Self-Segmentation of Sequences: Automatic Formation of Hierarchies of Sequential Behaviors

Ron Sun[1], Chad Sessions[2]
[1] CECS Dept, University of Missouri, Columbia, MO 65211
[2] University of Alabama, Tuscaloosa, AL 35487

rsun@cecs.missouri.edu
573-884-7662

Appeared in: IEEE Transactions on Systems, Man, and Cybernetics: Part B, Cybernetics, Vol.30, No.3, pp.403-418. 2000.

Abstract

The paper presents an approach for hierarchical reinforcement learning that does not rely on a priori domain-specific knowledge regarding hierarchical structures. Thus this work deals with a more difficult problem compared with existing work. It involves learning to segment action sequences to create hierarchical structures (for example, for the purpose of dealing with partially observable Markov decision processes, with multiple limited-memory or memoryless modules). Segmentation is based on reinforcement received during task execution, with different levels of control communicating with each other through sharing reinforcement estimates obtained by each other. The algorithm segments action sequences to reduce non-Markovian temporal dependencies, and seeks out proper configurations of long- and short-range dependencies, to facilitate the learning of the overall task. Developing hierarchies also facilitates the extraction of explicit hierarchical plans. The initial experiments demonstrate the promise of the approach.

1 Introduction

Sequential behaviors (sequential decision processes) are fundamental to cognitive agents. The use of reinforcement learning (RL) for acquiring sequential behaviors is appropriate, and even necessary, when there is no domain-specific a priori knowledge available to agents (Sutton and Barto 1983, Barto et al 1995, Kaelbling et al 1996, Bertsekas and Tsitsiklis 1996, Watkins 1989). Given the complexity and differing scales of events in the world, there is a need for hierarchical RL that can produce action sequences and subsequences that correspond with domain structures. This has been demonstrated time and again, in terms of facilitating learning and/or dealing with non-Markovian dependencies, e.g., by Dayan and Hinton (1993), Kaelbling (1993), Lin (1993), Wiering and Schmidhuber (1998), Tadepalli and Dietterich (1997), Parr and Russell (1997), Dietterich (1997), and many others. Different levels of action subsequencing correspond to different levels of abstraction (Dayan and Hinton 1993, Knoblock, Tenenberg, and Yang 1994). Thus, subsequencing facilitates hierarchical planning as studied in traditional AI as well (Sacerdoti 1974, Knoblock, Tenenberg, and Yang 1994, Sun and Sessions 1998).

However, we noticed the shortcomings of structurally pre-determined hierarchies in RL (whereby structures are derived from a priori domain knowledge and thus fixed). The problems of such hierarchies include cost (because it is costly to obtain a priori domain knowledge to form hierarchies), inflexibility (because the characteristics of the domain for which fixed hierarchies are built can change over time), and lack of generality (because domain-specific hierarchies most likely vary from domain to domain). Even when limited learning is used to fine tune structurally pre-determined hierarchies (Parr and Russell 1997, Dietterich 1997), some of these problems persist. [1]

A more general approach would be to automatically develop hierarchies (of actions), from scratch, based only on some generic structures (such as a fixed number of levels) and to automatically tailor details of structures and their parameters with reinforcement learning. What this process amounts to is automatically *segmenting* action sequences (self-segmentation) and creating a hierarchical organization of action subsequences (which can be viewed as "macro-actions" or "subroutines"). Subsequences resulting from self-segmentation may be reused throughout a sequence or by different sequences (i.e, shared among different tasks). [2] In the end we have a hierarchical organization of sequential behaviors.

Although segmentation is evidently important, some existing hierarchical RL models either do not involve segmentation (i.e., using pre-determined hierarchies; e.g., Sutton 1995, Precup et al 1998, Parr and Russell 1997, Dietterich 1997), or involve only domain-specific segmentation processes. [3] In contrast, we will address the issue of learning segmentation without (or with little) domain-specific knowledge or procedures, which is important when domain-specific knowledge is not available, or costly to obtain. (Ring 1991, Schmidhuber 1992, Weiss 1995, Thrun and Schwartz 1995, Nevill-Manning and Witten 1997). Even when

[1] The shortcomings of limited learning hierarchies include, as with fixed hierarchies, inflexibility to some extent and lack of generality (because the structures are mostly domain-specific).

[2] When adaptive formation of hierarchies is carried out in a multiple-task context, the learning of different sequences can influence each other, and common subsequences may emerge. As a result, different tasks may share some subsequences (subroutines; Thrun and Schwartz 1995).

[3] Models in the latter category rely on domain-specific knowledge or procedures for segmentation and are thus not generic or autonomous; for example, Lin (1993) and Singh (1994) in reinforcement learning, Knoblock et al (1994) in hierarchical planning, Kuniyoshi et al (1991) in robotic learning, and so on.

domain-specific knowledge is available, sometimes it may be advantageous to avoid using it for the sake of generality of models that can be widely applied.

A difficult issue that we face in RL is how to deal with (non-Markovian) temporal dependencies (that is, situations in which state transition probabilities are determined not only by the current state and action but also by previous ones, due to partial observability; Lin 1993, McCallum 1996, Kaelbling et al 1996). [4] Such dependencies can create problems in reinforcement learning. Through the use of reinforcement learning at different levels, we may seek out proper configurations of non-Markovian temporal dependencies, with the goal of reducing dependencies through segmenting at proper places at different levels so as to facilitate the learning of the overall task (see details later).

Therefore, we need to deal with the following issues: (1) automatic (self) segmentation of sequences, with each segment to be handled differently by a different module, to reduce or remove non-Markovian temporal dependencies; (2) automatic development of common subsequences (that is, subroutines or subtasks) that can be used in various places of a sequence and/or in different sequences, to simplify non-Markovian dependencies and to form compact representations; (3) moreover, segmentation without (or with little) a priori domain-specific knowledge as embodied in domain-specific structures, domain-specific ways of creating such structures, and so on (because relying on a priori knowledge leads to a lack of generality). In the remainder of this paper, we will develop an approach for self-segmentation of sequences in a hierarchical fashion (SSS). In section 2, we review the basics of RL. In section 3, we present the SSS algorithm. In section 4, we analyze the algorithm in various ways. In section 5, we present experiments with SSS. In section 5, we discuss a number of issues, including comparing SSS with existing work. Note that, in this paper, we can only focus on demonstrating the algorithm through examples. We leave the in-depth theoretical analysis to future work, since it by itself is a major undertaking.

2 Review of RL

Reinforcement learning (RL) (Sutton and Barto 1983, Watkins 1989, Barto et al 1995, Bertsekas and Tsitsiklis 1996, Kaelbling et al 1996) has received a great deal of attention recently. RL can be viewed as an on-line variation of value-iteration dynamic programming (DP). DP in general can be defined as follows: there is a 5-tuple: (S, U, T, P, g), in which S is the set of state, U is the set of actions, and T is the probabilistic state transition function that maps the current state and the current action to a new state in the next time step. P is the (reactive) stationary *policy* that determines the action at the current time step given the current state: $P(s_t) = u_t$. g is the cost (or reward) function that maps certain states and/or actions to real values. That is, in DP, there is a discrete-time system, the state transitions of which depend on actions performed by an agent. In probabilistic terms, a Markovian process is in the working in determining a new state (from state transition) after an action is performed: $prob(s_{t+1}|s_t, u_t, s_{t-1}, u_{t-1}, ...) = prob(s_{t+1}|s_t, u_t) = p_{s_t, s_{t+1}}(u_t)$, where u_t is determined based on the policy P (assuming $u_t = P(s_t)$). In this process, costs (or rewards) can occur for certain states and/or actions. Normally the costs/rewards accumulate additively, with or without a discount factor.

[4] In such situations, optimal actions at one point may be dependent on not only the current state but also states and actions occurring some time ago. See Sondik (1978), Monahan (1982), and Puterman (1994) for treatments of POMDPs (partially observable Markovian decision processes).

The cumulative cost/reward estimate (which we will denote as J) that results from following an optimal policy of actions satisfies the Bellman optimality equation:

$$J(s_t) = \max_u \sum_{s_{t+1} \in S} p_{s_t, s_{t+1}}(u)(g(s_t) + \gamma J(s_{t+1}))$$

where g denotes cost/reward (which is assumed to be a function of states), s_t is any state and s_{t+1} is the new state resulting from action u. Or, using the notation $Q(s_t, u_t)$ (where $\max_u Q(s, u) = J(s)$), we have

$$Q(s_t, u_t) = \sum_{s_{t+1} \in S} p_{s_t, s_{t+1}}(g(s_t) + \gamma \max_u Q(s_{t+1}, u))$$

Based on the Bellman optimality equation, there are a number of on-line (RL) algorithms for learning Q or J functions. One is the Q-learning algorithm of Watkins (1989). The updating is done completely on-line, without explicitly using probability estimates:

$$Q(s_t, u_t) := (1 - \alpha) Q(s_t, u_t) + \alpha(g(s_t) + \gamma \max_{u_{t+1}}(Q(s_{t+1}, u_{t+1})))$$

$$= Q(s_t, u_t) + \alpha(g(s_t) + \gamma \max_{u_{t+1}}(Q(s_{t+1}, u_{t+1})) - Q(s_t, u_t))$$

where u_t is determined by an action policy that allows sufficient exploration (Bertsekas and Tsitsiklis 1996). For example, it can be based on the Boltzmann distribution:

$$prob(s_t, u_t) = \frac{e^{Q(s_t, u_t)/\tau}}{\sum_u e^{Q(s_t, u)/\tau}}$$

where τ is the temperature that determines the degree of randomness in action selection (other policies are also possible; Bertsekas and Tsitsiklis 1996). The updating is done based on actual state transition instead of transition probabilities; that is, on-line "simulation" is performed. With enough sampling, the transition frequency from s_t and u_t to s_{t+1} should approach $p_{s_t, s_{t+1}}(u_t)$, and thus provides an estimation. Therefore, the result will be the same values as in the earlier specification of Q values. Such learning allows completely autonomous learning from scratch, without a priori domain knowledge.

3 The SSS Algorithm

Below we will present an approach for automatically developing hierarchical reinforcement learning systems, the SSS algorithm, which stands for *Self Segmentation of Sequences*. Different from usually Markovian domains for Q-learning, we will deal with non-Markovian domains (partially observable Markovian decision processes), where an agent observes local information that constitutes an observational state (or an observation; Bertsekas and Tsitsiklis 1996) but not a true state. The true state may be determined from the history of observational states. In such domains, a single Q learner may not work, because there may not be a consistent Markovian policy available. We create a hierachical structure that involves multiple modules (each with either little or no memory) for dealing with such domains, without a priori domain knowledge. (In the following discussion, the term "state" in general refers to observational states.)

3.1 The General Idea

The general idea is as follows: There are a number of individual action modules (referred to as Q modules below). Each of them selects actions to be performed at each step. However, for each Q module, there is also a corresponding controller CQ, which determines at each step whether the Q module should continue or relinquish control. When a Q module currently in control relinquishes its control, a higher-level controller, AQ, will decide which Q module should take over next from the current point on. So, in any given state, a pair of Q and CQ should be in control. When the CQ selects *continue*, the Q will select an action with regard to the current state that will affect the environment and thus generate reinforcement from the environment. When the CQ selects *end*, the control is returned to AQ, which will then proceed immediately to select (with regard to the current state) another CQ (and its corresponding Q) to take over. This cycle then repeats itself (cf. the idea of "options"; Precup et al 1998).

Each component in this system is learned from scratch (in contrast to e.g. Precup et al 1998, Dietterich 1997). Let us look into how each component learns:

- Each Q module tries to receive as much reinforcement as possible before it is forced to give up. i.e., performs optimization within a local segment (a set of adjacent states). However, local segments change over time and thus the world is not static for a Q module. That is, the learning of self-segmentation (determined jointly by CQ and AQ) is concurrent with the local learning of Q modules. The non-static situation adds to the complexity of learning.

- Each CQ tries to determine at the current point whether it is more advantageous to terminate the corresponding Q module or to let it continue, in terms of maximizing the overall reinforcement. [5] A CQ considers whether or not to give up based on which way will lead to more overall reinforcement. This is partially determined by the performance of all Q's executed subsequently and by the selection made by AQ. So, the CQ has to deal with the problem of dynamic interaction.

- AQ tries to learn the selection of Q modules at various points (by an abstract control action), by evaluating which selection will lead to more overall reinforcement. Again this is not a static situation: which states becomes decision points for AQ is partially determined by CQ's (which determines when to give control back to AQ, which is in turn determined by Q's, which select actions that generate reinforcement). As each Q and CQ pair learns, the selection by AQ may have to change also.

This two-level structure can be extended to more than two levels.

3.2 The Algorithm

3.2.1 The Overall Specification

Let s denote the observational state of an agent at a particular moment (not necessarily a complete description). Assume reinforcements are associated with the current observational

[5] If the Q module is continued, the overall reinforcement is the (discounted) sum of all the reinforcements that will be received by the Q module plus the (discounted) sum of reinforcement that will be received by subsequent Q's (since the active Q module will be terminated eventually). If the Q module is terminated, the overall reinforcement is the (discounted) sum of all the reinforcement that will be received by subsequent Q's.

state: $g(s)$. We may deal with either discounted cumulative reinforcement, undiscounted cumulative reinforcement (in finite-horizon problems), or even average reinforcement (a relatively straightforward extension). We will focus on Q-learning to illustrate our idea (although other RL algorithms are also applicable). Let us look into a minimal structure: a two level hierarchy, in which there are three types of learning modules:

- Individual action module Q: Each performs actions and learns through Q-learning.
- Individual controller CQ: Each CQ learns when a Q module (corresponding to the CQ) should continue its control and when it should give up the control. The learning is accomplished through (separate) Q-learning.
- Abstract controller AQ: It performs and learns abstract control actions, that is, which Q module to select under what circumstances. The learning is accomplished through (separate) Q-learning.

The overall algorithm is as follows:

- 1. Observe the current state s.
- 2. The currently active Q/CQ pair takes control. If there is no active one (when the system first starts), go to step 5.
- 3. The active CQ selects and performs a control action based on $CQ(s, ca)$ for different ca. If the action chosen by CQ is *end*, go to step 5. Otherwise, the active Q selects and performs an action based on $Q(s, a)$ for different a.
- 4. The active Q and CQ performs learning. Go to step 1.
- 5. AQ selects and performs an abstract control action based on $AQ(s, aa)$ for different aa, to select a Q/CQ pair to become active.
- 6. AQ performs learning. Go to step 3.

3.2.2 Learning

The learning rules, along with explanations of their purposes, are detailed as follows.

- Individual action modules. For the active Q_k, when neither the current action nor the next action by the corresponding CQ_k is *end*, [6] we use the usual Q-learning rule:

$$\Delta Q_k(s, a) = \alpha(g(s) + \gamma \max_{a'} Q_k(s', a') - Q_k(s, a))$$

where s' is the new state resulting from action a in state s.

Explanation: When CQ_k decides to continue, Q_k learns to estimate reinforcement generated (1) by its current action and (2) by its subsequent actions (assuming a greedy policy: taking the action that has the highest value in a state), (3) plus the reinforcement generated later by other modules after Q_k/CQ_k relinquishes control (see the next learning rule). The value of Q_k is the sum of the estimates of these three parts.

[6] Note that there is a one-step delay in the application of the learning rule.

When the current action of CQ_k (in s) is *continue* and the next action of CQ_k (in s') is *end*, the Q_k module receives as reward the maximum value of AQ:

$$\Delta Q_k(s,a) = \alpha(g(s) + \gamma \max_{aa'} AQ(s', aa') - Q_k(s,a))$$

where s' is the new state (resulting from action a in state s) in which control is returned to AQ by CQ_k, and aa is any abstract action of AQ.

Explanation: As the corresponding CQ terminates the control of the current Q module, AQ has to take an abstract action in state s', the value of which is the (discounted) total reinforcement that will be received from that point on (by the whole system), if that action is taken and if thereafter the greedy policy (taking the action that has the highest value in the current state) is followed by AQ and the current (stochastic) policies are followed in all the other modules. So, at termination, the Q module learns the total (discounted) reinforcement that will be received by the whole system thereon (if the greedy policy is followed by AQ and the current stochastic policies are followed by the other modules). Combining the explanations of the above two rules, we see that a Q module decides its action based on which action will lead to higher overall reinforcement from the current state on (both by itself and by subsequent modules). [7]

Remarks: If reinforcement is given only at the end when the goal is reached, the above learning rules amount to assigning a reward to a module based on the number of steps to the goal from the state in which it was terminated. Instead of counting the number of steps to the goal, we simply use discounting of reinforcement to achieve the same effect (because the total amount of discounting is determined by the number of steps to the goal). Thus, when Q is terminated, it estimates in this way how close it has gotten to the goal at that point. In this way, we do not need to use separate rewards for individual modules (which require a priori domain knowledge to set up). The same can be said about CQ below.

- Individual controllers. For the corresponding CQ_k, there are also two separate learning rules, for the two different actions. When the current action by CQ_k is *continue*, the learning rule is the usual Q-learning:

$$\Delta CQ_k(s, continue) = \alpha(g(s) + \gamma \max_{ca'} CQ_k(s', ca') - CQ_k(s, continue))$$

where s' is the new state resulting from action *continue* in state s and ca' is any control action by CQ_k.

Explanation: When CQ_k decides to continue, it learns to estimate reinforcement generated by the actions of the corresponding Q_k, assuming that CQ_k gives up control whenever *end* has a higher value, plus the reinforcement generated after Q_k/CQ_k relinquishes control (see the next learning rule). That is , CQ_k learns the value of *continue* as the sum of the expected reinforcement generated by Q_k and the expected reinforcement generated by subsequent modules thereafter.

When the current action by CQ_k is *end*, the learning rule is

$$\Delta CQ_k(s, end) = \alpha(\max_{aa} AQ(s, aa) - CQ_k(s, end))$$

[7] Note that we are being optimistic about the selections to be made by AQ. An alternative (the SARSA learning rule) is to use $AQ(aa, s)$ in the above formula, where aa is the actual abstract action taken by AQ. But that change will not produce an estimated average of reinforcement unless the learning rules for AQ are changed accordingly.

where aa is any abstract control action by AQ.

Explanation: That is, when a CQ ends, it learns the value of the best abstract action to be performed by AQ, which is equal to the (discounted) total reinforcement that will be accumulated by the whole system from the current state on (if the greedy policy is followed by AQ and the current stochastic policies are followed by the other modules). Combining the above two learning rules, in effect, the CQ module learns to make its decisions by comparing whether giving up or continuing control will lead to more overall reinforcement from the current state on.

- Abstract controllers. For AQ, we have the following learning rule:

$$\Delta AQ(s, aa) = \alpha(ag(s) + \gamma^m \max_{aa'} AQ(s', aa') - AQ(s, aa))$$

where aa is the abstract control action that selects a Q module to take control, s' is the new state (after s) in which AQ is required to make a decision, m is the number of time steps taken to go from s to s' (used in determining the amount of discounting γ^m), and ag is the (discounted) cumulative reinforcement received after the abstract control action for state s and before the next abstract control action for state s' (that is, $ag(s) = \sum_{k=0,1,....,m-1} g(s_k)\gamma^k$, where $s_0 = s$ and $s_m = s'$).

Explanation: AQ learns the value of an abstract control action that it selects, which is the (discounted) cumulative reinforcement that the chosen Q module will accrue before AQ has to make another abstract action decision, plus the accumulation of reinforcement from the next abstract action on, assuming the greedy policy will be followed by AQ and the current (stochastic) policies will be followed by the other modules. In other words, AQ estimates the total reinforcement to be accrued (if it is greedy and all the other modules follow their current policies). So in effect AQ learns to select lower-level modules by comparing what selections lead to more overall reinforcement from the current point on.

Technically, we only need either CQ's or AQ, but not both, since their values are closely related (if an identical input representation is given to them). However, as we will discuss later, we have to keep them separate when we have more than two levels. Furthermore, if we introduce different input representations for different levels (see section 4.1), we must keep them separate (even in the case of two levels). Finally, if we want these modules to be independent, autonomous entities (agents), then the separation of CQs and AQ is justified on the basis of separate learning by separate entities.

3.2.3 Parameters

In SSS, there are the following sets of parameters. We have learning rates for each of the three types of modules: α_Q, α_{AQ}, and α_{CQ}. We also have the parameters concerned with randomness (or temperature) for stochastic decision making for each of the three types of modules: τ_Q, τ_{AQ}, and τ_{CQ}. We may gradually lower the learning rates and the temperatures according to their respective schedules (see section 4 for details).

In this algorithm, due to complex interactions, it is sometimes necessary to tune the parameters to generate proper performance. Otherwise, some combinations of parameters and initial conditions might lead to very bad performance. Although this is a shortcoming, it is no more severe than many popular numerical methods such as backpropagation algorithms.

Our experiments (described later) showed that there were wide ranges of parameter values that could generate good performance and thus the dependency on parameter tuning was limited.

3.2.4 Temporal Representation in Modules

In SSS, we may need to construct temporal representations, in order to avoid excessive segmentation when there is too much temporal dependency. One possibility is recurrent neural networks (e.g., Elman 1990). RNNs have been used to represent temporal dependencies of various forms in much previous work, such as Elman (1990), Lin (1993), Whitehead and Lin (1995), and Giles et al (1995). RNNs have shortcomings, such as long learning time, possible inaccuracy, possibly improper generalizations, and so on. Theoretically, a RNN can memorize arbitrarily long sequences, but practically, due to limitations on precision, the length of a sequence that can be memorized is quite limited (Lin 1993).

As a more viable alternative to RNNs, we may use decision trees in RL, along the line of McCallum (1996), which splits a state if a statistical test shows that there are significant differences among the cases covered by that state (that is, if the state can be divided into two states by adding features from preceding steps and provide more accurate assessment of the (discounted) cumulative reinforcement that will be received). Compared with a priori specifications of non-Markovian temporal dependencies (i.e., specifying exactly which preceding step is relevant ahead of the time), this method is more widely applicable, because a priori knowledge may not be available. Compared with RNNs, this method can be more robust, because, due to limited precision, temporal representation in RNNs can fade away quickly over a few steps and thus RNNs may have trouble dealing with long-range dependencies. Compared with using a full-scale temporal representation such as the n-fold state space method (in which an n-fold state space is made up of states and their corresponding $n-1$ preceding states), the advantage of this method is that it may (ideally) seek out those and only those relevant preceding steps that are involved in temporal dependencies and thus reduces the complexity of representation. The shortcoming of the method is that, if unconstrained, the search for relevant historic features can be exhaustive and thus costly.

3.3 Extensions to More Levels

The above specification is with respect to a two-level hierarchy, the component of which can be denoted as $AQ_0(=Q)$, $CQ_0(=CQ)$, and $AQ_1(=AQ)$. We call it the zeroth-order system. We can easily extend it to more levels. That is, on top of AQ_0's, CQ_0's, and AQ_1's, we have AQ_2 for selecting among AQ_1's as well as CQ_1 that determines when the corresponding AQ_1 should give up. We obtain a first-order system this way. Then, on top of AQ_2's, we can have AQ_3 for selecting among AQ_2's and CQ_2's for determining termination. That is, we use a number of first-order systems to develop a second-order system. This process can be repeated for any number of times.

At any step, there should always be an active AQ_i/CQ_i pair at each level i (where $i = 0, 1, ..., M$). We use k (or j) to denote the pair of controllers that is active. Here is the algorithm:

/* using a bottom-level module */

– 1. Observe the current state s.

– 2. The currently active bottom-level pair, AQ_0^k/CQ_0^k, takes control. If there is no active one, set i to M, activate the controller AQ_M/CQ_M, and go to step 5.

– 3. The active CQ_0^k selects an action based on $CQ_0^k(s, ca)$ for different ca. If the action chosen by CQ_0^k is *end*, set i to 1 and go to step 5. Otherwise, the corresponding AQ_0^k selects and performs an action based on $AQ_0^k(s, a)$ for different a.

– 4. The active AQ_0^k and CQ_0^k perform learning in their respective ways. Go to step 1.

/* moving up or down the hierarchy */

– 5. The currently active controller at level i, CQ_i^k/AQ_i^k, takes control. CQ_i^k selects an action based on $CQ_i^k(s, ca)$ for different ca. (CQ_M always selects *continue*.)

– 6. If the CQ_i^k decision is to continue, an CQ_{i-1}^j/AQ_{i-1}^j pair is selected by AQ_i^k and activated ($i > 0$).

– 6.1. CQ_i^k and AQ_i^k perform learning respectively.

– 6.2. If $i - 1 > 0$, set i to $i - 1$ and go to step 5 (moving down the hierarchy). Otherwise, go to step 3 (reaching the bottom of the hierarchy).

– 7. If the CQ_i^k decision is to end, find the currently active controller at the higher level $i + 1$, CQ_{i+1}^j/AQ_{i+1}^j ($i < M$).

– 7.1. CQ_i^k and AQ_i^k perform learning respectively.

– 7.2. Set i to $i + 1$ and go to step 5 (moving up the hierarchy) ($i < M$).

The new learning rules are as follows (where p is a penalty term, which is set to 0 in our experiments).

- For the active AQ_0^k, when neither the current action nor the next action (at the next time step) by CQ_0^k is *end*:

$$\Delta AQ_0^k(s, a) = \alpha(g(s) + \gamma \max_{a'} AQ_0^k(s', a') - AQ_0^k(s, a))$$

  where s' is the new state resulting from an action a in state s. When the current action by CQ_0^k is *continue* but the next action by CQ_0^k is *end*,

$$\Delta AQ_0^k(s, a) = \alpha(g(s) + \gamma \max_{aa'} AQ_1^j(s', aa') - p - AQ_0^k(s, a))$$

  where p is a penalty for moving up the hierarchy and s' is the new state resulting from action a in state s. AQ_1^j is the active module at the higher level at the next time step (s').

- For the active CQ_0^k, there are two separate learning rules for two different actions:

$$\Delta CQ_0^k(s, continue) = \alpha(g(s) + \gamma \max_{ca'} CQ_0^k(s', ca') - CQ_0^k(s, continue))$$

$$\Delta CQ_0^k(s, end) = \alpha(\max_{aa} AQ_1^j(s, aa) - p - CQ_0^k(s, end))$$

where s' is the new state resulting from an action by AQ_0^k in state s and p is a penalty for moving up the hierarchy. AQ_1^j is the active module at the higher level at the current time step.

- For the active AQ_i^k, where $i = 1, 2, ..., M$, we have the following learning rule, when both the current action and the next action by the corresponding CQ_i^k is *continue*:

$$\Delta AQ_i^k(s, aa) = \alpha(ag(s) + \gamma^m \max_{aa'} AQ_i^k(s', aa') - AQ_i^k(s, aa))$$

where aa is the abstract control action that selects a lower-level module, s' is the new state encountered (after aa is completed), m is the number of time steps taken to go from s to s' (used for determining the amount of discounting γ^m), and $ag(s)$ is the (discounted) cumulative reinforcement received after the current abstract control action for state s and before the next abstract control action for state s' (that is, $ag(s) = \sum_{k=0,1,....,m-1} g(s_k)\gamma^k$, where $s_0 = s$ and $s_m = s'$). When the current action by the corresponding CQ_i^k is *continue* but the next action by the CQ_i^k is *end*,

$$\Delta AQ_i^k(s, aa) = \alpha(ag(s) + \gamma^m \max_{aa'} AQ_{i+1}^j(s', aa') - p - AQ_i^k(s, aa))$$

where aa is the abstract control action that selects a lower-level module, s' is the new state encountered (after aa is completed). AQ_{i+1}^j is the active module at the higher level when s' is encountered.

- For the active CQ_i^k, where $i = 1, 2, ..., M$, we have the following learning rule for *end*:

$$\Delta CQ_i^k(s, end) = \alpha(\max_{aa} AQ_{i+1}^j(s, aa) - p - CQ_i^k(s, end))$$

where p is a penalty for moving up a hierarchy and AQ_{i+1}^j is the active module at the higher level at the current time step. We have the following learning rule for *continue*:

$$\Delta CQ_i^k(s, continue) = \alpha(ag(s) + \gamma^m \max_{ca'} CQ_i^k(s', ca') - CQ_i^k(s, continue))$$

where s' is the next state in which CQ_i^k is to make a decision, and m is the number of time steps taken to go from s to s'.

Note that, at different levels, we can set different learning rates and temperatures (for CQ_is and AQ_is). Note also that, although a two-level hierarchy can be easily simplified (if the same representation is used throughout) by examining the values of all the CQ's without using AQ at all in the selection of lower-level modules, a hierarchy of more than two levels cannot be easily collapsed into one level in this way. Hierarchical decision making has its advantages.

4 Analysis

4.1 Analysis of Non-Markovian Dependencies

Let us examine several special cases of non-Markovian dependencies that are particularly suitable for SSS. For simplicity, we will assume that a two-level hierarchy is used. We use

"non-Markovian tasks" to refer to tasks in which non-Markovian dependencies (as defined in section 1) exist, and "Markovian tasks" to refer to tasks in which no such dependencies exist.

1. Using Markovian subsequencing to turn non-Markovian tasks into Markovian ones. This is the simplest case of dealing with non-Markovian dependencies, which was addressed by a similar but more limited model proposed by Wiering and Schmidhuber (1998). An example used in Wiering and Schmidhuber (1998) is as follows: An agent is supposed to turn left at the first intersection, and then turn right at the second. Thus it is a non-Markovian sequence, as the second turn is dependent on the first turn (assuming that states are composed of visual indicators for intersections and other features). However, if we make a switch to a different (sub)sequence after the first turn, we then have a chain of two Markovian subsequences, instead of one non-Markovian sequence. Through the use of subsequencing, a non-Markovian sequence may become multiple Markovian sequences in a chain, as higher-level chaining provides the necessary context for local Markovian decision making. The action modules (for handling respective subsequences) have different policies, in accordance with the global context, i.e., their respective positions in a chain. [8]

   Segmentation that turns a non-Markovian sequence into multiple Markovian sequences in a chain can be accomplished by SSS through (implicitly) comparing overall reinforcement received through segmentation at different points. Both the higher-level and lower-level modules use atemporal representation in this case (i.e., without memory of any sort). Because a better way of segmentation will lead to a higher amount of overall reinforcement, eventually better ways of segmentation will likely dominate. [9] This is an extension of the approach of Wiering and Schmidhuber (1998). In Wiering and Schmidhuber's (1998) model, an a priori determined chain of modules is used, and each module has to select a subgoal and keep running until the goal is achieved (and cannot be flexibly switched off).

2. Using non-Markovian subsequencing to turn non-Markovian tasks into Markovian ones. When non-Markovian dependencies are dense but limited within small temporal segments (subsequences), if we can isolate and deal with separately these small segments, the overall task can be treated as Markovian. The difficulty lies in how we determine these local segments. Using a two-level structure in which the lower level uses temporal representation and the higher level uses atemporal representation, SSS can determine relevant subsequences through (implicitly) comparing overall reinforcement received through segmentation at different points. Segmentation at the proper points that separate local segments will result in the largest overall reinforcement (because such segmentation leads to capturing all the dependencies), and therefore those points will likely be adopted. The algorithm typically adjusts the lengths of subsequences in different modules to accommodate the different lengths of the non-Markovian dependencies in different local segments. [10]

[8] Clearly, not all non-Markovian sequences with long-range dependencies can be handled this way. The scenario represents a portion of non-Markovian situations.

[9] We cannot guarantee that though, because it is possible to be trapped in a "local minimum". The same caveat applies below.

[10] In addition, local non-Markovian processes can be different from each other, and thus the very seg-

We refer to such a process as "automatic temporal localization of non-Markovianness". We refer to sequences in which non-Markovian dependencies can be localized as having the "temporal locality property" (as opposed to the "spatial locality property" assumed in Dayan and Hinton 1993 for deriving their spatially hierarchical model). Although local non-Markovian dependencies can be handled by further segmentation without the use of temporal representation, it is unwieldy to do so, because too many segments may be generated and thus too many modules may be needed.

3. Using Markovian subsequencing and non-Markovian meta-sequencing for handling long-range (but sporadic) non-Markovian dependencies. Using a two-level structure in which the lower level uses atemporal representation and the higher level uses temporal representation, SSS can seek out and handle long-range dependencies. Under the assumption that long-range dependencies are sporadic, the higher-level process will determine those points in a sequence that are involved in a non-Markovian dependency, while the lower-level processes will handle Markovian processes in between. Ideally, the segmentation of subsequences occurs at those points that are involved in long-range dependencies. The SSS algorithm is capable of such segmentation, because it can (implicitly) compare overall reinforcement received through segmentation at different points and tend to find the best way(s) of segmentation that results in the largest reinforcement. Given the higher-level temporal representation and the lower-level atemporal representation, segmentation at those points that are involved in long-range dependencies will result in the largest overall reinforcement (because such segmentation leads to capturing all the dependencies without involving other states unnecessarily), and therefore those points will likely be adopted.

4. Separating the handling of coexisting short-range and long-range non-Markovian dependencies. Our approach can also be suitable for dealing with a mixture of long-range and short-range (global and local) non-Markovian dependencies. In the presence of both types of dependencies, SSS typically finds a way of separating different (global or local) dependencies and treat them separately. Using temporal representation at both the higher and lower level, the separation of the two types of dependencies is accomplished by comparing overall reinforcement received through segmentation at different points. The better ways of segmentation that treat global and local non-Markovian dependencies separately and correctly should result in more overall reinforcement eventually. This separation may have various advantages. For example, it may simplify the overall complexity of temporal representation. It may also facilitate the convergence of learning. [11]

In all, SSS learns to use both global and local contexts through seeking out proper configurations of temporal structures (global and local dependencies).

We characterize several different dimensions of non-Markovian temporal dependencies as follows, which may shed further light on these above special cases.

---

mentation into these local subsequences captures certain global non-Markovian dependencies (as in the first case).

[11] Local non-Markovian processes can be different from each other, and thus the segmentation per se captures certain (global) non-Markovian dependencies. So it is also possible to use atemporal representation at the higher level if such representation is sufficient to handle existent global non-Markovian dependencies (as in the first case).

- Degree of dependency: the maximum number of previous states that the current step is dependent on. In one extreme, the degree can be unlimited or infinite. In the other extreme, the degree can be zero, in which case there is no temporal dependency.
- Distance of dependency: the maximum number of steps between the current step and the furthest previous state that the current step is dependent on (given all the intermediate steps). In one extreme, the distance can be unlimited or infinite. In the other extreme, the distance can be zero.
- Density of dependency: the ratio between the degree of dependency and the distance of dependency.

According to these dimensions, (1) sporadic long-range dependencies involve long *distance* but low *density* temporal dependencies. Thus they can be handled by Markovian subsequencing and non-Markovian meta-sequencing, because such dependencies can be captured as segmentation points and handled by temporal representation at the higher level, which makes the lower level Markovian (because non-Markovian dependencies are of long distance and low density, they need not show up in lower-level processes). (Note that it is also possible that Markovian subsequencing and non-Markovian meta-sequencing can handle a non-Markovian task with long *distance*, high *density*, and high *degree* temporal dependencies, if subsequencing can insulate lower-level processes from such dependencies, as in the case of e.g. the parity problem when presented sequentially.) (2) Using Markovian subsequencing (without non-Markovian meta-sequencing) to turn a non-Markovian task into a Markovian task is a special case in which low-*density* (long- or short-*distance*) dependencies disappear when the task is properly segmented, due to, on the one hand, the localization of the lower-level processes and, on the other hand, the insulation of most of the states from the higher-level process. These processes likely become Markovian because the low-*density* temporal dependencies are likely absorbed by the chain of Markovian processes. (3) Using non-Markovian subsequencing to turn non-Markovian tasks into Markovian ones typically involves temporal dependencies that are of short *distance* (but maybe of high *density* and of high *degree*) and can be localized within the lower-level processes. (4) Separating the handling of coexisting global and local non-Markovian dependencies typically involves cases in which temporal dependencies are of varied *distances* that are preferably bifurcated (i.e., naturally separated into two categories: long-*distance* and short-*distance*), so that each type can be handled by a corresponding level.

4.2 Analysis of Difficulties

The major difficulties SSS faces are the following:

- Large search space. The simultaneous learning of multiple levels (involving abstract control, control, and individual action modules) results in a much larger search space in which the system has to find an optimal (or near optimal) configuration. In comparison, structurally pre-determined hierarchical RL has much smaller spaces to deal with.
- Cross-level interactions. Between any two adjacent levels, the higher-level control interacts with the lower-level control. In other words, the learning of the lower-level control is based on the higher-level control, but in turn the higher-level control need

also to adapt based on the learning and performance of the lower-level control. Thus, there are complex dynamic interactions (section 3.1). Structurally pre-determined hierarchies do not deal with such problems

- Within-level interactions. Interactions can occur among different modules at the same level too. Due to learning, the performance of one module can change and therefore affect other modules. While these other modules adapt to the change through learning, their outcomes in turn affect the original module. Again, there are complex, dynamic interactions.

There are in general three possible training regimes (all of which can be used in our model), from the computationally simplest, which reduces most the interactions between different components of a hierarchical system, to the most complex, as follows:

- Incremental elemental-to-composite learning (Lin 1993, Singh 1994): first train the system using only "elemental" tasks (with each elemental task being mapped to a particular module) and then, after the completion of the training on elemental tasks, train the system using composite tasks (which are composed of elemental tasks). There is in effect no autonomous self-segmentation in this case. This is therefore the easiest setting (thus SSS is not needed).

- Simultaneous elemental-and-composite learning (Singh 1994, Tham 1994): train the whole system using both elemental and composite tasks (each associated with a task label) simultaneously. In this case, the elemental tasks, learned along with composite tasks, in fact provide the clues for segmenting the composite tasks. So the segmentation in this case is not completely autonomous.

- Simultaneous composite learning: train all the components of the system simultaneously using the same composite tasks with the same reinforcement information from the environment; in other words, the system learns to decompose the task autonomously by itself. This is the most difficult setting and the focus of SSS.

Thus, although SSS can accommodate any of these methods, it does not require separate training for different modules, incremental training that goes from elemental to composite tasks, or simultaneous training of composite and elemental tasks (pre-segmentation). SSS instead relies on reinforcement for completely autonomous self-segmentation: It compares different amounts of reinforcement resulting from different ways of segmentation and tends to choose the best way(s) on that basis.

5 Experiments

5.1 Some Data

We choose three maze domains, which are representive tasks for reinforcement learning, for demonstrating our approach. The first domain is taken from McCallum (1996 b). It is chosen because it is simple but involves multiple possible segments, which is useful in illustrating our approach. The second domain is taken from Tadepalli and Dietterich (1997), which demonstrates the reuse of segments. The third domain is adopted from Wiering and Schmidhuber (1998). It is larger and used here to demonstrate how our approach fares

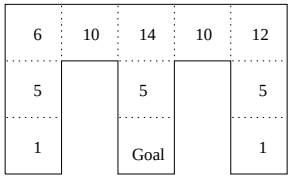

Figure 1: A maze requiring segmentation. Each number (randomly chosen) indicates an unique observational state as perceived by the agent.

in relation to the size of the state space and how shared subroutines develop. It involves both short-range and long-range non-Markovian dependencies and thus also enables the exploration of the interaction between the two types. In each domain, we will show that proper configurations of modules can be learned. For simplicity, we will use a two-level hierarchy.

5.1.1 Maze 1

In this maze, there are two possible starting locations and one goal location. The agent occupies one cell at a time and at each step obtains local information (observation) concerning the four adjacent cells (the left, right, above, and below cell) regarding whether each is an opening or a wall. It can make a move to any adjacent cell at each step (either go left, go right, go up, or go down). But if the adjacent cell is a wall, the agent will remain in the original cell at the end of the step. See Figure 1, where "1" indicates the starting cells. Each number indicates an observational state (i.e., an observation, not a true state) as perceived by the agent. In this domain, the minimum path length (i.e., the number of steps of the optimal path from either starting cell to the goal cell) is 6. When not all the paths are optimal, the average path length is a measure of the overall quality of a set of paths. The reinforcement structure (for applying Q-learning) is simple: The reward for reaching the goal location is 1, and no other reward (or punishment) is given.

The parameter settings for Q-learning modules are as follows: the Q value discount rate is 0.95, the initial learning rates for all modules ($\alpha^0_Q, \alpha^0_{AQ}, \alpha^0_{CQ}$) are uniformly 0.9, the learning rates change according to $\alpha^t = \alpha^{t-1}/t^{1/5}$ (where t is the number of episodes completed), the initial temperatures are $\tau^0_Q = 0.5$ and $\tau^0_{AQ} = \tau^0_{CQ} = 0.9$, and the temperatures change according to $\tau^t = \tau^{t-1}/t^{1/2}$ (where t is the number of episodes completed). The total number of training episodes is 5,000. The number of steps allowed in each episode during learning is 100 (an episode ends when the limit is reached, or as soon as the goal is reached)

As shown in Figure 1, the three cells marked as "5" are perceived to be the same by the agent. However, different actions are required in these cells in order to obtain the shortest paths to the goal. Likewise, the two cells marked as "10" are perceived to be the same but require different actions. In order to remove non-Markovian dependencies in each module, we can divide up each possible sequence (from one of the starting cells to the goal) into a number of segments so that, when different modules are used for these segments, a consistent policy can be adopted in each module.

As confirmed by the results of our experiments, a single agent (with atemporal representation) could not learn the task at all (the learning curve was flat), due to oscillations

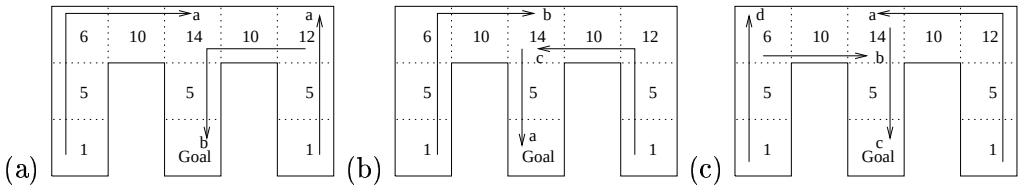

Figure 2: Different segmentations: (a) using two modules, (b) using three modules, (c) using four modules. Each arrowed line indicates an action (sub)sequence produced by a module, learned by SSS.

(and averaging) of Q values. Adding temporal representation (i.e., memory) may partially remedy the problem, but this approach has difficulty dealing with long-range dependencies (see earlier discussions) and, moreover, it is not comparable with SSS using atemporal representation. Thus it was not used here (but see the maze 3 experiments for a comparison when SSS uses temporal representation too). On the other hand, as confirmed by the experiments, SSS can segment sequences to remove non-Markovian dependencies. In this domain, it can be easily verified that a minimum of two Q/CQ pairs (modules) are needed in order to remove non-Markovian dependencies in each module. See Figure 2. However, finding the optimal paths using only two modules is proven difficult, because the agent has to be able to find switching points (between the two modules) that are exactly right for a sequence. In general, in this domain, the more modules there are, the easier the segmentation can be done (that is, the faster the learning is). This is because the more modules there are, the more possibilities (alternatives) there are for a proper segmentation that removes non-Markovian dependencies. [12] An ANOVA analysis (number of modules x block of training) shows that there is a significant main effect of number of modules ($p < 0.05$) and a significant interaction between number of modules and block of training ($p < 0.05$), which indicates that the number of modules has significant impact on the learning performance. The performance of the resulting systems is shown in Figure 3, under two conditions: using the completely deterministic policy in each module, or using a stochastic policy (with the Boltzmann distribution action selection with $\tau = 0.006$). In either case, as indicated by Figures 3, there are significant performance improvements in terms of percentages of the optimal path traversed and in terms of the average path length, when we go incrementally from 2 modules to 14 modules. Pairwise t tests on successive average path lengths confirmed this conclusion ($p < 0.05$). See Figure 3 for the test results.

The segmentation, as a result of learning using SSS, can be discerned from a set of Q, CQ, and AQ policies. An example set of policies that represent a particular segmentation learned in one run of SSS is shown in Figure 4.

As a variation, to reduce the complexity of this domain and thus to facilitate learning, we limit the starting location to one of the two cells. This setting is easier because there is no need to be concerned with the other half of the maze (see Figure 5). The learning performance is indeed enhanced by this change. We found that this change leads to better learning performance using either two, three, or four modules. Another possibilities for reducing the complexity is to always start with one module. Again, the learning performance is enhanced by this change. Therefore, the segmentation performance is determined to a large extent by the inherent difficulty of the domain that we deal with. The performance

[12] When there are more modules available, there are more possibilities for segmentation.

# of modules	Perc Optimal Path $\tau = 0$	Perc Optimal Path $\tau = 0.006$	Avg Path Length $\tau = 0.006$	Pairwise t test $T(> 2.33)$
2	32%	30%	17.32	
3	46%	44%	10.54	5.387
4	66%	59%	9.50	4.959
6	83%	70%	9.33	3.649
10	90%	76%	8.88	2.617
14	92%	81%	8.10	2.184
18	93%	84%	7.50	1.207
22	96%	87%	7.53	0.175

Figure 3: The effect of number of modules on the performance (after learning), in terms of the percentage of optimal paths and the average path length (with the maximum length of 24 imposed on each path).

```
State 1
 Module#0  Q: **UP(0.10)    RIGHT(0.00)    DOWN(0.00)    LEFT(0.00)
          CQ:          **CONTINUE(0.25)     END(0.12)
 Module#1  Q: **UP(0.30)    RIGHT(0.00)    DOWN(0.00)    LEFT(0.00)
          CQ:            CONTINUE(0.29)   **END(0.30)
 Module#2  Q: **UP(0.11)    RIGHT(0.00)    DOWN(0.00)    LEFT(0.00)
          CQ:          **CONTINUE(0.26)     END(0.22)
          AQ:   0(0.28)   1(0.06) **2(0.29)
State 5
 Module#0  Q: **UP(0.13)    RIGHT(0.00)    DOWN(0.08)    LEFT(0.00)
          CQ:            CONTINUE(0.30)   **END(0.31)
 Module#1  Q:   UP(0.24)    RIGHT(0.00)  **DOWN(0.37)    LEFT(0.00)
          CQ:          **CONTINUE(0.36)     END(0.16)
 Module#2  Q: **UP(0.14)    RIGHT(0.00)    DOWN(0.07)    LEFT(0.00)
          CQ:          **CONTINUE(0.32)     END(0.20)
          AQ:   0(0.25)   1(0.09) **2(0.30)
State 6
 Module#0  Q:   UP(0.00)  **RIGHT(0.17)    DOWN(0.09)    LEFT(0.00)
          CQ:          **CONTINUE(0.41)     END(0.16)
 Module#1  Q:   UP(0.00)    RIGHT(0.07)  **DOWN(0.23)    LEFT(0.00)
          CQ:            CONTINUE(0.23)   **END(0.40)
 Module#2  Q:   UP(0.00)  **RIGHT(0.17)    DOWN(0.10)    LEFT(0.00)
          CQ:            CONTINUE(0.39)   **END(0.41)
          AQ: **0(0.41)   1(0.10)   2(0.04)
State 10
 Module#0  Q:   UP(0.00)  **RIGHT(0.22)    DOWN(0.00)    LEFT(0.12)
          CQ:          **CONTINUE(0.51)     END(0.16)
 Module#1  Q:   UP(0.00)  **RIGHT(0.13)    DOWN(0.00)    LEFT(0.06)
          CQ:            CONTINUE(0.15)   **END(0.38)
 Module#2  Q:   UP(0.00)    RIGHT(0.15)    DOWN(0.00)  **LEFT(0.19)
          CQ:          **CONTINUE(0.46)     END(0.15)
          AQ: **0(0.37)   1(0.08)   2(0.06)
State 12
 Module#0  Q:   UP(0.00)    RIGHT(0.00)    DOWN(0.11)  **LEFT(0.13)
          CQ:            CONTINUE(0.25)   **END(0.39)
 Module#1  Q:   UP(0.00)    RIGHT(0.00)  **DOWN(0.24)    LEFT(0.08)
          CQ:            CONTINUE(0.23)   **END(0.40)
 Module#2  Q:   UP(0.00)    RIGHT(0.00)    DOWN(0.10)  **LEFT(0.16)
          CQ:          **CONTINUE(0.38)     END(0.24)
          AQ:   0(0.16)   1(0.07) **2(0.40)
State 14
 Module#0  Q:   UP(0.00)  **RIGHT(0.18)    DOWN(0.11)    LEFT(0.09)
          CQ:            CONTINUE(0.39)   **END(0.64)
 Module#1  Q:   UP(0.00)    RIGHT(0.14)  **DOWN(0.40)    LEFT(0.14)
          CQ:            CONTINUE(0.40)   **END(0.60)
 Module#2  Q:   UP(0.00)  **RIGHT(0.18)    DOWN(0.07)    LEFT(0.05)
          CQ:            CONTINUE(0.39)   **END(0.64)
          AQ:   0(0.26) **1(0.62)   2(0.05)
```

Figure 4: The learned values in different modules (as a result of SSS). For each state, the Q, CQ and AQ values of each module are listed, where ** indicates the best value of a module in a state.

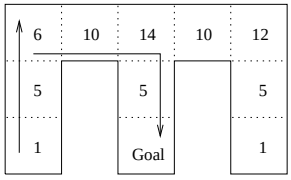

Figure 5: A possible segmentation using two modules, when only one starting location is allowed.

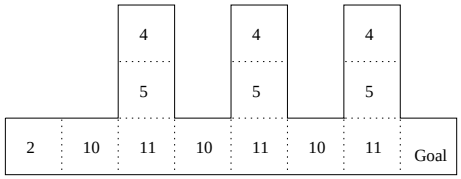

Figure 6: Maze 2: A maze requiring segmentation and reuse of modules. Each number (randomly chosen) indicates an observational state as perceived by the agent.

changes gradually in relation to the change in the complexity of the domain.

In all, the above experiments demonstrated that proper segmentation was indeed possible using SSS. Through learning a set of Markovian processes (with atemporal representations at both the higher and lower level), SSS removed non-Markovian dependencies and handled the sequences using a (learned) chain of Markovian processes. It was also clear from the experiments that it was not guaranteed that the segmentation (learning) process would succeed. The convergence to optimal solutions was probabilistic (as illustrated by Figure 3).

5.1.2 Maze 2

In this maze, there is one starting location at one end (marked as "2") and one goal location at the other. However, before reaching the goal location, the agent has to reach the top of each of the three arms. At each step, the agent obtains local observation concerning the four adjacent cells regarding whether each is an opening or a wall. See Figure 6, where each number indicates an observational state (an observation) as perceived by the agent. The agent can make a move to any adjacent cell at each step (either go left, go right, go up, or go down). But if the adjacent cell is a wall, the agent will remain in the original cell at the end of the step. The reward for reaching the goal location (after visiting the tops of all the three arms) is 1.

The parameter settings for the modules are as follows: the Q value discount rate is 0.97, the initial learning rates for all the modules ($\alpha^0_Q, \alpha^0_{AQ}, \alpha^0_{CQ}$) are uniformly 0.9, the learning rates change according to $\alpha^t = \alpha^{t-1}/t^{1/5}$ (where t is the number of episodes completed), the initial temperatures are $\tau^0_Q = 0.6$ and $\tau^0_{AQ} = \tau^0_{CQ} = 0.8$, and the temperatures change according to $\tau^t = \tau^{t-1}/t^{1/2}$. The total number of training episodes is 10,000. The number of steps allowed in each episode during learning is 200 (an episode ends when the limit is reached, or as soon as the goal is reached after having reached the tops of all the three arms).

In this domain, the shortest path consists of 19 steps. A minimum of two modules (two

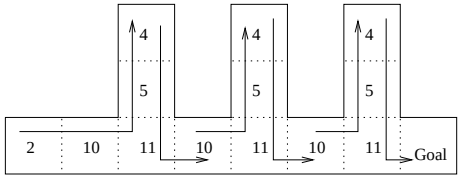

Figure 7: A segmentation using only two modules. Each arrowed line indicates an action sequence produced by a module, learned by SSS. The two modules alternate.

Q/CQ pairs) are needed (using atemporal input representation), in order to remove non-Markovian dependencies through segmentation and thus to obtain the shortest path. (As confirmed by the results of our experiments, a single agent with atemporal representation cannot learn the task at all.) With two modules, there is exactly one way of segmenting the sequence (considering only the shortest path to the goal): switching to a different module at the top cell of each arm (marked as "4") and switching again at the middle cell between any two arms (marked as "10"). See Figure 7. As confirmed by our experiments, this segmentation allows repeated use of the same modules along the way to the goal. Reuse of modules lead to the *compression* of sequences. When more modules are added, reuse of modules might be reduced: a third module, for example, can be used in place of the second use of module 1 (Figure 7).

As shown by the experiments, in this domain, unfortunately, learning does not become easier when we add more modules. This is because, in this domain, there is only exactly one way of segmentation (that is, switching at the top cell of each arm and at the cell between any two arms) that may lead to an optimal path, and thus there is no advantage in using more modules. Instead, we observed in the experiments a slight (but statistically significant) decrease of performance when more modules are added, in terms of both the learning process and the learned product. In general, the more modules we have, the slower the learning is. An ANOVA analysis (number of modules x block of training) shows that there is a significant main effect of number of modules ($p < 0.05$) and a significant interaction between number of modules and block of training ($p < 0.05$), which indicates that the number of modules has significant negative (detrimental) impact on learning. After learning, in terms of the performance of the resulting systems, we looked into both percentages of the optimal path (under both the deterministic and stochastic condition) and average path lengths (under both conditions). See Figures 9. The t test shows that the increases in average path lengths (i.e., the degradation of performance), corresponding to the increases from 2 to 3 modules and from 3 to 4 modules, are statistically significant. See Figure 8 for an example of the learned modules (when two Q/CQ pairs are used).

In all, the experiments in this domain further demonstrated the point made earlier regarding the feasibility of self-segmentation. Non-Markovian dependencies were removed through a set of Markovian processes (with atemporal representations). Furthermore, the experiments demonstrated the reuse of a module (in several different places) in a sequence or, in other words, the possibility of subroutines that could be called into use any number of times.

Reuse of a module occurs when two segments are similar enough in terms of their expected Q values. Reuse of a module at improper places leads to oscillations of values, which leads to the lower Q values in the module. Reuse in turn becomes less likely at such

```
State 2
 Module#0  Q:   UP(0.00)  **RIGHT(0.60)    DOWN(0.00)    LEFT(0.00)
          CQ:           **CONTINUE(0.70)     END(0.03)
 Module#1  Q:   UP(0.00)  **RIGHT(0.30)    DOWN(0.00)    LEFT(0.00)
          CQ:           **CONTINUE(0.04)     END(0.03)
          AQ: **0(0.72)   1(0.02)
State 4
 Module#0  Q:   UP(0.00)    RIGHT(0.00)  **DOWN(0.61)    LEFT(0.00)
          CQ:             CONTINUE(0.70)   **END(0.74)
 Module#1  Q:   UP(0.00)    RIGHT(0.00)  **DOWN(0.66)    LEFT(0.00)
          CQ:           **CONTINUE(0.74)     END(0.01)
          AQ:   0(0.70) **1(0.74)
State 5
 Module#0  Q: **UP(0.62)    RIGHT(0.00)    DOWN(0.61)    LEFT(0.00)
          CQ:           **CONTINUE(0.70)     END(0.12)
 Module#1  Q:   UP(0.64)    RIGHT(0.00)  **DOWN(0.66)    LEFT(0.00)
          CQ:           **CONTINUE(0.67)     END(0.10)
          AQ:   0(0.04) **1(0.05)
State 10
 Module#0  Q:   UP(0.00)  **RIGHT(0.75)    DOWN(0.00)    LEFT(0.60)
          CQ:           **CONTINUE(0.80)     END(0.04)
 Module#1  Q:   UP(0.00)    RIGHT(0.40)    DOWN(0.00)  **LEFT(0.66)
          CQ:             CONTINUE(0.74)   **END(0.79)
          AQ: **0(0.80)   1(0.03)
State 11
 Module#0  Q: **UP(0.62)    RIGHT(0.29)    DOWN(0.00)    LEFT(0.29)
          CQ:           **CONTINUE(0.72)     END(0.11)
 Module#1  Q:   UP(0.64)  **RIGHT(0.68)    DOWN(0.00)    LEFT(0.29)
          CQ:           **CONTINUE(0.77)     END(0.09)
          AQ: **0(0.15)   1(0.10)
```

Figure 8: The resulting modules after learning. For each state, the Q, CQ and AQ values of each module are listed, where ** indicates the best value of a module in the state.

# of modules	Perc Optimal Path $\tau = 0$	Perc Optimal Path $\tau = 0.001$	Avg Path Length $\tau = 0.001$	Pairwise t test $T(> 2.33)$
2	60%	52%	22.51	
3	43%	38%	23.85	8.600
4	27%	20%	24.85	4.010
5	27%	22%	25.42	1.281

Figure 9: The effect of number of modules on the performance (after learning), in terms of the percentage of optimal paths and the average path length (with a maximum length of 39).

places.

Reuse of modules has significant advantages. It leads to the compression of descriptions of sequences (related to the idea of minimum description length; Thrun and Schwartz 1995). Moreover, it allows the handling of sequences that cannot be (efficiently) handled otherwise. For example, the above domain could not be handled by simpler models in which segmentation was limited to a linear chaining (of a small number of modules) without any way for reusing modules (such as Wiering and Schmidhuber 1998). However, a simpler model might handle the domain by using six or more modules so that a chain of six modules could be formed. This approach resulted in inefficiency. While a minimum of two modules was required for SSS in this domain, Wiering and Schmidhuber's model required at least 6 modules in order to obtain the optimal path. If we increased the number of arms to, say, 10, then at least 20 modules would be required by Wiering and Schmidhuber's model, while two modules would still suffice with SSS. Yet another advantage is that after training with three arms, the SSS model can be applied to a maze with more arms without retraining.

5.1.3 Maze 3

In this maze, there is one starting location (marked as "S") and one goal location (marked as "G"). However, the agent has to reach the key (marked as "K") first in order to open the door (marked as "D"). Only after opening the door, can the agent reach the goal. See Figure 10. The agent has local information concerning the four adjacent cells regarding whether each is an opening or a wall, as well as the coordinates of S, K, D, and G (so that the locations of these entities can be varied without re-training the agent). But the agent does not have information regarding its own current location at each step. The agent can make a move to any adjacent cell at each step (either go left, go right, go up, or go down). The reward for reaching the goal location is 1, and no other reward (or punishment) is given.

The parameter settings for Q-learning in the modules are as follows: the Q value discount rate is 0.97, the initial learning rates for all modules ($\alpha^0_Q, \alpha^0_{AQ}, \alpha^0_{CQ}$) are uniformly 0.9, the learning rates change according to $\alpha^t = \alpha^{t-1}/t^{1/5}$ (where t is the number of episodes completed), the initial temperatures are $\tau^0_Q = 0.6$ and $\tau^0_{AQ} = \tau^0_{CQ} = 0.8$, the temperatures change according to $\tau^t = \tau^{t-1}/t^{1/2}$. The total number of training episodes is 10,000. The number of steps allowed in each episode during learning is 200. We randomly select a group of 5 cells in the maze as possible S locations, as well as sets of 5 cells as possible K, D and G locations respectively.

This domain involves both long-range and short-range non-Markovian dependencies. The long-range dependencies result from the requirement of reaching S, K, D and G sequentially in that order. However, by providing AQ with the S, K, D, and G information and then requiring AQ to select lower-level modules (which are to accomplish some subtasks), we can reduce the long-range dependencies to what are in effect short-range dependencies within AQ, because AQ can in effect skip all the intermediate steps. Shortening the distance of dependencies is one of the main advantages of using hierarchical reinforcement learning. On the other hand, the short-range dependencies (in the Q and CQ modules) are the results of the lack of current location information, because the observational input (relied on by the Q and CQ modules to make action decisions) cannot distinguish different cells with identical adjacent cell settings. In sum, all the modules (including Q's, CQ's, and AQ) must deal with non-Markovian dependencies.

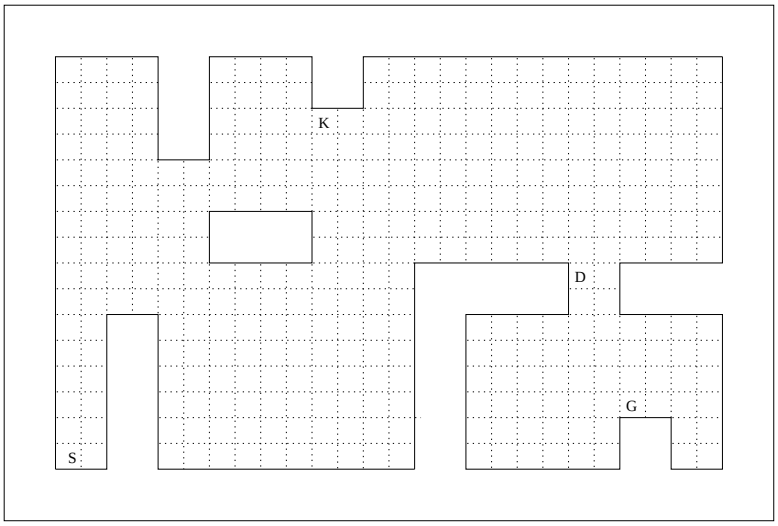

Figure 10: Maze 3: A maze involving both long- and short-range dependencies.

# of modules	Perc Optimal Path $\tau = 0$	Perc Optimal Path $\tau = 0.006$
1	73%	72%
2	85%	84%
3	92%	89%
4	95%	90%
5	95%	91%

Figure 11: The effect of number of modules on the performance (after learning), in terms of the percentage of optimal paths.

We used McCallum's (1996) method for constructing temporal representation to deal with non-Markovian dependencies within each module. This method was adopted because it appeared to be a better alternative compared with other methods (see section 3.2.4 for comparisons). Specifically, in each module, the method performs statistical tests of the differences in Q value distributions with regard to different state extensions (each of which consists of the current observational input plus some features of the preceding steps). The test used for distinguishing different state extensions is the well known Kolmogorov-Smirnov test. Learning was simplified in our experiments by imposing an order for testing preceding steps that goes backwards from the immediately preceding state/action pair to those furthest away (because dependencies in each module are in effect short-range).

The results showed that the more modules there were, the faster and better the learning was (in terms of average path lengths), up to a certain limit. (A maximum of 200 steps was allowed for each path.) An ANOVA analysis (number of modules x block of training) showed that there was a significant main effect of number of modules ($p < 0.05$) and a significant interaction between number of modules and block of training ($p < 0.05$), which indicated that the number of modules had significant positive impact on learning. After learning, in terms of the performance of the resulting systems, we looked into percentages of the optimal path (under both the deterministic and stochastic condition). The t tests showed that there were significant differences in these cases. See Figures 11.

It turns out that, after learning, different modules involved different history features, which is desirable because of the savings it entails (compared with brute force approaches that include all the history features, such as the n-fold state space). For example, through learning, AQ came to rely on the information concerning the locations of S, K, D, and G (so that it could select modules to go from S to K, from K to D, or from D to G based on that information) and, due to non-Markovianness, also used the history of past actions in order to make correct selections of modules. For example, only after choosing a module for going from S to K, then AQ chose a module to go from K to D, and so on. The Q and CQ modules, on the other hand, came to rely on the local information concerning four adjacent cells and they utilized past states in order to disambiguate locations to make correct action decisions for reaching their respective subgoals.

The separation of long-range and short-range dependencies (dealt with in AQ and Q/CQ respectively) resulted in efficiency and compactness of resulting policies. As shown by our experiments, if temporal representation was used in an one-module system, the acquisition of the temporal representation (using McCallum's method) took a very long time, since the temporal dependencies were of long distance in this domain. On the other hand, if atemporal representation was used in the lower-level modules (with segmentation), many more segments had to be created in order to remove all local temporal dependencies. Correspondingly, many more modules were needed.

In the results, shared subroutines were developed among several (sub)tasks, which also contributed to the compactness of the resulting representation. That is, instead of completely separate sets of modules being used for different (sub)tasks, some modules were shared by several different (sub)tasks, and thus fewer modules were needed. For example, when S and K in one episode were in the same pair of locations as K and D in another episode, the same module was called to go from S to K and to go from K to D in these two different episodes.

In sum, the above experiments demonstrated (1) the separation of long-range and short-range non-Markovian temporal dependencies (as analyzed in section 4), and (2) the possibility of developing shared subroutines, both of which resulted in efficiency and compactness of policies.

5.2 Some Observations

Let us discuss a few observations concerning SSS:

- The number of natural segments in a task vs. the number of modules: Our experiments indicate that in any domain, it is preferable that there are the same number of, or slightly more, modules as natural segments in order to make segmentation easier.

- Reuse of subsequences (subroutines) in a sequence: Reusable subsequences can be developed through SSS, and thus in segmenting a sequence, the same module may be used multiple times if similar situations are encountered multiple times. This capability can be very useful in developing a compact action policy, or a compact plan representation for hierarchical planning (Sun and Sessions 1998), in domains with recurring structures.

- Subsequences (subroutines) shared by multiple tasks: Modules can be automatically shared among multiple tasks (subtasks), as demonstrated by experiments in maze 3,

without extra complicated processes and mechanisms such as those used in Thrun and Schwartz (1995). SSS is a straightforward process to achieve a compact representation in domains with shared sequential segments.

- Convergence: From the experimental results, it is clear that SSS is not guaranteed to converge onto optimal policies. It might be futile to look for any convergence proof. However, empirically, it is highly likely, judging from the results, that SSS often achieves optimal policies in practice.

It appears that the algorithm is advantageous when there are temporal dependencies, especially long-range dependencies, but there are few internal states that are needed for disambiguation (that is, temporal dependencies are not of both long-distance and high-density). In such cases, a few modules, with limited or no memory, may work. The algorithm may fail (diverge or oscillate) when there are not enough modules to handle existent temporal dependencies in a domain (or when there is no sufficient exploration).

We did not use domains with probabilistic state transitions or noisy observations. It is not clear how SSS will fare in these circumstances. They remain to be explored in the future.

Note also that we did not experimentally compare SSS with e.g. Precup et al (1998), Dietterich (1997), and Parr and Russell (1997), because these algorithms require a substantial amount of a priori domain knowledge and thus are not comparable to SSS. On the other hand, although Parr and Russell (1995) and McCallum (1996b) deal with partial observability without requiring a priori knowledge, they do not segment sequences and develop hierarchies of subsequences, and thus are not comparable to SSS in that respect.

5.3 Plan Extraction

SSS enables the extraction of explicit hierarchical plans. In Sun and Sessions (1998), we proposed plan extraction algorithms that constructed explicit plans from the value functions obtained by reinforcement learning. [13] Explicit plans perform open-loop (or semi-open-loop) execution with no (or much less) environmental feedback; that is, they either do not rely on sensing the current state at each step, or rely much less on such sensing. Explicit plans can be more useful than closed-loop policies in many situations, as discussed in Sun and Sessions (1998), due to their ability for explicit sequencing and open-loop execution. Plan extraction also leads to "compression" of policies (Sun and Sessions 1998) and significant savings in representation.

Basically, our plan extraction algorithm applies an A*-like beam search process that focuses on most likely paths from the starting state to the goal state, based on Q values learned through RL, and collects the best actions for the states encountered along the way into an explicit sequence. For further details, see Sun and Sessions (1998). [14] Applying the plan extraction algorithm to maze 2, we extracted the following hierarchical plan:

[13] RL algorithms such as Q-learning only learn closed-loop policies, i.e., implicit (reactive) plan, not an explicit plan that can be used in an open-loop (or semi-open-loop) fashion (Barto et al 1995): It does not work in such a way that, to achieve a goal, actions are selected to form a sequence ahead of the execution time and run without (or with minimum) feedback from the environment.

[14] The algorithm has been analyzed with regard to its essential theoretical properties. It is complete and sound given certain conditions (see Sun and Sessions 1998 for details of the analysis).

Plan: (1) go right; (2) call SubPlan 1; (3) call SubPlan 2; (4) call SubPlan 1; (5) call SubPlan 2; (6) call SubPlan 1; (7) call SubPlan 2.

SubPlan 1: (1) go right; (2) go up; (3) go up.

SubPlan 2: (1) go down; (2) go down; (3) go right.

Hierarchical plan extraction will not be possible in this domain without segmentation. Note that here both reinforcement learning and subsequent plan extraction are based on autonomous learning without a priori knowledge and without pre-determined structures.

6 Discussions

Hierarchical reinforcement learning. There are many existing models of hierarchical reinforcement learning (Kaelbling et al 1996). In examining different ways of doing hierarchical RL, we can separate two cases: (1) the use of structurally pre-determined domain-specific hierarchies and (2) automatic building of hierarchies.

In RL, there are a number of existing approaches that assume a fixed domain-specific hierarchy in place before learning starts. Mahadevan and Connell (1992) used a hand-coded behavior-based ("subsumption") hierarchy in which each module was predestined for a particular subtask by receiving a separate special reward. Similarly, Humphrys (1996) used pre-wired, differential input features and reward functions for different modules (although GA might be used to optimize reward functions for different modules). In the hierarchical models of Dayan and Hinton (1993), Parr and Russell (1997), and Dietterich (1997), although learning at every level of hierarchies was involved, domain-specific hierarchical structures were pre-constructed from a priori knowledge. Sutton (1995) similarly included an a priori determined mixture of models at different levels of abstraction to model the world at different time scales (see also Precup et al 1998). Some of these models (e.g., Sutton 1995 and Parr and Russell 1997) do not deal with non-Markovian dependencies, while some others deal with the issue non-adaptively (e.g, Precup et al 1998). Wiering and Schmidhuber (1998) independently developed a scheme similar to SSS. However, they applied a fixed structure that chained lower-level modules (although the lower-level modules were adapted).

We can distinguish two directions in automatically building hierarchies: upward and downward building of hierarchies. For example, Ring (1991) constructed hierarchies from the bottom up by using the lowest-level nodes to represent primitive actions and then setting up nodes at higher levels to represent sequences of actions through chaining lower-level nodes (see also Weiss 1995). The model constructed only chains, based solely on sequential occurrence of events. Lin (1993) used a pre-training scheme, which trained a system (especially its lower-level modules) on "elemental" tasks first and then trained the whole system on composite tasks that consisted of constituent elemental tasks. Singh (1994) developed the CQ-L model, which trained lower-level modules using stand-alone elemental tasks while applying ML-based gating methods to learn to assign components (i.e., constituent elemental tasks) of composite tasks to these modules. Tham (1995) applied EM-based gating methods in a similar way. Rosca and Ballard (1996) used evolutionary programming techniques for developing more and more complex subroutines. In contrast, Moore and Atkeson (1994) created resolution hierarchies by exploring the state space in domain-specific ways and splitting relevant states (when needed) so that lower levels with

finer resolutions can be formed. Similarly, Sun and Peterson (1999) used an algorithm that splits regions of the input space that have "inconsistent" errors and thus built hierarchies of the input space regions downward.

Our approach in the present work is automatically building hierarchies simultaneously at multiple levels, and thus it is neither upward nor downward. Our approach of building hierarchies does not use separate rewards for different modules or different levels (Mahadevan and Connell 1992, Humphrys 1996), separate training for modules at different levels (Lin 1993, Tham 1995), or pre-segmentation into elemental tasks (Singh 1994, Tham 1995). We do not require structurally pre-determined hierarchies (based on a priori knowledge) either (Dayan and Hinton 1993, Dietterich 1997, Parr and Russell 1997). Our approach also does not involve explicit use of global measures (and the optimization of such measures) as in Roy et al (1996) and Kaelbling (1993).

SSS uses one and only one principle in automatically forming hierarchies, that is, the maximization of expected reinforcement, which is used in all of the following aspects: learning action policies, learning control policies, and learning abstract control policies. This is in contrast with work that uses additional, auxiliary principles in forming hierarchies; for example, the principle of description length minimization as in Thrun and Schwartz (1995), secondary error terms as in Schmidhuber (1992), or entropy measures as in Schmidhuber (1993). [15]

Comparing with the similar but separately developed work of Wiering and Schmidhuber (1998), beside being more general (e.g., allowing more than two levels and not limited to a single chain), SSS is more flexible in that it does not have to wait for the currently active lower-level module to achieve its goal (that is, in SSS, a module learns to end at multiple possible places). The control learns to terminate a module at the right point based on (on-line) comparison of overall reinforcement that can be received through termination at different points. Our approach does not rely on the assumption that each module achieves one specific goal (which is not always possible).

Of course, the more general a problem is, the more difficult it is to learn. Therefore, we cannot expect the performance of SSS to be better than algorithms that rely more on domain-specific knowledge. Note that, even for SSS, some decisions regarding the algorithm have to be made prior to learning (such as setting parameter values). Another limitation is that SSS only segments action sequences but does not perform state abstraction.

Hierarchical planning. Hierarchical reinforcement learning is clearly related to the idea of hierarchical planning (Sacerdoti 1974), which has been prominent in the planning literature. In hierarchical planning, an abstract plan (or solution) is proposed first and the solution is subsequently reduced into more primitive plans or procedural structures before it is executed. Bacchus and Yang (1994) studied the formal properties of hierarchical decomposition in which abstract operators were refined into more concrete operators iteratively until primitive operators were obtained, and identified the conditions under which such decomposition was possible. Pfleger and Hayes-Roth (1996) proposed the idea of plans as "intention". Plans that represent intention were expressed as constraints rather than abstract actions. This view on planning did not require the a priori specification of the refinement of abstract operators, so as to allow an agent to exploit, during the execution in accordance with the plan, knowledge acquired separately as well as particular charac-

[15] Schmidhuber (1992, 1993) are not directly comparable to our work, because they deal with supervised learning.

teristics of environments that were encountered. In contrast, our hierarchical organization is neither fixed reduction of operators, nor mere expression of constraints. It is learned stochastic "reduction" of stochastic procedures (not a priori specified), and thus it is more flexible (in the sense that there is no pre-determined steps, goals, or routines) than operator reduction, but more structured than constraint specifications.

Long-range dependencies. An important current issue in machine learning is how to deal with long-range (non-Markovian) temporal dependencies. One may use a priori domain-specific knowledge to structure models in domain-specific ways to enable them to deal with such dependencies, as in Frasconi et al (1995). One may also use unexpected events to construct a higher level in a model so that the higher level can handle dependencies that are more remote, as in Schmidhuber (1992). Or, one may construct a chain of Markovian processes to reduce some long-range dependencies, as in Wiering and Schmidhuber (1998). In contrast, we use overall reinforcement as a criterion for segmentation to build hierarchies. Compared with existing work dealing with long-range dependencies, the present approach is relatively general and non-domain-specific.

Models for POMDPs. Non-Markovian dependencies are often created by partial observability in POMDPs. Lin (1993) used recurrent neural networks for representing states with compressed history traces captured in hidden layers. The practical problem with the approach is that long time lags often weaken the trace in hidden layers (due to limited precisions) and blur the distinction of states. McCallum (1996) used decision tree procedures to split a state when it was "inconsistent" by incorporating additional history features. Wiering and Schmidhuber (1998), as reviewed earlier, could handle some special cases of POMDPs where subgoaling could eliminate non-Markovian dependencies with the use of a chain of subgoals. In contrast, our algorithm uses a combination of subsequencing (similar to and extending "subgoaling" as in Wiering and Schmidhuber 1998) and temporal representation (as in Lin 1993 and McCallum 1996, but only for subsequences), as a means for simplifying learning and dealing with more general types of non-Markovian dependencies (e.g., more general than those handled in Wiering and Schmidhuber 1998). [16]

The advantages of SSS. We summarize the advantages as follows:

- In SSS, there is no need for pre-determined hierarchies, a priori domain-specific structures to help with the formation of hierarchies, domain-specific built-in ways for creating hierarchies, and so on. In sum, there is no requirement of a priori domain-specific knowledge in forming hierarchies (although such knowledge may be beneficially incorporated when available).

- SSS uses only one principle in forming hierarchies (that is, the maximization of expected reinforcement).

- SSS seeks out somewhat proper (and sometimes optimal) configurations of hierarchical structures, in correspondence with temporal dependency structures of a domain (extending Wiering and Schmidhuber 1998).

- SSS is able to develop subroutines, which can potentially be reused at different points of a sequence (unlike Wiering and Schmidhuber 1998), or shared by multiple (different) sequences (as in Thrun and Schwartz 1995).

[16] Other methods for dealing with POMDPs include Ring (1991), Chrisman (1993), Cassandra et al (1994), and Parr and Russell (1995). The costs of these methods are high. See also Sondik (1978), Monahan (1982), and Puterman (1994).

- In relation to classical planning, with hierarchical structuring, SSS can be viewed as performing planning at different levels of abstraction (Knoblock et al 1994), with lower-level modules being macro-actions.

Future work. There is much further work that can be done along the line outlined here. For example, we need to scale the algorithm up to deal with much larger state spaces, in which function approximation or other schemes must be used to handle the complexity of Q values, and the speed of convergence must be dealt with. We need to look into domains with probabilistic state transitions and/or noisy observations. We also need to systematically examine the interplay of long-range and short-range temporal dependencies, through experiments, by gradually varying the degree, distance, and density of such dependencies in a systematic way. We may also want to test systems with more than two levels and to adaptively determine the number of levels in such systems through learning, in the same way as the adaptive determination of segmentation points within each level. Above all, we need to perform detailed experimental or mathematical analyses of convergence properties of the algorithm.

References

F. Bacchus and Q. Yang, (1994). Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*. 71, 1, 43-100.

D. Bertsekas and J. Tsitsiklis, (1996). *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA.

A. Cassandra, L. Kaelbling, and M. Littman, (1994). Acting optimally in partially observable stochastic domains. *Proc. of 12th National Conference on Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA.

L. Chrisman, (1993). Reinforcement learning with perceptual aliasing: the perceptual distinction approach. *Proc. of AAAI*. 183-188. Morgan Kaufmann, San Mateo, CA.

P. Dayan and G. Hinton, (1993). Feudal reinforcement learning. *Advances in Neural Information Processing Systems*. MIT Press, Cambridge, MA.

T. Dietterich, (1997). Hierarchical reinforcement learning with MAXQ value function decomposition. *http* : *//www.engr.orst.edu/* $\sim$ *tgd/cv/pubs.html*.

J. Elman, (1990). Finding structure in time. *Cognitive Science*. 14, 179-212.

P. Frasconi, M. Gori, and G. Soda, (1995). Recurrent neural networks and prior knowledge for sequence processing. *Knowledge Based Systems*. 8, 6, 313-332.

C.L. Giles, B.G. Horne, and T. Lin, (1995). Learning a class of large finite state machines with a recurrent neural network. *Neural Networks*, 8 (9), 1359-1365.

M. Humphrys, (1996). W-learning: a simple RL-based society of mind. Technical report 362, University of Cambridge, Computer Laboratory.

L. Kaelbling, (1993). Hierarchical learning in stochastic domains: preliminary results. *Proc. of ICML*, 167-173. Morgan Kaufmann, San Francisco, CA.

L. Kaelbling, M. Littman, and A. Moore, (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4, 237-285.

C. Knoblock, J. Tenenberg, and Q. Yang, (1994). Characterizing abstraction hierarchies for planning. *Proc of AAAI'94*. 692-697. Morgan Kaufmann, San Mateo, CA.

Y. Kuniyoshi, M. Inaba, and H. Inoue, (1991). Learning by watching: extracting reusable task knowledge from visual observation of human performance. *IEEE Transactions on Robotics and Automation*.

L. Lin, (1993). *Reinforcement Learning for Robots Using Neural Networks*. Ph.D. Thesis, Carnegie Mellon University, Pittsburgh.

S. Mahadevan and J. Connell (1992). Automatic programming of behavior-based robot with reinforcement learning. *Artificial Intelligence*, 55, 311-365. 1992.

A. McCallum, (1996). Learning to use selective attention and short-term memory in sequential tasks. *Proc. Conference on Simulation of Adaptive Behavior*. 315-324. MIT Press, Cambridge, MA.

A. McCallum, (1996 b). *Reinforcement Learning with Selective Perception and Hidden State*. Ph.D Thesis, Department of Computer Science, University of Rochester, Rochester, NY.

G. Monohan, (1982). A survey of partially observable Markov decision processes: theory, models, and algorithms. *Management Science*, 28 (1), 1-16.

A. Moore and C. Atkeson, (1994). The parti-game algorithm for variable resolution reinforcement learning in multidimensional state spaces. *Machine Learning.*

C. Nevill-Manning and I. Witten, (1997). Identifying hierarchical structure in sequences: a linear-time algorithm. *Journal of Artificial Intelligence Research*, 7, 67-82.

R. Parr and S. Russell, (1995). Approximating optimal policies for partially observable stochastic domains. *Proc. of IJCAI'95.* 1088-1094. Morgan Kaufmann, San Mateo, CA.

R. Parr and S. Russell, (1997). Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems 9.* MIT Press, Cambridge, MA.

D. Precup, R. Sutton, and S. Singh, (1998). Multi-time models for temporary abstract planning. *Advances in Neural Information Processing Systems 10.* MIT Press, Cambridge, MA.

K. Pfleger and B. Hayes-Roth, (1997). Plan should abstractly describe intended behavior. *Proc. JCIS*, 1, 29-33. Duke University Press, Durham, NC.

M. Puterman, (1994). *Markov Decision Processes.* Wiley-Inter-science. New York.

M. Ring, (1991). Incremental development of complex behaviors through automatic construction of sensory-motor hierarchies. *Proc. of ICML.* 343-347. Morgan Kaufmann, San Francisco, CA.

J. Rosca and D. Ballard, (1996). Evolution-based discovery of hierarchical behavior. *Proc. of AAAI-96.* MIT Press. Cambridge, MA.

A. Roy, S. Govil, and R. Miranda, (1996). A neural network learning theory and a polynomial time RBF algorithm. Manuscript.

E. Sacerdoti, (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence.* 5, 115-135.

J. Schmidhuber, (1992). Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4 (2), 234-242.

J. Schmidhuber, (1993). Learning unambiguous reduced sequence descriptions. *Advances in Neural Information Processing Systems*, 291-298.

S. Singh, (1994). *Learning to Solve Markovian Decision Processes.* Ph.D Thesis, University of Massachusetts, Amherst, MA.

S. Singh, T. Jaakkola, and M. Jordan, (1994). Reinforcement learning with soft state aggregation. In: S.J. Hanson J. Cowan and C. L. Giles, eds. *Advances in Neural Information Processing Systems 7.* Morgan Kaufmann, San Mateo, CA.

E. Sondik, (1978). The optimal control of partially observable Markov processes over the infinite horizon: discounted costs. *Operations research*, 26 (2).

R. Sun and T. Peterson, (1999). Multi-agent reinforcement learning: weighting and partitioning. *Neural Networks*, Vol.12 No.4-5. pp.127-153. A shortened version in *Proceedings of ICONIP'98*, Springer-Verlag. Berlin.

R. Sun and C. Sessions, (1998). Learning plans without a priori knowledge. Submitted for publication. A shortened version in *Proceedings of WCCI-IJCNN'98*, vol.1, 1-6. IEEE Press, Piscateway, NJ.

R. Sutton, (1995). TD models: modeling the world at a mixture of time scales. *Proc. of ICML*. Morgan Kaufmann, San Francisco, CA.

P. Tadepalli and T. Dietterich, (1997). Hierarchical explanation-based reinforcement learning. *Proc. International Conference on Machine Learning*. 358-366. Morgan Kaufmann, San Francisco, CA.

C. Tham, (1995). Reinforcement learning of multiple tasks using a hierarchical CMAC architecture. *Robotics and Autonomous Systems*. 15, 247-274.

S. Thrun and A. Schwartz, (1995). Finding structure in reinforcement learning. *Neural Information Processing Systems*. MIT Press, Cambridge, MA.

C. Watkins, (1989). *Learning with Delayed Rewards*. Ph.D Thesis, Cambridge University, Cambridge, UK.

G. Weiss, (1995). Distributed reinforcement learning. *Robotics and Autonomous Systems*, vol.15, no.1-2, 135-142.

S. Whitehead and L. Lin, (1995). Reinforcement learning of non-Markov decision processes. *Artificial Intelligence*. 73 (1-2). 271-306.

M. Wiering and J. Schmidhuber, (1998). HQ-learning. *Adaptive Behavior*, 6 (2), 219-246.