision Making

In Section 1, we described several reasons for why coarticulation could be one natural way for generating parallel execution plans in systems with multiple DOF. For example in the driving task, we can design three controllers associated with the subgoals: $\mathcal{C}_{navigate}$, $\mathcal{C}_{cell}$, and $\mathcal{C}_{coffee}$, where each controller is defined over an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}^{\mathcal{C}}, \mathcal{P}\rangle$. The overall objective can be approximated in terms of the concurrent optimization of the subgoals using the following priority ranking system: $\mathcal{C}_{navigate} \lhd \mathcal{C}_{cell} \lhd \mathcal{C}_{coffee}$. For each controller, we can compute the redundant sets, and then perform the action selection algorithm that we described in Section 2.

As stated in Section 1, one immediate problem with applying coarticulation to this problem is the combinatorial space of concurrent actions. In this example, the set of actions in the MDP $\mathcal{M}$ can be described via a set of action variables $\mathbf{a} = \{a_i\}_{i=1}^n$, where each a_i controls a DOF in the system. Each assignment $\bar{\mathbf{a}}$ to the set of action variables denotes a concurrent action $a \in \mathcal{A}$. In the driving task, for example, there are action variables associated with the arms, hands, eyes, head, and so on. The total number of concurrent actions is exponential in the number of action variables, and hence the algorithms for computing the redundant sets and the action selection mechanism that we described in Section 2 are computationally intractable for such problems. In the rest of this section, we present an approximate method for efficiently solving this problem.

The general idea in our approach is rather than verifying the conditions in Equations 1 and 2 for the exponential set of concurrent actions, which is intractable, we only verify them for the top h concurrent actions that have the top h state-action values in state s. We show that our approximate algorithm computes the top h concurrent actions with the computational complexity logarithmic in h. The parameter h is an input parameter which can be tuned to counterbalance the tradeoffs between the computational complexity and the flexibility of the controller.

Let $\mathcal{C}$ be a redundant controller defined over an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}\rangle$, where the set of concurrent actions $\mathcal{A}$ are described via a set of discrete action variables $\mathbf{a} = \{a_i\}_{i=1}^n$. Each action variable a_i takes on discrete values from some finite domain $Dom(a_i)$. Without loss of generality, we assume that $\forall a_i \in \mathbf{a}$, $|Dom(a_i)| = d$. We further assume that the optimal state-action value function $\mathcal{Q}^*$ associated with the controller $\mathcal{C}$ is approximated using linear function approximation techniques (Sutton & Barto, 1998) and admits the following linear additive form:

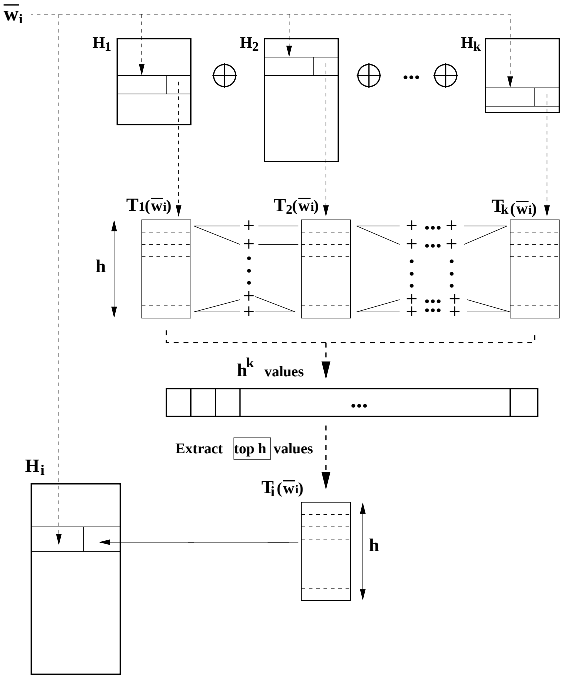

Figure 1. Visualization of step 5 of the Algorithm 1. Each function $\mathcal{H}_i$ returns a sorted table of size h. From cross summation of table values across $\mathcal{H}_i$ functions, a new table of the top h summations is produced.

$$\mathcal{Q}^*(s, \{a_i\}_{i=1}^n) \approx \bar{\mathcal{Q}}^*(s, \{a_i\}_{i=1}^n) = \sum_{i=1}^m \mathcal{Q}_i(s, \mathbf{u}_i) \tag{3}$$

where each $\mathcal{Q}_i(s, \mathbf{u}_i)$ is a local function defined over states and a subset of action variables $\mathbf{u}_i \subset \mathbf{a}$. We introduce an operator $\Gamma^h_{\mathbf{a}}$ which returns the top h values of a function $\mathcal{Q}^*(s, \mathbf{a})$. By exploiting the linearity of the approximate state-action value function (Equation 3) we can use an algorithm in spirit similar to the variable elimination algorithm in Bayesian networks and efficiently compute $\Gamma^h_{\mathbf{a}}.\bar{\mathcal{Q}}^*(s, \mathbf{a})$. Our approach is inspired by the action selection algorithm introduced in (Guestrin et al., 2002) that actually solves the special case for $h = 1$ (i.e., $\Gamma^1_{\mathbf{a}}$), which is the $\max_{\mathbf{a}}$ operator. It is also closely related to the problem of finding the h most probable configurations in probabilistic expert systems (Nilsson, 1998). The general idea is rather than summing all local functions and then performing the $\Gamma^h_{\mathbf{a}}$ operator, we perform it over variables one at a time, using only summands that involve the eliminated variable. For example, consider the following state-action value function defined over a set of action variables $\mathbf{a} = \{a_1, a_2, a_3\}$:

$$\bar{\mathcal{Q}}^*(s, a_1, a_2, a_3) = \mathcal{Q}_1(s, a_1) + \mathcal{Q}_2(s, a_1, a_2) + \mathcal{Q}_3(s, a_2, a_3)$$

by applying the $\Gamma^h_{\mathbf{a}}$ operator and the variable elimination algorithm, we obtain:

$$\begin{aligned}
&\Gamma^h{}_{\{a_1,a_2,a_3\}} . \bar{\mathcal{Q}}^*(s, a_1, a_2, a_3) = \\
&\Gamma^h{}_{\{a_1,a_2,a_3\}} . (\mathcal{Q}_1(s, a_1) + \mathcal{Q}_2(s, a_1, a_2) + \mathcal{Q}_3(s, a_2, a_3)) = \\
&\Gamma^h{}_{\{a_1\}} . \, (\mathcal{Q}_1(s, a_1) \oplus \\
&\quad \Gamma^h_{\{a_2\}} . \left(\mathcal{Q}_2(s, a_1, a_2) \oplus \Gamma^h_{\{a_3\}} . \mathcal{Q}_3(s, a_2, a_3)\right)\Big)
\end{aligned}$$

Note that in the above equation, we used the special sum operator $\oplus$, because at each elimination step the summation is performed over functions that return the top h maximum values of the past elimination steps, which need to be combined in order to obtain the updated top h maximum values as a result of the elimination of the next variable. Note that in the special case $h = 1$, this operator turns into the plus operator. The above procedure is summarized in Algorithm 1. The key computational steps are the steps 5 and 6 of this algorithm.

Before describing the details of these two steps, first we introduce some useful notation. Let $\mathcal{H}(\mathbf{w})$ denote a one-to-many function defined over a set of variables $\mathbf{w}$. Figure 1 shows a tabular view of this function, and demonstrates the details of the computations performed in the step 5 of the Algorithm 1. For every setting of variables $\bar{\mathbf{w}}$, it returns the sorted top h values and assignments to the subset of eliminated variables from the previous steps (tables $T(\bar{\mathbf{w}})$ in Figure 1). Before the elimination algorithm starts, we can represent each summand $\mathcal{Q}_i(s, \mathbf{u}_i)$ as some function $\mathcal{H}_i(\mathbf{u}_i)$ (for simplifying equations, we drop the state s from the notations), where every assignment of the variables $\bar{\mathbf{u}}_i$ is mapped to a single value $\mathcal{Q}_i(s, \bar{\mathbf{u}}_i)$. Assume that the algorithm is at iteration i, where the variable a_i is selected for elimination. Let $\{\mathcal{H}_j\}_{j=1}^k$ be the set of summands that involve the variable a_i. Also let $\mathbf{y}_i$ denote the rest of the variables involved in $\{\mathcal{H}_j\}_{j=1}^k$ that are connected to a_i, and let $\mathbf{w}_i = \mathbf{y}_i \cup \{a_i\}$. As shown in Figure 1 , for every setting of variables $\bar{\mathbf{w}}_i$ summand $\mathcal{H}_j$ returns a sorted top h values and also the assignments to a subset of past eliminated variables (represented as tables $T_j(\bar{\mathbf{w}}_i)$). There are k such tables and we need to compute the top h maximum values from the set of all cross summations of k elements, one from each table $T_j(\bar{\mathbf{w}}_i)$.

There are h^k such values and a naive approach would first compute the whole h^k summations, and then extract the top h maximum values, with the computational complexity of $O(h^k(k-1) + h\,log(h))$ (the first term is the complexity of computing the summations, and the second term denotes the complexity of sorting these values). However considering that each table is sorted, we can perform the above computation more efficiently. Rather than summing all the values across all tables, we perform the summation over two tables at a time, and extract a new table with the top h maximum values of the pairwise table summation. We then repeat it for the rest of the tables. When performing the pairwise cross summation over two tables, we only need to perform the summation only over the top $\sqrt{h}$ elements from each table, since the tables are sorted. The computational complexity of this method is $O((k-1)(h + h\,log(h)))$. The final top h elements are stored in a new function $\mathcal{H}_i(\mathbf{w}_i)$ for the setting $\bar{\mathbf{w}}_i$.

Algorithm 1 Function $\Gamma^h_{\mathbf{a}}$

```
Inputs:
    $\mathbf{s}$                                         ▷ Current state
    $\sum_{i=1}^m \mathcal{Q}_i(s, \mathbf{u}_i)$        ▷ $\bar{\mathcal{Q}}^*$ function
    $\{a_i\}$                                            ▷ Elimination order
    $h$                                      ▷ Number of top max elements
Outputs:
    $\{\bar{\mathbf{a}}_i, v_i\}_{i=1}^h$        ▷ Top $h$ assignments and values

1: Let $\mathcal{F} = \{\mathcal{Q}_i(s, \mathbf{u}_i)\}_{i=1}^m$     ▷ set of summands
2: while not all variables eliminated do
3:     Pick the next variable $a_i$
4:     Extract all summands $\{\mathcal{H}_j\}$ from $\mathcal{F}$
           that involve $a_i$
5:     Perform: $\mathcal{H}_i \leftarrow \oplus(\{\mathcal{H}_j\})$
6:     Eliminate $a_i$ from $\mathcal{H}_i$ to obtain $\mathcal{H}_i^-$
7:     Add $\mathcal{H}_i^-$ to $\mathcal{F}$
8: end while
```

The details of the computations of step 6 of the Algorithm 1 are demonstrated in Figure 2. Note that step 5 returns a newly introduced function $\mathcal{H}(\mathbf{w}_i)$ that involves the variable a_i. The elimination takes place in step 6. First, a new function $\mathcal{H}^-(\mathbf{y}_i)$ is introduced that involves only the variables connected to a_i (i.e., $\mathbf{y}_i$). Every setting $\bar{\mathbf{y}}_i$ is mapped to d tables (where $|Dom(a_i)| = d$), each for one assignment of the variable a_i. Each table contains the top h values and settings for a subset of eliminated variables in the previous steps. We need to extract the top h values across these tables. There are d sorted tables of size h, and we can extract the top h values across them with the computational complexity of $O(h.d)$. The new set of values are then stored in the function $\mathcal{H}_i^-(\mathbf{y}_i)$ for the setting $\bar{\mathbf{y}}_i$ (see Figure 2).

This yields the overall computational complexity of

$O(n\,d^{|\mathbf{w}|}\,(k\mathbf{h}log(\mathbf{h})+\mathbf{h}.d)) = O(n\,k\,d^{|\mathbf{w}|}\,\mathbf{h}log(\mathbf{h}))$ for the Algorithm 1. This complexity is logarithmic in $\mathbf{h}$, and exponential in the *network width* (Dechter, 1999) induced by the structure of the approximate state-action value function.

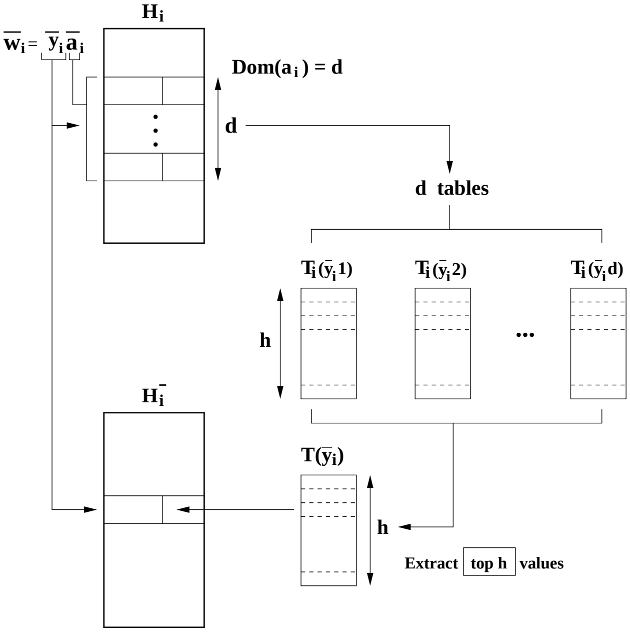
$\mathbf{H_i}$
$\overline{\mathbf{w}}_\mathbf{i} = \overline{\mathbf{y}}_\mathbf{i}\,\overline{\mathbf{a}}_\mathbf{i}$
$\mathbf{Dom(a_i) = d}$
d
d tables
$\mathbf{T_i(\bar{y}_i 1)}$
$\mathbf{T_i(\bar{y}_i 2)}$
...
$\mathbf{T_i(\bar{y}_i d)}$
h
$\mathbf{H_i^-}$
$\mathbf{T(\bar{y}_i)}$
h
Extract top h values

Figure 2. Visualization of step 6 of the Algorithm 1 where the variable a_i is eliminated.

By performing Algorithm 1 in state s, we obtain the top h concurrent actions and their values. We can then verify the ascendancy and ϵ-optimality conditions that we described in Section 2, for each action. In either case, we need to compute $\mathcal{V}^*(s)$. From the Bellman optimality equation (Sutton & Barto, 1998) we have $\mathcal{V}^*(s) = \max_{\mathbf{a}} \mathcal{Q}^*(s,\mathbf{a}) \approx \Gamma^1_{\mathbf{a}}\bar{\mathcal{Q}}^*(s,\mathbf{a})$ which can be computed using Algorithm 1. Thus verification of the ϵ-optimality condition can be efficiently done. For the ascendancy condition, we need to compute the expected optimal value of the next states given that the concurrent action $\bar{\mathbf{a}}$ is executed in state s. Expanding the optimal state-action value function for the action $\bar{\mathbf{a}}$ yields:

$$\mathcal{Q}^*(s,\bar{\mathbf{a}}) = \mathcal{R}(s,\bar{\mathbf{a}}) + \gamma E_{s'\sim\mathcal{P}^{\bar{\mathbf{a}}}_s}\{\mathcal{V}^*(s')\}$$

by subtracting $\gamma\mathcal{V}^*(s)$ from both sides and rearranging the terms, we obtain:

$$E_{s'\sim\mathcal{P}^{\bar{\mathbf{a}}}_s}\{\mathcal{V}^*(s')\} - \mathcal{V}^*(s) = \frac{1}{\gamma}(\mathcal{Q}^*(s,\bar{\mathbf{a}}) - \gamma\mathcal{R}(s,\bar{\mathbf{a}}) - \gamma\mathcal{V}^*(s))$$

Note that the right hand side of the Equation 4 can be efficiently computed for a concurrent action $\bar{\mathbf{a}}$ and can be used to verify the ascendancy condition.

Table 1. Action Variables

Left arm ($\mathbf{a}_l$)	Right arm ($\mathbf{a}_r$)	Eyes ($\mathbf{a}_e$)
pick	*pick*	*fixate-on-washer*
washer-to-front	*washer-to-front*	*fixate-on-front*
front-to-rack	*front-to-rack*	*fixate-on-rack*
rack-to-front	*rack-to-front*	*no-op*
front-to-washer	*front-to-washer*	
stack	*stack*	
no-op	*no-op*	

Given a set of redundant controllers $\{\mathcal{C}_i\}_{i=1}^k$, we can perform Algorithm 1 and compute the redundant sets for each controller. Note that the cardinality of the redundant set for the controller $\mathcal{C}_i$ is at most h_i. Thus the computational complexity of the action selection mechanism that we described in Section 2 in every state s becomes $O((k-1)\,(\max_i h_i)^2)$ which is polynomial in h.

4. Experiments

In this section we present a concurrent decision making task and apply the coarticulation approach using the approximate methods that we described in Section 3. Figure 3 shows a robot with three degrees of freedom,

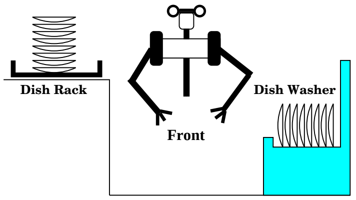

Figure 3. The robot's task is to empty the dish-washer and stack the plates in the dish-rack.

namely, the *eyes*, the *left arm*, and the *right arm*. The robot's task is to empty the dish-washer and stack the plates in the dish-rack. Each arm of the robot at any time can be in three predefined positions: *washer*, *rack*, and *front* as shown in Figure 3. In order to make a successful arm movement from a source position to a target position, the robot needs to first fixate at the target position. The eyes of the robot can also fixate on any of these positions. The set of actions that the robot can perform is described via a set of action variables $\mathbf{a} = \{a_l, a_r, a_e\}$, each controlling a DOF in the system. Action variable a_l controls the left arm, a_r controls the right arm, and a_l controls the eyes of the robot.

Table 1 shows the control actions for each action vari-

Table 2. State Variables

$\mathbf{s}_{washer}$	$\mathbf{l}_{pos}$, $\mathbf{r}_{pos}$, $\mathbf{e}_{pos}$	$\mathbf{l}_{stat}$, $\mathbf{r}_{stat}$	$\mathbf{s}_{rack}$
$0, 1 \ldots, n$	*washer*	*has-plate*	*stacked*
	front	*empty*	*not-stacked*
	rack		

able. There are control actions that move one arm from a source position to a destination position. However, the arms cannot move directly from washer to rack, and vice versa. In order to perform such movements, the robot needs to first move the arm from the source position to the front position, and then from that position to the target position in two primitive steps. The control action *pick* picks up a plate from the washer, if the arm is positioned at the dish-washer, and there is a plate to pick up. The control action *stack* stacks a plate into the dish-rack if the arm is holding a plate and is positioned at the dish-rack. The robot can also transfer a plate from one arm to the other, if both arms are positioned in front of the robot, and the empty arm executes the *pick* control action. The control actions for the eye movements cause the robot to fixate on the specified position. There is also a *no-op* action for each DOF, that does not influence that DOF.

The states of the robot are also described via a set of state variables summarized in Table 2. State variable $\mathbf{s}_{washer}$ keeps track of the number of plates in the dish-washer. State variable $\mathbf{l}_{pos}$ shows the current position of the left arm (i.e, *washer*, *front*, *rack*). State variable $\mathbf{l}_{stat}$ describes the current status of the left hand, i.e., whether it is holding a plate or it is empty. Similarly state variable $\mathbf{r}_{pos}$ and $\mathbf{r}_{stat}$ describe the position and status of the right arm. State variable $\mathbf{e}_{pos}$, describes the current gaze of the robot's eyes. Finally, state variable $\mathbf{s}_{rack}$ describes whether or not a plate has been stacked in the dish-rack.

Any assignment to the set of action variables forms a concurrent action. However, not all concurrent actions are allowed for execution in every state. Actions are pruned to simplify learning and enforce safety constraints (Huber, 2000). For example the left arm can execute the action *pick* only when it is located at the dish-washer and is empty, and there is also a plate to pickup. Any concurrent action that violates the safety constraints is referred to as an invalid action. If the robot executes an invalid action, it receives a large negative reward. The actions that control the gaze of the robot reflect the limitations of a real robot system. The robot is required to look at a target position before being able to move any of its arms to that position.

Recall that the overall objective is to empty the dish-washer and stack the plates in the dish-rack in the smallest number of steps. This objective can be approximated in terms of concurrent optimization of two competing subgoals: ω_{stack}, and ω_{pick}, with the priority ranking system: $\omega_{stack} \lhd \omega_{pick}$. ω_{stack} is the subgoal of stacking a plate in the dish-rack, and ω_{pick} is the subgoal of picking up a plate from the dish-washer. These subgoals compete for the DOF in the robot (i.e., eyes and arms). We design two controllers $\mathcal{C}_{pick}$, and $\mathcal{C}_{stack}$ that achieve each subgoal. Note that such controllers can be viewed as general purpose object manipulation controllers for picking up and stacking objects across different tasks. We can model the controller $\mathcal{C}_{pick}$ as an option $\langle \mathcal{I}_{pick}, \pi_{pick}, \beta_{pick} \rangle$, where $\mathcal{I}_{pick}$ is the set of states at which the robot can pick up a plate. This consists of the states where at least one of the robot's hands is empty and there is a plate in the dish-washer . The policy π_{pick}, specifies a closed loop policy for picking up a plate. β_{pick} defines the termination condition for this option and it occurs when the robot picks up a plate from the dish-washer.

Similarly, we can model the controller $\mathcal{C}_{stack}$ as an option $\langle \mathcal{I}_{stack}, \pi_{stack}, \beta_{stack} \rangle$, where $\mathcal{I}_{stack}$ is the set of states at which the robot can stack a plate . This consists of the states where the robot is holding at least one plate. The policy π_{stack} specifies a closed loop policy for stacking a plate. β_{stack} defines the termination condition for this option and it terminates when the robot stacks a plate in the dish-rack. Note that due to multiple DOF in the system, both controllers are ϵ-redundant for some ϵ. For example the robot can pick up a plate either by its left arm, or by its right arm. Or, it can stack a plate either by moving the arm that is currently holding the plate to the dish-rack and stack the plate, or it can hand it to the other hand and use the other hand to stack it.

It can be verified that the sequential solution (no coarticulation) that involves executing $\mathcal{C}_{pick}$ and then $\mathcal{C}_{stack}$ in sequence, does not provide the most efficient solution. For example while the robot is stacking a plate held by its right hand, it can concurrently pick up a new plate with its left hand. By coarticulating between these two controllers, the robot can perform an action that achieves the objective of the superior controller (i.e., $\mathcal{C}_{stack}$), while committing to the objective of the subordinate controller (i.e., $\mathcal{C}_{pick}$), if the intersection of the redundant sets of these two controllers is non-empty in the current state. To further

Table 3. CMACs tilings

Tiling	# of tiles
$\mathcal{Q}_1(s_{washer}, l_{pos}, l_{stat})$	12
$\mathcal{Q}_2(s_{washer}, r_{pos}, r_{stat})$	12
$\mathcal{Q}_3(s_{washer}, s_{rack}, r_{pos}, r_{stat})$	24
$\mathcal{Q}_4(s_{washer}, s_{rack}, \mathbf{a}_l, \mathbf{a}_r)$	196
$\mathcal{Q}_5(s_{washer}, l_{pos}, l_{stat}, e_{pos}, \mathbf{a}_l)$	252
$\mathcal{Q}_6(s_{washer}, r_{pos}, r_{stat}, e_{pos}, \mathbf{a}_r)$	252
$\mathcal{Q}_7(s_{washer}, s_{rack}, l_{pos}, l_{stat}, e_{pos}, \mathbf{a}_l)$	504
$\mathcal{Q}_8(s_{washer}, l_{pos}, l_{stat}, e_{pos}, \mathbf{a}_l, \mathbf{a}_e)$	1008
$\mathcal{Q}_9(s_{washer}, r_{pos}, r_{stat}, e_{pos}, \mathbf{a}_r, \mathbf{a}_e)$	1008
$\mathcal{Q}_{10}(s_{washer}, l_{pos}, r_{pos}, e_{pos}, \mathbf{a}_l, \mathbf{a}_r)$	2646

illustrate this, consider the following scenario: assume that the robot is currently holding a plate with its right arm positioned at the front and the controller $\mathcal{C}_{stack}$ is in progress. Also, assume that its left arm is positioned at front and is empty. In this state, the robot can execute at least two ϵ-ascending actions with respect to the $\mathcal{C}_{stack}$ controller: (1) move the right arm to the dish-rack and concurrently look at the front position; (2) move the right arm to the dish-rack and concurrently look at the dish-washer position. Note that the second action is also ϵ-ascending with respect to the $\mathcal{C}_{pick}$ controller, since by looking at the dish-washer position, the robot can then move its empty left arm to the dish-washer in order to pick a plate. By coarticulating between these two controllers, the robot executes the second action that is ϵ-ascending with respect to both controllers. Note that while this action moves the robot's right arm to the dish-rack to stack the plate, concurrently it moves the empty left arm to the dish-washer in order to pick a new plate.

In our experiments, both controllers are defined over an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$, where the states and actions are described via the set of variables given in Tables 2, and 1. All actions are stochastic; they succeed with probability p and fail with probability $(1-p)$. When actions succeed, they change the state of the robot to the next state as described above. Upon failure, or executing an invalid action, the robot does not change its state. All actions are also rewarded -1 upon termination.

In order to learn the optimal state-action value function associated with each controller, we used sparse-coarse-coded function approximator (CMACs) (Albus, 1981) combined with Sarsa(λ) algorithm (Sutton, 1996). We used CMACs consisting of 10 tilings (for the total of 5914 tiles) as listed in Table 3. For each controller, we learned the approximate value function which can be expressed as $\bar{\mathcal{Q}}^*(\mathbf{s}, \{\mathbf{a}_l, \mathbf{a}_r, \mathbf{a}_e\}) = \sum_{i=1}^{10} \mathcal{Q}_i(\mathbf{s}_i, \mathbf{a}_i)$, where $\mathbf{s}_i$ and $\mathbf{a}_i$ are the subset of state and action variables that are involved in the tiling $\mathcal{Q}_i$. Next, we applied the approximation algorithm that we described in Section 3 for computing the ϵ-redundant sets for each controller, using the elimination order $\{\mathbf{a}_l, \mathbf{a}_r, \mathbf{a}_e\}$. Figure 4 shows the performance of the coarticulation method, and also the sequential approach in which the $\mathcal{C}_{stack}$ and $\mathcal{C}_{pick}$ are executed in sequence (using their optimal policy) with no coarticulation. The performance is measured in terms of the total number of steps for completion of the task. These results are averaged over 20 tasks, each consisting of 27 episodes. Each episode is associated with a starting state, with 20 plates in the dish-washer, and the robots arms are set to empty. The horizontal axis depicts the starting states. An starting state is defined in terms of the various configurations of the state variables (e.g., initial positions of the arms and the eyes, and their status, etc) that are relevant to the task (i.e., at least one controller can be initiated). The bottom two plots are the performance of the coarticulation framework using $\epsilon = 0.70, 0.85$, and $h = 10$ when computing the redundant sets for each controller. From these results, we can observe that the coarticulation approach outperforms the sequential case in every starting state. Figure 5 shows the total number of coarticulation in the above task when the system is initialized in different starting states, for different values of ϵ. Note that by choosing $\epsilon = 0.99$, the controllers offer less flexibility and hence the total amount of coarticulation decreases. Next, in order to measure the accuracy of our approximate method when computing the redundant sets, we computed the exact value function without the function approximation, and then computed the redundant sets. Figures 6 and 7 shows the total number of hits and misses of the ϵ-ascending actions in ev-

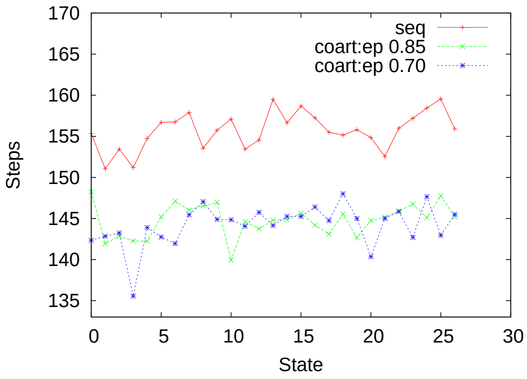

Figure 4. Performance of the coarticulation approach using different values of ϵ and the sequential approach.

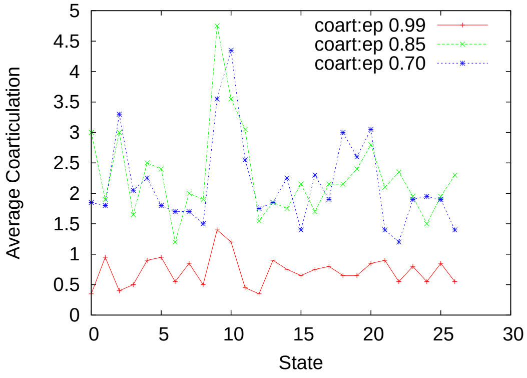

Figure 5. Total number of coarticulation for different starting states, and different values of ϵ.

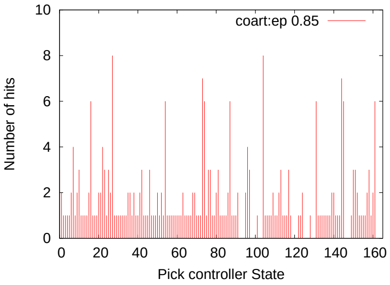

Figure 6. Number of correct ϵ-ascending actions computed by the approximate method for controller $\mathcal{C}_{pick}$ in every state that the controller can be executed.

ery state in which the controller $\mathcal{C}_{pick}$ can be executed. As it can be seen from these figures, our approximation technique has missed only a few actions in a small number of states. We also measured the false positive rate in every state and interestingly there was no false alarm in neither of states. This suggests that our approximation method can compute the redundant sets with a high precision.

5. Concluding Remarks

In this paper we studied an approach for scaling the coarticulation framework to large domains, such as concurrent decision making in systems with excess DOF. We presented an efficient approximate algorithm for computing the set of ϵ-redundant policies for redundant controllers. Although we considered structured actions, we did not fully exploit the underlying structure that many MDPs offer. We are currently investigating how such structure could be exploited for more efficient action selection mechanism in the coarticulation framework.

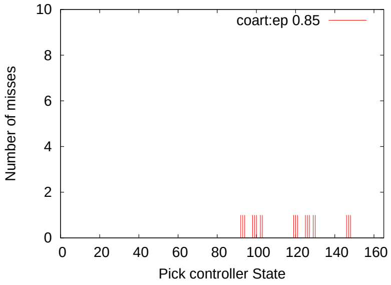

Figure 7. Number of missed ϵ-ascending actions computed by the approximate method for controller $\mathcal{C}_{pick}$ in every state that the controller can be executed.

Acknowledgments

We would like to thank Nathan Srebro for his useful comments and discussion. This research was supported in part by the National Science Foundation under grant ECS-0218125.

References

Albus, J. (1981). *Brain, behavior, and robotics*. ByteBooks.

Boutilier, C., Brafman, R., & Geib, C. (1997). Prioritized goal decomposition of Markov decision processes: Towards a synthesis of classical and decision theoretic planning. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence* (pp. 1156–1163). San Francisco: Morgan Kaufmann.

Dechter, R. (1999). *Bucket elimination: A unifying framework for probabilistic inference*. Artificial Intelligence.

Gabor, Z., Kalmar, Z., & Szepesvari, C. (1998). Multi-criteria reinforcement learning.

Guestrin, C., & Gordon, G. (2002). Distributed planning in hierarchical factored mdps. *In the Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence* (pp. 197 – 206). Edmonton, Canada.

Guestrin, C., Lagoudakis, M., & Parr, R. (2002). Coordinated reinforcement learning. *In Proceedings of the ICML-02*. Sydney Australia.

Huber, M. (2000). *A hybrid architecture for adaptive robot control*. Doctoral dissertation, University of Massachusetts, Amherst.

Nilsson, D. (1998). An efficient algorithm for finding the m most probable configurations in bayesian networks. *Statistics and Computing*, *8*, 159–173.

Precup, D. (2000). *Temporal abstraction in reinforcement learning*. Doctoral dissertation, Department of Computer Science, University of Massachusetts, Amherst.

Rohanimanesh, K., Platt, R., Mahadevan, S., & Grupen, R. (2004). Coarticulation in markov decision processes. *Proceedings of the Eighteenth Annual Conference on Neural Information Processing Systems: Natural and Synthetic*. Vancouver, Canada.

Singh, S., & Cohn, D. (1998). How to dynamically merge markov decision processes. *Proceedings of NIPS 11*.

Sutton, R., & Barto, A. (1998). *An introduction to reinforcement learning*. Cambridge, MA.: MIT Press.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems* (pp. 1038–1044). The MIT Press.