Mach Learn (2010) 81: 283–331
DOI 10.1007/s10994-010-5182-y

Improving reinforcement learning by using sequence trees

Sertan Girgin · Faruk Polat · Reda Alhajj

Received: 11 September 2007 / Revised: 16 March 2010 / Accepted: 17 March 2010 /
Published online: 24 April 2010

Abstract This paper proposes a novel approach to discover options in the form of stochastic conditionally terminating sequences; it shows how such sequences can be integrated into the reinforcement learning framework to improve the learning performance. The method utilizes stored histories of possible optimal policies and constructs a specialized tree structure during the learning process. The constructed tree facilitates the process of identifying frequently used action sequences together with states that are visited during the execution of such sequences. The tree is constantly updated and used to implicitly run corresponding options. The effectiveness of the method is demonstrated empirically by conducting extensive experiments on various domains with different properties.

Keywords Reinforcement learning · Options · Conditionally terminating sequences · Temporal abstractions · Semi-Markov decision processes

1 Introduction

Reinforcement learning (RL) is the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment; the agent receives perceptions from the environment, takes actions based on its perceptions, and receives (immediate) rewards in return (Littman et al. 1996; Sutton and Barto 1998). In most realistic and complex domains, the task that the agent is trying to solve is composed of various sub-tasks and has a hierarchical structure formed by the relations among them (Barto and Mahadevan

Editor: R. Khardon.

S. Girgin · F. Polat
Department of Computer Engineering, Middle East Technical University, Ankara, Turkey

R. Alhajj (✉)
Department of Computer Science, University of Calgary, Calgary, Alberta, Canada
e-mail: alhajj@ucalgary.ca

R. Alhajj
Department of Computer Science, Global University, Beirut, Lebanon

2003). Each of these sub-tasks repeats many times at different regions of the state space. Although, all instances of the same sub-task, or similar sub-tasks, have almost identical solutions (sub-behaviors), without any (self-) guidance, an agent has to learn these solutions independently of each other, by going through similar learning stages again and again. This situation affects the learning process in a negative way, making it difficult to converge to optimal behavior in a reasonable time.

We argue that the main reason for the problem is the lack of abstractions that would allow to share solutions between similar sub-tasks. *Temporally abstract actions*, or options, are macro actions that generalize primitive actions and last for more than one time step (Parr and Russell 1998; Precup et al. 1998; Hauskrecht et al. 1998; Sutton et al. 1999; Dietterich 2000; McGovern and Sutton 1998; Barto and Mahadevan 2003). In most applications, they are part of the problem specification; they are provided by the system developer prior to learning process. However, this necessitates extensive domain knowledge and defining them becomes a more difficult task as the complexity of the problem increases. An alternative and probably more convenient way is to construct macro actions *automatically*, without requiring any prior explicit definitions or user intervention, based on the acquired domain information as the learning process progresses—a problem known as *option discovery*.

Motivated by the shortcomings of the existing approaches for option discovery, as highlighted in Sect. 2, this paper proposes a method which efficiently discovers useful options in the form of a single meta-abstraction solely based on the experience acquired by the agent during the learning process. We first extend the *conditionally terminating sequences* (CTS) of McGovern (1998, 2002) in order to make a better use of the hierarchical decomposition inherent in the reinforcement learning problem and to add the ability to follow different courses of actions during execution. This extension essentially encapsulates the behavior of a given set of conditionally terminating sequences; it is realized using a particular computational structure, a *sequence tree*. We formalize the notion of a sequence tree through a novel abstraction in the form of *stochastic conditionally terminating sequences*. We investigate the case where the conditionally terminating sequences are not known in advance but need to be generated during the learning process. From the history of observed events, trajectories of possible optimal policies are generated and stored in a modified version of a sequence tree. This tree contains additional eligibility and reward attributes which enable the identification and compact representation of frequently used action sequences together with states that are visited during their execution. Throughout the learning process, this tree is constantly updated and used to implicitly run represented abstractions. The proposed method can be integrated into other reinforcement learning algorithms as a meta-heuristic. We demonstrate the effectiveness of the approach by reporting test results on three domains, namely six-room maze, Dietterich's Taxi problems and keepaway sub-task of robotic soccer. The results show that the proposed method attains substantial level of improvement when used in conjunction with widely used reinforcement learning algorithms. In addition, we compare our work with acQuire-macros, the option framework of McGovern (1998, 2002).

The rest of the paper is organized as follows. Section 2 presents an overview of the related work. Section 3 presents the standard reinforcement learning framework of discrete time finite Markov decision processes. Section 4 describes the notion of options. Section 5 covers conditionally terminating sequences, extends them into sequence trees that have higher representational power, and introduces a novel method to discover useful abstractions based on sequence trees. Experimental results are reported in Sect. 6. Section 7 presents our conclusions.

2 Related work

Most of the research on option discovery relies on two main approaches. In the *subgoal-based* approach, possible sub-goals of the problem are identified first and then sub-policies solving them are found; these sub-policies are converted into abstractions and added to the action set of the agent. The works of Digney (1998), and Stolle and Precup (2002) use a statistical approach and define sub-goals as states that are visited frequently or have a high reward gradient. McGovern and Barto (2001) select as sub-goals the most diversely dense regions of the state space, i.e., the set of states that are visited frequently on successful experiences, where the notion of success is problem dependent.

Assuming that the state space is uniformly connected, i.e. all states have approximately the same number of states with transition probability greater than 0, Goel and Huber (2003) and Asadi and Huber (2005) consider states that lie in a substantially larger number of paths compared to their predecessors as potential sub-goals and determine them using Monte Carlo sampling. In their connection-based approach, Chen et al. (2007) propose a localized version of this idea in which only the states whose all neighbors possess this property are eligible for being sub-goals. The main target of HEXQ algorithm by Hengst (2002) is to automatically construct a task hierarchy based on the change in the values of state variables and the premise that variables which change more frequently retain their transition properties in the context of variables that change less frequently. Starting from the variable that changes most frequently, a level of hierarchy is generated for each variable, and at each level the states that are represented by the values of the corresponding variable are partitioned into regions; the regions are identified by state-action pairs (called *exits*) that cause unpredictable transitions; separate policies that leave each region through its exists form the temporal abstractions. In the factored reinforcement learning setting, TexDYNA algorithm of Kozlova et al. (2009) simultaneously decomposes a factored MDP into a set of options and improves incrementally the local policy of each option by using a particular decision tree based instance of the model-based reinforcement learning framework SDYNA (Degris et al. 2006). In their approach, the options are determined by exits as in HEXQ, but the variable whose values determine the context itself are explicit in the exit definition; furthermore, to ensure their relevance exits are updated every time the model of transitions change.

Jonsson and Barto (2001) adapt McCallum's U-Tree algorithm to automatically build option-specific representation of the state feature space. Other researchers, e.g., Menache et al. (2002), follow a graph theoretical approach and their Q-cut algorithm uses a maximum-flow/minimum-cut algorithm to partition the graph derived from the history of state transitions, and the sub-goals are defined as the bottleneck states that connect the strongly connected components of this graph. Kazemitabar and Beigy (2009) also focus on strongly connected components of the state transition graph, but based on the observation that the constructed graph is sparse in most reinforcement learning problems, they employ depth-first search algorithm with adjacency list representation to find the bottleneck states in linear-time. Simsek and Barto (2004, 2005) use a similar definition of sub-goals, but they propose two methods for searching them locally. In the first method, the sub-goal discovery problem is formulated as a classification problem in which the classification criteria is the *relative novelty* of states (Simsek and Barto 2004); the novelty of a state is defined as the frequency of visits since a designated start time, and its relative novelty is the ratio of the novelty of states that followed that state in a transition sequence to the novelty of the states that preceded it. Sub-goal states usually have higher relative novelty values which can be used to differentiate them. In their second method, they formulate the problem as the partitioning of local state transition graphs that reflect only the most recent experiences of the agent (Simsek et al. 2005). In another graph based method due to Mannor et al. (2004), the state space is

partitioned into different regions using a clustering algorithm (based on the graph topology or state values), and the policies that move the agent between different regions in the state space are transformed into macro actions. Once the sub-goals are identified, the methods mentioned above generate abstractions corresponding to the discovered sub-goals by using auxiliary reinforcement learning processes such as *action replay* (Lin 1992) executed on the corresponding restricted sub-problems with artificial rewards. Although these methods are effective in general, when the problem to be solved contains sub-tasks with similar policies but different sub-goals, they consider these sub-tasks independently of each other leading to multiple abstractions in which the resulting behaviors of the agent are quite similar.

In relatively less explored second approach, temporal abstractions are generated directly, that is without identifying sub-goals, by analyzing the common parts of multiple policies. An example of this approach is proposed by McGovern (1998, 2002). The *acQuire-macros* algorithm begins with detecting action sequences that occur frequently on successful trajectories, eliminates sequences leading to similar results by applying a static filter, and finally creates options for the remaining ones. One drawback of this method is that common action sequences are identified at regular intervals, which is a costly operation and requires storing all state and action histories since the beginning of the learning process. Also, since every prefix of an action sequence has at least the same support as that action sequence (i.e., occurs at least as many times as the action sequence itself), the number of possible options increases rapidly unless limited in a problem specific way (in this case, realized through the use of a static filter). Recently, Zang et al. (2009) proposed an algorithm called *Oplearn* to discover options from example trajectories that also make use of frequently occurring action sequences. However, instead of directly building options on them, for each frequent action sequence they first determine its context (i.e. the set of state variables that are needed for abstraction), and then they identify possible sub-problem goals by extending the instances of the sequence in sample trajectories until a change in the context is observed, that is, one or more extra state variables are also needed. The sub-problems are scored by estimating the computational gain that they would introduce for solving the base problem; the SMDP corresponding to the sub-problem with the highest score is solved by value iteration and the resulting policy is used to define a new option. This procedure is repeated recursively to build a hierarchical decomposition until no other useful sub-problems can be found. Although action sequences are used to identify the sub-problems via sub-goals, the context of each sub-problem goal depends on the change in the values of the state variables, and overall *Oplearn* falls in the category of subgoal-based approaches.

3 Background

In this section, we introduce the background necessary to understand the material presented in this paper. We start by defining Markov decision processes and the reinforcement learning problem. The experienced reader can skip the introductory material and move directly to Sect. 5.

A *Markov decision process*, denoted MDP, is a tuple $\langle S, A, T, R\rangle$, where S is a finite set of states, A is a finite set of actions, $T : S \times A \times S \to [0, 1]$ is a state transition function such that $\forall s \in S, \forall a \in A$, $\sum_{s' \in S} T(s, a, s') = 1$, and $R : S \times A \to \Re$ is a reward function. $T(s, a, s')$ denotes the probability of making a transition from state s to state s' by taking action a. $R(s, a)$ is the *immediate* expected reward received by the agent when action a is executed in state s. A (stationary) *policy*, $\pi : S \times A \to [0, 1]$, is a mapping that defines the probability of selecting an action from a particular state. If $\forall s \in S$, $\pi(s, a_s) = 1$ and $\forall a \in A$, $a \neq a_s$, $\pi(s, a) = 0$ then π is called a *deterministic policy*. The *value* of a

state s under policy π, $V^\pi(s)$, is the expected infinite discounted sum of rewards that the agent will gain if it starts in state s and follows π (Littman et al. 1996). It is computed as $V^\pi(s) = \sum_{t=0}^{\infty} \gamma^t E(r_t|\pi, s_0 = s)$, where r_t is the reward received at time t, and $0 \leq \gamma < 1$ is the discount factor. Let $Q^\pi(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^\pi(s')$ denote the expected infinite discounted sum of reward that the agent will gain if it selects action a at s, and then follows π. Then, we have $V^\pi(s) = \sum_{a \in A} \pi(s,a) Q^\pi(s,a)$; $V^\pi(s)$ and $Q^\pi(s,a)$ are called the state value function and the state-action value function of the policy, respectively.

In a Markov decision process, the objective of an agent is to find an *optimal policy*, π^*, which maximizes the state value function for all states (i.e., $\forall \pi$, $\forall s \in S$, $V^{\pi^*}(s) \geq V^\pi(s)$). Every MDP has a deterministic stationary optimal policy; and the following Bellman equation holds (Bellman 1957) $\forall s \in S$:

$$V^*(s) = \max_{a \in A} \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right) = \max_{a \in A} Q^*(s,a). \tag{1}$$

Here, V^* and Q^* are called the optimal value functions. Using Q^*, it is possible to specify π^* by choosing action a at state s which maximizes $Q^*(s,a)$. Note that, the optimal policy is not unique because there may be multiple actions that maximize the value of Q^* at state s.

When the reward function R and the state transition function T are known, π^* can be found by using *dynamic programming* techniques (Littman et al. 1996; Sutton and Barto 1998). When such information is not readily available Monte Carlo or temporal-difference (TD) learning methods are used. Instead of requiring the complete knowledge of the underlying model, these approaches rely on experience in the form of sample state, action, and reward sequences collected from on-line or simulated trial-and-error interactions with the environment.

TD learning methods are built on bootstrapping and sampling principles. Estimate of the optimal state(-action) value function is kept and updated in part on the basis of other estimates. Let $Q(s,a)$ denote the estimated value of $Q^*(s,a)$. Various algorithms basically differ from each other based on how they update the estimation of the optimal value function. In the well-known *Q-learning* algorithm (Watkins and Dayan 1992), given the observation tuple (s,a,r,s') such that s' is the state observed by the agent after taking action a at state s and receiving reward r, $Q(s,a)$ is updated according to the learning rule

$$Q(s,a) = (1-\alpha) Q(s,a) + \alpha \left[r + \gamma \max_{a' \in A} Q(s',a') \right] \tag{2}$$

where $\alpha \in [0,1)$ is the learning rate. In simple TD learning algorithms, the update of the estimation for the current state (-action pair) depends only on the immediate reward and the value of the next state is used as an approximation for the remaining rewards. Instead of a single step backup, n-step TD and TD(λ) algorithms, such as SARSA(λ), consider a fixed-length sequence of rewards over a trajectory or the discounted average of all such reward sequences in the estimation; at each step, the change in the state(-action) value of the current state is gradually reflected backwards to the previously visited states (state-action pairs) in the trajectory.

4 Semi-Markov decision processes and options

Markov decision processes introduced in the previous section, and consequently algorithms based on the MDP framework are restricted in the sense that all actions are presumed to take unit time duration. Therefore, it is not possible to model situations in which actions

take variable amount of time, i.e., they are temporally extended. *Semi-Markov Decision Processes* (SMDP) extend MDPs to incorporate transitions with stochastic time duration.

A discrete-time SMDP is a tuple $\langle S, A, T, R\rangle$, where S is a finite set of states, A is a finite set of actions, $T : S \times A \times S \times \mathbb{N} \to [0, 1]$ is a state transition function such that $\forall a \in A, \sum_{s',n} T(s, a, s', n) = 1$, $R : S \times A \to \Re$ is a reward function; $T(s, a, s', n)$ denotes the probability of making a transition from state s to state s' by taking action a in n time steps; $R(s, a)$ is the *expected* reward that will be received until the next transition, i.e., when action a is executed in state s. During a transition from one state to another upon executing an action, the state of the environment may change continually, but the agent has no direct effect on the course of events until the current action terminates. Similar to MDPs, a (stationary) policy for an SMDP is a mapping from states to actions, and the corresponding Bellman equations hold for an optimal policy (Parr 1998). Reinforcement learning algorithms for MDPs can be generalized or adapted to SMDPs by taking into account the length of transitions (Bradtke and Duff 1994; Mahadevan et al. 1997; Parr 1998). For example, the update rule of Q-learning becomes:

$$Q(s, a) = (1 - \alpha) Q(s, a) + \alpha \left[\hat{r} + \gamma^t \max_{a' \in A} Q(s', a')\right] \tag{3}$$

where a is the action selected at state s, s' is the observed state after a terminates, $\hat{r}$ is the appropriately weighted sum of rewards received during the transition from state s to s', and t is the time passed in between.

In SMDP formalism, the actions are treated as "black boxes", indivisible flow of execution, which are used as they are irrespective of their underlying internals. Nevertheless, this assumption makes it hard to analyze and modify temporally extended actions if their behavior is not determined beforehand, such as when they need to adapt to changes in the environment or be learned from simpler actions. By embedding a discrete-time SMDP over a MDP, the *options* framework of Sutton et al. (1999) extends the theory of reinforcement learning to include temporally extended actions with an explicit interpretation in terms of the underlying MDP. While keeping the unit time transition dynamics of MDPs, actions are generalized in the sense that they may last for a number of discrete time steps and referred to as *options*.

An *option* is a tuple $\langle I, \pi, \beta\rangle$, where $I \subseteq S$ is the set of states that the option can be initiated at, called *initiation set*; π is the option's local policy; and β is a probability distribution induced by the termination condition. Once an option is initiated by an agent at a state $s \in I$, π is followed and actions are selected according to π until the option terminates (stochastically) at a specific condition determined by β. It is possible to alter the behavior of an option by changing I, β, or π, which is simply a restricted policy over actions. In a *Markov option*, action selection and option termination decisions are made solely on the basis of the current state, i.e., $\pi : S \times A \to [0, 1]$, and $\beta : S \to [0, 1]$. During option execution, if the environment makes a transition to state s, then the Markov option terminates with probability $\beta(s)$ or else continues, determining the next action a with probability $\pi(s, a)$. It is generally assumed that an option can also be initiated at a state where it can continue, which means that the set of states with termination probability less than one is a subset of I.

One drawback of Markov options is that they are limited in the sense that their local policies and termination probability distributions depend only on the current state. For more flexibility, *Semi-Markov options* extend Markov options and allow π and/or β to depend on all prior events that occurred since the option was initiated.

Let $s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, \ldots, r_\tau, s_\tau$ be the sequence of states, actions and rewards observed by the agent starting from time t until time τ. This sequence is called a *history* from

t to τ, denoted $h_{t\tau}$ (Sutton et al. 1999). The length of $h_{t\tau}$ is $\tau - t$. If the set of all possible histories is denoted by Ω, then in a Semi-Markov option, β and π are defined over Ω instead of S, i.e., $\beta : \Omega \to [0, 1]$, $\pi : \Omega \times A \to [0, 1]$. Let O be the set of available options, which also includes primitive actions as special cases (options of unit duration), then a (stationary) policy over options $\mu : S \times O \to [0, 1]$ is a mapping that defines the probability of selecting an option from a particular state. If state s is not in the initiation set of an option o, then $\mu(s, o)$ is zero. Options defined in this way induce an SMDP where each action of the SMDP is an option (Sutton et al. 1999). Hence, the results given above for SMDPs also hold for options, and optimal value functions and Bellman equations can be generalized to options and to policies over options; SMDP learning methods can be employed by replacing the action set by the option set.

5 Options in the form of conditionally terminating sequences

In this section, we present the theoretical foundations and building blocks of our automatic option discovery method. In Sect. 5.1, we describe a special case of Semi-Markov options in the form of *conditionally terminating sequences* as defined in McGovern (2002). In Sect. 5.2, we highlight their limitations and propose the novel concept of *sequence trees* (and the corresponding formalism of *stochastic conditionally terminating sequences*) that extend conditionally terminating sequences and enable richer abstractions; they are capable of representing and generalizing the behavior of a given set of conditionally terminating sequences. Finally, we introduce a method which utilizes a modified version of a sequences tree to find useful abstractions during the course of learning; it is based on the idea of reinforcing the execution of action sequences that are experienced frequently by the agent and yield a high return.

5.1 Conditionally terminating sequences

Definition 5.1 (Conditionally terminating sequence) A *conditionally terminating sequence* (CTS) is a sequence of n ordered pairs $\sigma = \langle C_1, a_1\rangle\langle C_2, a_2\rangle \ldots \langle C_n, a_n\rangle$; each ordered pair $\langle C_i, a_i\rangle$ consists of a continuation set $C_i \subseteq S$ and action $a_i \in A$. At step i, a_i is selected and executed; as a result the sequence advances to the next step if current state s is in C_i; otherwise the sequence terminates.

C_1 is the initiation set of σ; it is denoted $init_\sigma$. The sequence $act\text{-}seq_\sigma = a_1 a_2 \ldots a_n$ is called the *action sequence* of σ. We use $C_{\sigma,i}$ and $a_{\sigma,i}$ to denote the ith continuation set and action of σ. For every conditionally terminating sequence σ, one can define a corresponding Semi-Markov option o_σ. Details of the construction are given in Lemma A.1 in the Appendix.

The most important feature of conditionally terminating sequences is that they can be used to represent frequently occurring and useful patterns of actions in a reinforcement learning problem. For example, consider the 5×5 grid world shown in Fig. 1; starting from any location, the agent's goal is to reach with minimum number of actions (i.e., as quickly as possible) the top rightmost cell marked with "G". At each time step, the agent can move in one of four directions; assume $A = \{n, s, e, w\}$ is the set of actions and $S = \{(i, j)|0 \leq i, j \leq 4\}$ is the set of states, where (i, j) denotes the coordinates of the agent. We will represent the rectangular region on the grid with corners at (r_1, c_1) and (r_2, c_2) by $[(r_1, c_1), (r_2, c_2)]$. Note that each such region is a subset of S. In order to reach the goal cell, one of the useful

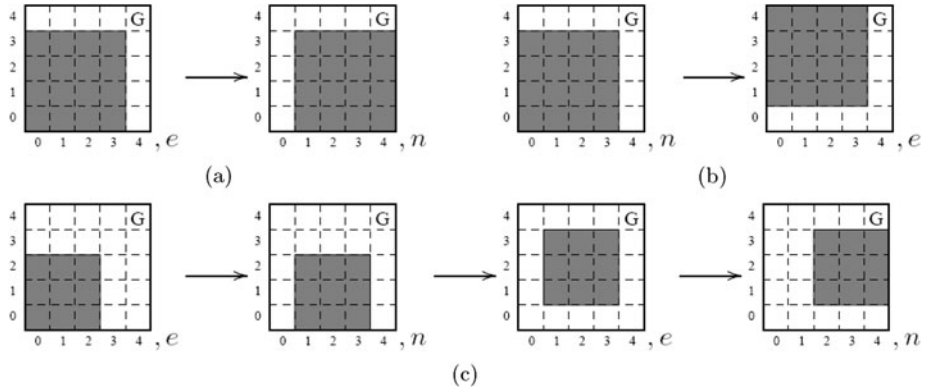


Fig. 1 Sample conditionally terminating sequences for moving diagonally in the north-east direction. (**a**) σ_{en} (east followed by north), (**b**) σ_{ne} (north followed by east), and (**c**) σ_{enen} (two consecutive east-north moves). *Shaded areas* denote the continuation sets

action patterns that can be used by the agent is to move diagonally in the north-east direction, i.e., e followed by n or alternatively n followed by e. These patterns can be represented by the following conditionally terminating sequences presented in Fig. 1(a) and (b):

$$\sigma_{en} = \langle [(0,0),(3,3)], e\rangle \langle [(0,1),(3,4)], n\rangle$$
$$\sigma_{ne} = \langle [(0,0),(3,3)], n\rangle \langle [(1,0),(4,3)], e\rangle$$

Conditionally terminating sequences allow an agent to reach the goal more directly by shortening the path to a solution; in our grid world example, any primitive action can reduce the Manhattan distance to the goal position (i.e., $|4-i|+|4-j|$ where (i, j) is the current position of the agent) by 1 at best, whereas when applicable σ_{en} and σ_{ne} defined above reduce the goal position by 2. As the complexity of the problem increases, the shortening through the use of conditionally terminating sequences makes it possible to efficiently explore the search space to a larger extent. Consequently, this leads to faster convergence and improves the performance of learning. Although each conditionally terminating sequence has a simple structure, a set of conditionally terminating sequences can be quite effective in exploiting temporal abstractions.

Now, consider the longer conditionally terminating sequence σ_{enen} given in Fig. 1(c), which represents moving diagonally in the north-east direction two times; it is defined as:

$$\sigma_{enen} = \langle [(0,0),(2,2)], e\rangle \langle [(0,1),(2,3)], n\rangle \langle [(1,1),(3,3)], e\rangle \langle [(1,2),(3,4)], n\rangle$$

Note that the action sequence of σ_{enen} starts with the action sequence of σ_{en}; for the first two steps, the actions imposed by σ_{en} and σ_{enen} are the same. Therefore, by taking the union of the continuation sets, it is possible to merge σ_{en} and σ_{enen} into a new conditionally terminating sequence $\sigma_{en-enen}$ which exhibits the behavior of both sequences:

$$\begin{aligned}\sigma_{en-enen} &= \langle C_{\sigma_{en},1} \cup C_{\sigma_{enen},1}, e\rangle \langle C_{\sigma_{en},2} \cup C_{\sigma_{enen},2}, n\rangle \langle C_{\sigma_{enen},3}, e\rangle \langle C_{\sigma_{enen},4}, n\rangle \\ &= \langle C_{\sigma_{en},1}, e\rangle \langle C_{\sigma_{en},2}, n\rangle \langle C_{\sigma_{enen},3}, e\rangle \langle C_{\sigma_{enen},4}, n\rangle\end{aligned}$$

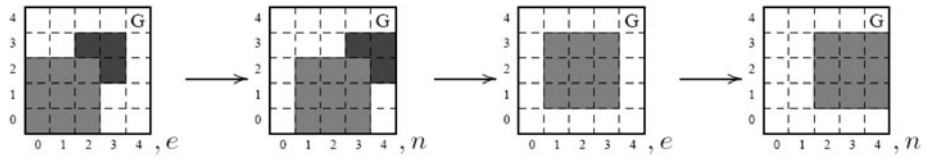


Fig. 2 The union of the conditionally terminating sequences β_{en} and σ_{enen}. *Dark shaded areas* show the continuation sets of β_{en} and *light shaded areas* show the continuation sets of σ_{enen}

Similar to σ_{enen}, the action sequence of $\sigma_{en-enen}$ is also *enen*; initially it behaves as if it is both σ_{en} and σ_{enen}, i.e., selects action e and then n at viable states. At the third step, if the current state allows σ_{enen} to continue (i.e., in $C_{\sigma_{enen},3}$), then the sequence continues execution as if it is σ_{enen}; otherwise it terminates. We call $\sigma_{en-enen}$ the *union* of σ_{en} and σ_{enen}.

Definition 5.2 (Union of two CTSs) Let $u = \langle C_{u,1}, a_{u,1}\rangle \ldots \langle C_{u,m}, a_{u,m}\rangle$ and $v = \langle C_{v,1}, a_{v,1}\rangle \ldots \langle C_{v,n}, a_{v,n}\rangle$ be two conditionally terminating sequences, such that the action sequence of v starts with the action sequence of u, i.e., $a_{u,i} = a_{v,i}$ for $1 \leq i \leq m$ and $m \leq n$. The conditionally terminating sequence $u \cup v$, called the union of u and v, is defined as:

$$u \cup v = \langle C_{u,1} \cup C_{v,1}, a_{v,1}\rangle\langle C_{v,2} \cup C_{v,2}, a_{v,2}\rangle \ldots \langle C_{u,m} \cup C_{v,m}, a_{v,m}\rangle \langle C_{v,m+1}, a_{v,m+1}\rangle \ldots \langle C_{v,n}, a_{v,n}\rangle$$

It is worth noting that given a sequence of observed states, there may be cases in which both u and v would terminate within $|u| = m$ steps, but $u \cup v$ continue to execute. For example, let $\beta_{en} = \langle[(2,2)(3,3)], e\rangle\langle[(2,3)(3,4), n\rangle$ be a restricted version of diagonal movement in the north-east direction. We have $\beta_{en} \cup \sigma_{enen} = \langle[(0,0)(2,2)] \cup [(2,2)(3,3)], e\rangle\langle[(0,1)(2,3)] \cup [(2,3)(3,4), n\rangle\langle[(1,1),(3,3)], e\rangle\langle[(1,2),(3,4)], n\rangle$ (Fig. 2). Initiated at state $(3,2)$, which is in the continuation set of the first tuple of β_{en} but not that of σ_{enen}, $\beta_{en} \cup \sigma_{enen}$ would start to behave like β_{en} and select action e. Suppose that, due to the non-determinism in the environment, this action moves the agent to $(2,2)$ instead of $(3,3)$. By definition, β_{en} can not continue to execute from $(2,2)$ since $(2,2) \notin C_{\beta_{en},2}$; however $(2,2)$ is in the continuation set of the second tuple of σ_{enen}, and therefore $\beta_{en} \cup \sigma_{enen}$ resumes execution from $(2,2)$, in essence switching to σ_{enen}. Thus, the union of two conditionally terminating sequences also generalizes their behavior in favor of longer execution patterns whenever possible, resulting in a more effective abstraction.

5.2 Extending conditionally terminating sequences

One prominent feature of conditionally terminating sequences is that they have a linear flow of execution; i.e., actions are selected sequentially as specified by the tuples provided that the corresponding continuation conditions hold. In this respect, they cannot be used to represent situations in which different courses of actions may be followed depending on the observed history of events. On the other hand, such situations are frequent in most real life problems due to the hierarchical decomposition inherent in their structure; abstractions contain common action sequences that solve various similar sub-tasks involved. When we want to use conditionally terminating sequences in such problems, a separate conditionally terminating sequence is required for each trajectory of a component that corresponds to a particular sub-task in the hierarchy. As the complexity of the problem increases, this situation leads to a drastic increase in the number of conditionally terminating sequences that

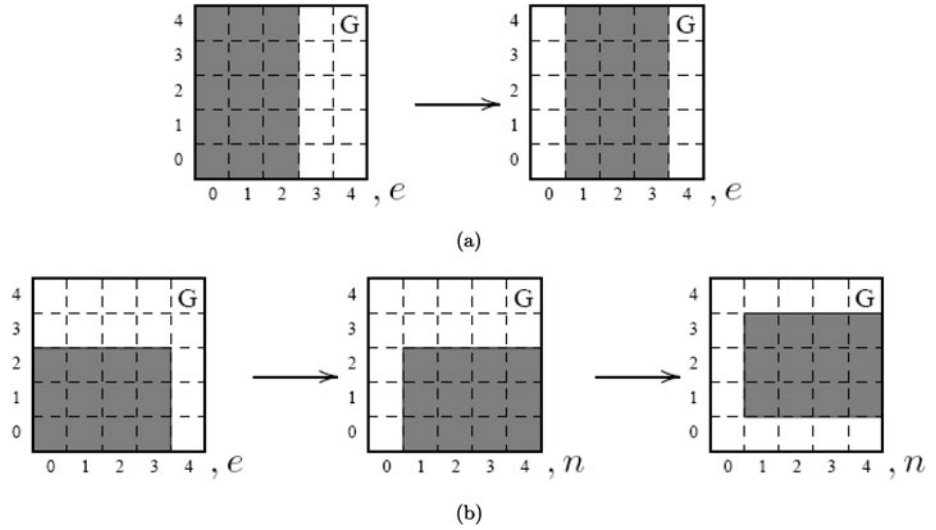


Fig. 3 The conditionally terminating sequences (**a**) σ_{ee} (east followed by north), and (**b**) σ_{enn} (east followed by two norths). *Shaded areas* denote the continuation sets

need to be defined, which constitutes one of the major drawbacks of conditionally terminating sequences. By extending conditionally terminating sequences to incorporate conditional branching in action selection, it is possible to make use of available abstractions in a more compact and effective way, and therefore overcome this shortcoming.

As a demonstrative example, consider the conditionally terminating sequence σ_{enen} defined in the previous section and the two new conditionally terminating sequences presented in Fig. 3, which are defined as:

$$\sigma_{ee} = \langle[(0,0),(4,2)],e\rangle\langle[(0,1),(4,3)],e\rangle$$
$$\sigma_{enn} = \langle[(0,0),(2,3)],e\rangle\langle[(0,1),(2,4)],n\rangle\langle[(1,1),(3,4)],n\rangle$$

σ_{ee} has an action pattern of moving east twice, and σ_{enn} has an action pattern of moving east followed by moving north twice. Note that, the action sequences of these three conditionally terminating sequences have common prefixes. They all select action e at the first step, and furthermore both σ_{enn} and σ_{enen} select action n at the second step. Suppose that the conditionally terminating sequence to be initiated at state s is chosen based on a probability distribution $P : S \times \{\sigma_{ee}, \sigma_{enn}, \sigma_{enen}\} \to [0,1]$; let $viable_i = \{\sigma \in \{\sigma_{ee}, \sigma_{enn}, \sigma_{enen}\} | \exists I_{\sigma,i}, s_i \in I_{\sigma,i}\}$ denote the set of conditionally terminating sequences which are compatible with the state s_i observed by the agent at step i, and σ_i be the conditionally terminating sequence chosen by P over $viable_i$. Then, by taking the union of common parts and directing the flow of execution based on $viable_i$, it is possible to combine the behavior of these conditionally terminating sequences as follows:

1. If $viable_1 = \emptyset$ then terminate. Otherwise, execute action e.
2. If $viable_2 = \emptyset$ then terminate. Otherwise,
   (a) If $\sigma_2 = \sigma_{ee}$ then execute action e.
   (b) Otherwise, i.e., if $\sigma_2 \in \{\sigma_{enn}, \sigma_{enen}\}$, execute action n.
      i. If $viable_3 = \emptyset$ then terminate. Otherwise,

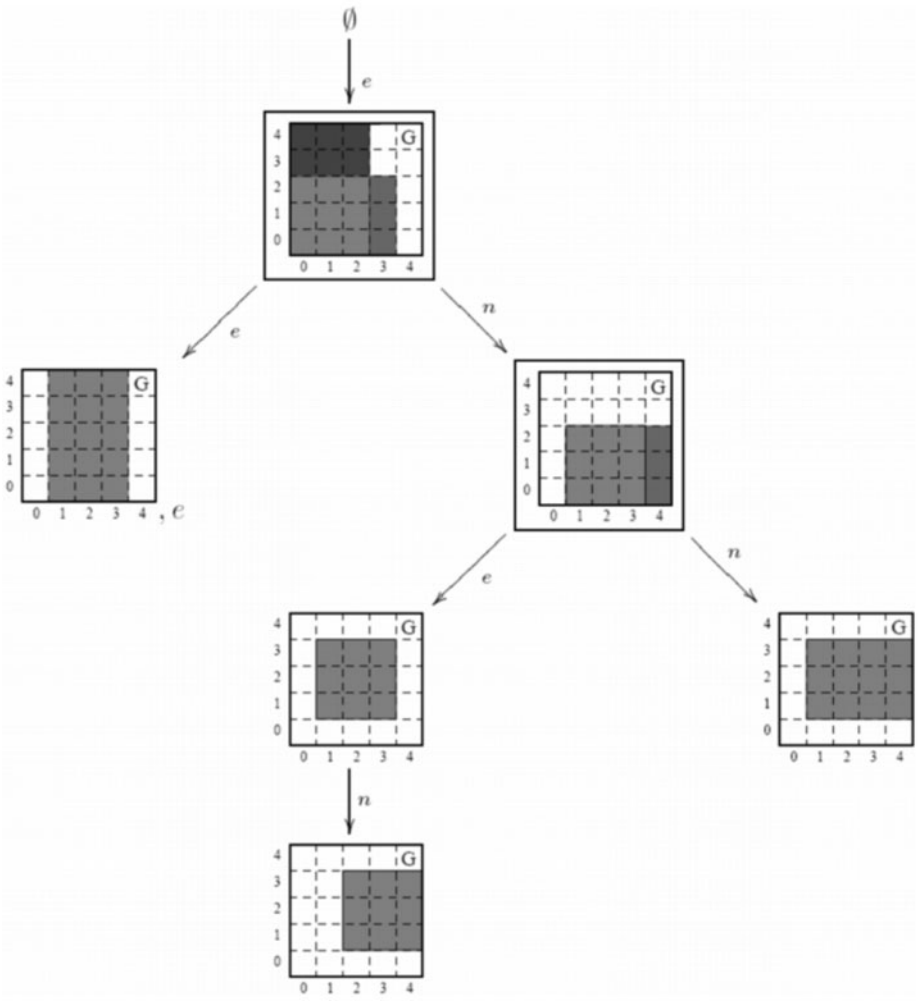


Fig. 4 Combination of the three conditionally terminating sequences σ_{ee}, σ_{enn} and σ_{enen}. *Shaded areas* show the set of states where the corresponding action (denoted by the label of the incoming edge, such as going *e*ast or *n*orth) can be chosen; the framed grids indicate the decision points

A. If $\sigma_3 = \sigma_{enn}$ then execute action n.
B. Else, i.e., if $\sigma_3 = \sigma_{enen}$, execute action e followed by the fourth tuple of $\sigma_{enen} = \langle I_{\sigma_{enen},4}, n \rangle$.

Steps 2 and 2(b)i essentially introduce conditional branching to action selection; they are called *decision points*. The entire process can be encapsulated and represented in a tree form as depicted in Fig. 4; ∅ represents the root of the tree and decision points are enclosed in a rectangle. At each node, shaded areas on the grid denote the states for which the corresponding action (label of the incoming edge) can be chosen. They are comprised of the union of continuation sets of compatible conditionally terminating sequences. We call such a tree a *sequence tree*.

Algorithm 1 Algorithm to construct a sequence tree from a given set of conditionally terminating sequences.

```
1: function CONSTRUCT(Σ)                    ▷ Σ is a set of conditionally terminating sequences.
2:     N = {∅}                              ▷ N is the set of nodes. Initially it contains the root node.
3:     E = {}                               ▷ E is the set of edges.
4:     for all σ ∈ Σ do                     ▷ For each conditionally terminating sequence σ in Σ
5:         current = ∅                      ▷ Start from the root node.
6:         for i = 1 to |σ| do              ▷ For each tuple of σ
7:             if ∃⟨current, p, a_{σ,i}⟩ ∈ E then   ▷ Is the current node already connected to a
   node p by an edge with label a_{σ,i}?
8:                 cont_p = cont_p ∪ I_{σ,i}        ▷ Combine the continuation sets.
9:             else
10:                Create a new node p with cont_p = {I_{σ,i}}
11:                N = N ∪ {p}              ▷ Add the new node p to the existing tree.
12:                E = E ∪ {⟨current, p, a_{σ,i}⟩}   ▷ Connect the current node to p by an edge
   with label a_{σ,i}.
13:            end if
14:            current = p                  ▷ Node p becomes the current node.
15:        end for
16:    end for
17:    return ⟨N, E⟩                        ▷ Return the constructed tree.
18: end function
```

Definition 5.1 (Sequence tree) A sequence tree is a tuple $\langle N, E\rangle$ where N is the set of nodes and E is the set of edges. Each node represents a unique action sequence; the root node, denoted by $\emptyset$, represents the empty action set. If the action sequence of node q can be obtained by appending action a to the action sequence represented by node p, then p is connected to q by an edge with label a; it is denoted by the tuple $\langle p, q, a\rangle$. Furthermore, q is associated with a continuation set $cont_q$ specifying the states where action a can be chosen after the execution of action sequence p. A node p with $k > 1$ out-going edges is called *decision point* of order k.

Generalizing the example given above, given a set of conditionally terminating sequences $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$, a corresponding sequence tree T_Σ that captures their behavior in a compact form can be constructed using the CONSTRUCT procedure presented in Algorithm 1. The algorithm initially creates a sequence tree comprised of the root node only. Then, the conditionally terminating sequences in Σ are one by one added to the existing tree starting from the root node and following edges according to their action sequence; new nodes are created as necessary; and the continuation sets of the nodes are updated by uniting them with the continuation sets of the conditionally terminating sequences. The number of nodes in T_Σ is equal to the total number of unique action sequence prefixes of conditionally terminating sequences in Σ. The space requirement is dominated by continuation sets of nodes, which is bounded by the total space required for the storing the continuation sets. A sequence tree is the representational form of a novel type of abstraction that we named *stochastic conditionally terminating sequence* (S-CTS) (Girgin et al. 2006a, 2006b, 2007). In Appendix, the formal definition of stochastic conditionally terminating sequences is given and the mapping between S-CTSs and sequence trees is discussed extensively.

Initiated at state s, a sequence tree T can be used to select actions thereafter by starting from the root node and following edges according to the continuation sets of the outgoing

nodes. Initially, the active node of T is the root node. At each time step, if the active node has children which contain in their continuation sets the current state observed by the agent, then:

1. one of them is chosen using a probability distribution defined over the set of conditionally terminating sequences that T is constructed from (in the simplest setting, chosen uniformly),
2. the action specified by the label of the edge connecting the active node to the chosen node is executed,
3. the active node is set to the chosen node.

Otherwise, the action selection procedure terminates. In a reinforcement learning problem, when the set of conditionally terminating sequences Σ is known in advance, one can construct a corresponding sequence tree and by employing the process described above utilize the tree instead of the sequences. However, determining a set of useful conditionally terminating sequences is a complex process and it requires extensive domain knowledge; such an information may not be always available prior to learning. Discovering useful conditionally terminating sequences on-the-fly and integrating them as the learning progresses is an alternative approach which is certainly more interesting for machine learning. Learning macro-actions in the form of conditionally terminating sequences has been previously studied by McGovern and an algorithm named *acQuire-macros* has been proposed (McGovern 1998). In acQuire-macros, all state-action trajectories experienced by the agent are stored and a list of eligible sequences is kept. Periodically, such as at the end of each episode,

1. using the stored trajectories, frequent action sequences having a support over a given threshold are identified and added to the list of eligible sequences by applying a process that makes use of successive doubling starting from sequences of length 1 (i.e., the primitive actions),
2. the eligibility values of the identified sequences are incremented; and a new option is created for an action sequence if the eligibility value of that particular sequence is over a given threshold *and* the sequence passes a problem-specific static filter, and finally
3. the eligibility values of all eligible sequences are decayed.

Although it is shown empirically to be quite effective, this approach has several drawbacks: (i) it requires storing all state-action trajectories since the beginning of the learning; (ii) identification of frequent sequences which is repeated at each step is a costly operation since it requires processing of all state-action trajectories that have been stored so far; and (iii) a separate option is created for each frequent sequence which necessitates problem-specific static filtering to prevent options that are "similar" to each other. In order to overcome these shortcomings, we propose a novel approach which utilizes a single abstraction that is modified continuously and the agent executes it as an exploration policy, but does not maintain a value for it in the traditional sense. This single option is a combination of many action sequences and is represented as a modified version of a sequence tree. During execution, the choice among different branches is made using an estimate of the expected return obtained by following each branch. The tree is periodically updated to incorporate useful abstractions by using the observed state-action trajectories. We start our discussion with the determination of valuable sequences.

Definition 5.3 (π-history) A history is called a π-history of state s if it starts with state s and it is obtained by following a policy π until the end of an episode (or between designated conditions, such as reaching a reward peak).

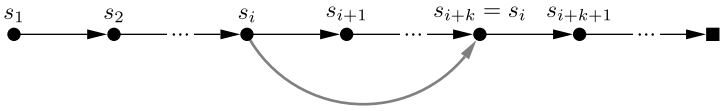


Fig. 5 Two history alternatives for state s_1 where s_t denotes that state observed at time t and square node is a final state. The first one is the entire sequence that follows the horizontal arrows; the second one is obtained by unifying the second instances of state s_i at s_{i+k} with the first one, i.e. by following the lower arrow and omitting all transitions in between

Let π and π^* denote the agent's current policy and an optimal policy, respectively, and $h = s_1 a_1 r_2 \ldots r_t s_t$ be a π-history of length t for state s_1. The total cumulative reward of h is defined as $R(h) = r_2 + \gamma r_3 + \cdots + \gamma^{t-2} r_t$, which reflects the discounted accumulated reward obtained by the agent upon following action choices and state transitions in h. Now, suppose that in h a state appears at two positions i and $i+k$, i.e., $s_i = s_{i+k}$, $k > 0$; and consider the sequence $h' = s_1 a_1 r_2 \ldots r_i s_i a_{i+k+1} r_{i+k+1} \ldots r_t s_t$, where s_i and s_{i+k} are collapsed and the sequence in between is removed (see Fig. 5); we can have the following observations:

Observation 5.1 *h' is also a (synthetic) π-history for state s_1 and could be a better candidate as a π^*-history if $R(h') > R(h)$.*

Observation 5.2 *Every suffix of h of the form $h_i = s_i a_i r_{i+1} \ldots r_t s_t$ for $i = 2, \ldots, t-1$ is a π-history of s_i.*

Combining these two observations, we can generate a set of potential π^*-history candidates by processing h from the end of the history to the front. Let $best(s)$ denote the π-history for state s with maximum total cumulative reward; initially $best(s_{t-1}) = s_{t-1} a_{t-1} r_t s_t$. For each $s_i, i = t-2, \ldots, 1$, if s_i is not encountered before (i.e., for all $j > i, s_j \neq s_i$) or $r_i + \gamma R(best(s_{i+1}))$ is higher than the total cumulative reward of the current $best(s_i)$, i.e., $R(best(s_i))$, then $best(s_i)$ is replaced by $s_i a_i r_{i+1} \circ best(s_{i+1})$, where $\circ$ is the concatenation operator and appends the history represented by $best(s_{i+1})$ to $s_i a_i r_{i+1}$. Finally, for each unique s_i in $(s_1, \ldots, s_t)$, the resulting $best(s_i)$ is used as a probable π^*-history for state s_i. The complete procedure is given in Algorithm 2. Here it is worth noting that the

Algorithm 2 Algorithm to generate probable π^*-histories from a given history h.

function GENERATE-PROBABLE-HISTORIES(h)
h is a history events of the form $s_1 a_1 r_2 \ldots r_t s_t$
  $best[s_{t-1}] = s_{t-1} a_{t-1} r_t s_t$ ▷ $best[s]$ holds the current π^*-history candidate for state s.
  $R[s_{t-1}] = r_t$ ▷ $R[s]$ holds the total cumulative reward for state s.
  for $i = t-2$ down to 1 **do** ▷ Process from the end of the history to the front
    if $R[s_i]$ is not set or $r_{i+1} + \gamma R[s_{i+1}] > R[s_i]$ **then** ▷ Is s_i not encountered before or does it have a lower return estimate?
      $best[s_i] = s_i a_i r_{i+1} \circ best[s_{i+1}]$ ▷ Create or update the candidate π^*-history and total cumulative reward corresponding to the state s_i.
      $R[s_i] = R_{i+1} + \gamma R[s_{i+1}]$
    end if
  end for
  return *best* ▷ Return the π^*-history candidates.
end function

concatenation operation can be implemented efficiently without actually merging the history segments by using pointers. As the learning process progresses, probable π^*-histories with action sequences that are part of useful conditionally terminating sequences (i.e., sub-policies) would appear more frequently in the recent episodes, whereas the occurrence rate would be low for histories whose action sequences belong to conditionally terminating sequences with dominated sub-policies or that are less general. Therefore, by keeping track of the generated histories and generalizing them in terms of the continuation sets, it is possible to identify valuable abstractions and utilize them to improve the learning performance. For efficiency and scalability, this must be accomplished without storing and processing all observed state-action trajectories. Also, note that existing abstractions are not expected to change drastically between successive iterations. Therefore, instead of explicitly creating the conditionally terminating sequences first and then constructing the corresponding sequence tree at each iteration, it would be more preferable and practical to build the tree directly in an incremental manner. For this purpose, we propose to extend the structure of sequence trees.

Definition 5.4 (Extended sequence tree) An *extended sequence tree* is a tuple $\langle N, E\rangle$, where N is the set of nodes and E is the set of edges. Each node represents a unique action sequence that is used to reach that node; the root node, denoted by $\emptyset$, represents the empty action set. If the action sequence of node q can be obtained by appending action a to the action sequence represented by node p, then p is connected to q by an edge with label $\langle a, \psi\rangle$; it is denoted by the tuple $\langle p, q, \langle a, \psi\rangle\rangle$. ψ is the eligibility value of the edge to indicate how frequently the action sequence of q is executed. Furthermore, q holds a list of tuples $\langle s_1, \xi_{s_1}, R_{s_1}\rangle, \ldots, \langle s_k, \xi_{s_k}, R_{s_k}\rangle$ stating that action a can be chosen at node p if current state observed by the agent is in $\{s_1, \ldots, s_k\}$, which is called the continuation set of node q, denoted $cont_q$. R_{s_i} is the expected total cumulative reward that the agent can collect by selecting action a at state s_i after having executed the sequence of actions represented by node p. ξ_{s_i} is the eligibility value of state s_i at node q and indicates how frequently action a is actually selected at state s_i.

An extended sequence tree is basically an adaptation of a sequence tree that contains additional eligibility and reward attributes to keep statistics about the represented abstractions; the additional attributes allow discrimination of frequent sequences with high expected reward.

A π-history, $h = s_1 a_1 r_2 \ldots r_t s_t$, can be added to an extended sequence tree T by invoking Algorithm 3. Similar to the CONSTRUCT procedure presented in Algorithm 1, ADD-HISTORY starts from the root node of the tree and follows edges according to the action sequence of the history. Initially, the active node of T is the root node. At step i, if the active node has a child node n connected to the active node by an edge with label $\langle a_i, \psi\rangle$ then

1. ψ is incremented to reinforce the eligibility value of the edge,
2. if node n contains a tuple $\langle s_i, \xi_{s_i}, R_{s_i}\rangle$ then ξ_{s_i} is incremented to reinforce the eligibility value of state s_i, and R_{s_i} is updated with a certain rate α using $R_i = r_{i+1} + \gamma r_{i+2} + \cdots + \gamma^{t-i-1} r_t$; R_i denotes the discounted cumulative reward obtained following h starting from step i. Otherwise, a new tuple $\langle s_i, 1, R_i\rangle$ with an eligibility value of 1 is added to node n.

If the active node does not have such a child node, then a new node n that contains the tuple $\langle s_i, 1, R_i\rangle$ is created and connected to the active node by an edge with label $\langle a_i, 1\rangle$; the eligibility values are initially set to 1. In both cases, n becomes the active node. When the

Algorithm 3 Algorithm for adding a π-history to an extended sequence tree.

procedure ADD-HISTORY(h, T)
h is a π-history of the form $s_1 a_1 r_2 \ldots r_t s_t$, and T is an existing extended sequence tree.
  $R[t] = 0$ ▷ For each time step, calculate discounted cumulative rewards obtained by the agent.
  for $i = t - 1$ to 1 **do**
    $R[i] = r_i + \gamma R[i+1]$
  end for
  current =root node of T ▷ The *current* node is initially the root node.
  for $i = 1..t-1$ **do**
    if ∃ a node n such that *current* is connected to n by an edge with label $\langle a_i, \psi \rangle$ **then**
      Increment ψ. ▷ Reinforce the eligibility value of the edge.
      if n contains a tuple $\langle s_i, \xi_{s_i}, R_{s_i} \rangle$ **then**
        Increment ξ_{s_i}. ▷ Reinforce the eligibility value of state s_i at node n.
        $R_{s_i} = R_{s_i} + \alpha * (R[i] - R_{s_i})$ ▷ Update the expected discounted cumulated reward.
      else
        Add a new tuple $\langle s_i, 1, R[i] \rangle$ to node n.
      end if
    else
      Create a new node n containing the tuple $\langle s_i, 1, R[i] \rangle$.
      Connect *current* node to n by an edge with label $\langle a_i, 1 \rangle$.
    end if
    current $= n$ ▷ Node n becomes the *current* node.
  end for
end procedure

π-history h is added to the extended sequence tree, only the nodes representing the prefixes of the action sequence of h are modified and associated attributes are updated in support of observing such sequences.

In order to identify and store useful abstractions, first a set of probable π^*-histories are generated using Algorithm 2 based on the sequence of states, actions and rewards observed by the agent during a specific period of time (such as throughout an episode, or between reward peaks in case of non-episodic tasks) and they are added to the extended sequence tree using Algorithm 3. Then, the eligibility values of edges are decremented by a factor of $0 < \psi_{decay} < 1$, and eligibility values in the tuples of each node are decremented by a factor of $0 < \xi_{decay} \leq 1$. For an action sequence σ that is frequently used, the edges on the path from the root node to the node representing σ and tuples corresponding to the visited states in the nodes over that path would have higher eligibility values since they are incremented each time a π-history with action sequence σ is added to the tree; on the other hand, they would decay to 0 for sequences that are used less often. This has an overall effect of supporting valuable sequences that are encountered frequently. A very small eligibility value of an edge (less than a given threshold $\psi_{threshold}$) indicates that the action sequence represented by the outgoing node is rarely executed by the agent; consequently, the edge and the subtree below it can be removed from the tree to preserve compactness. Similarly, a very small eligibility value of a tuple $\langle s, \xi, R \rangle$ in a node n (less than a given threshold $\xi_{threshold}$) means that the agent no longer observes state s frequently after executing the action sequence on the path from the root node to node n. Such tuples can also be pruned to reduce the size of the continuation sets of the nodes. After performing these operations, the resulting extended

Algorithm 4 Algorithm for updating extended sequence tree T.

procedure UPDATE-TREE(T, e) ▷ e is the history of events observed by the agent during a specific period of time.
  H = GENERATE-PROBABLE-HISTORIES(e)
  for all $h \in H$ **do** ▷ Add each generated π-history to T.
    ADD-HISTORY(h, T)
  end for
  UPDATE-NODE(root node of T) ▷ Traverse and update the tree.
end procedure

procedure UPDATE-NODE(n)
  Let E be the set of outgoing edges of node n.
  for all $e = \langle n, n', \langle a_{n'}, \psi_{n,n'} \rangle \rangle \in E$ **do** ▷ For each outgoing edge.
    $\psi_{n,n'} = \psi_{n,n'} * \psi_{decay}$ ▷ Decay the eligibility value of the edge.
    if $\psi_{n,n'} < \psi_{threshold}$ **then** ▷ Prune the edge if its eligibility value is below $\psi_{threshold}$.
      Remove e and the subtree rooted at n'.
    else
      UPDATE-NODE(n') ▷ Recursively update the child node n'.
      if tuple list of n' is empty **then** ▷ Prune the edge if its continuation set is empty.
        Remove e and the subtree rooted at n'.
      end if
    end if
  end for
  for all $t = \langle s_i, \xi_{s_i}, R_{s_i} \rangle$ in tuple list of n **do** ▷ For each tuple in n.
    $\xi_{s_i} = \xi_{s_i} * \xi_{decay}$ ▷ Decay the eligibility value of the tuple.
    if $\xi_{s_i} < \xi_{threshold}$ **then** ▷ Prune the tuple if its eligibility value is below $\xi_{threshold}$.
      Remove t from the tuple list of n.
    end if
  end for
end procedure

sequence tree represents recent and useful conditionally terminating sequences in a compact form. The entire process is presented in Algorithm 4.

The action selection mechanism in an extended sequence tree is similar to that of a sequence tree. However, a prior probability distribution to determine the conditional branching is not available as in the case of sequence trees. Therefore, when there are multiple viable actions, i.e., current state observed by the agent is contained in continuation sets of several children of the active node of the tree, the edge to follow is chosen dynamically based on the properties of the children. One important consequence of this situation is that, instead of a single conditionally terminating sequence, the agent in fact starts with a set of them, which initially contains all conditionally terminating sequences represented by the extended sequence tree; it selects a subset of conditionally terminating sequences from this set that are compatible with the observed history of events and concurrently follows them by executing their common action. This enables the agent to broaden the regions of the state space where the conditionally terminating sequences are applicable and results in longer execution patterns than the one that may be attained by employing only a single conditionally terminating sequence (since it covers smaller region of the state space). One possible option for branching is to apply an ϵ-greedy method. With $1 - \epsilon$ probability the agent selects the edge connected to the child node containing the tuple with the highest discounted cumulative re-

ward for the current state, otherwise one of the edges is chosen randomly. This is what we have opted for the experiments presented in the next section.

The extended sequence tree, and the associated mechanism defined above to select which actions to execute based on its structure, is not an option in the traditional sense, but rather a single dynamic *meta-option* that incorporates a set of evolving conditionally terminating sequences. It is possible to employ the extended sequence tree as a single option within the options framework and accordingly calculate the value of executing such option over the state space. However, doing so may lead to suboptimal behavior due to the "forgetting" effect resulting from focusing on frequently occurring action sequences on successful trajectories and pruning the nodes and edges having low eligibility values. This is true because the set of conditionally terminating sequences that were inherently represented by the sequence tree and followed on the last update of the value of a particular state might have evolved or partially invalidated if they are no longer considered as frequently occurring (especially, in the case of low eligibility decay rates). The empirical results which will be presented in Sect. 6 also provide support for this argument. Instead, the extended sequence tree can be integrated into the reinforcement learning framework by transparently augmenting the action selection mechanism of an underlying reinforcement learning algorithm with it. The idea is to extend the action set of the agent by a meta-action that initiates the extended sequence tree in permissible states, i.e., states are defined as the states that exist in the continuation sets of the children of the root node of the extended sequence tree. Once this action is selected, the aforementioned action selection mechanism of the extended sequence determines the primitive actions to be executed by the agent in the successive time steps until it terminates; termination occurs if the active node of the extended sequence tree does not have any child node, or the current state is not in the continuation sets of its children. However, rather than explicitly keeping an estimate of the expected return for this meta-action and updating it based on the cumulative discounted reward received during the time period in which the extended sequence tree was active and the length of this time period, the actions chosen by the extended sequence tree and observations of the agent are directly passed to the underlying reinforcement learning algorithm (such as Q-learning); the underlying reinforcement learning algorithm then updates its value function estimates. Note that, each $\langle s_i, \xi_{s_i}, R_{s_i} \rangle$ tuple stored in a node of the extended sequence tree holds the estimates of the expected total cumulative reward, R_{s_i}, that the agent can collect by selecting the associated action (i.e., the label of the edge connecting the node to its parent node) at state s_i after having executed the sequence of actions represented by that node. These estimates are updated whenever a compatible π-history is added to the extended sequence tree, regardless of whether the action selection mechanism of the extended sequence tree was active or not at that particular point of the source trajectory in which the corresponding state and action sequence have been observed (see Algorithms 3 and 4), and as such reflect more reliable information on the current state of the extended sequence tree and the represented abstractions. At a given state s, the value estimates in the tuples of the children of the root node of the extended sequence tree that contain s in their continuation sets, and the value function estimation of the underlying reinforcement learning algorithm can be used together to decide the next action to be executed (that is, either one of the existing actions or the meta-action that initiates the extended sequence tree). As an example, a modified version of the ϵ-greedy strategy is presented in Algorithm 5; other action selection mechanisms can be derived similarly.

This leads to the learning model given in Algorithm 6 that (given an underlying reinforcement learning algorithm) discovers and utilizes useful temporal abstractions by generating an extended sequence tree and treating it as a meta-action. Let a_{next} denote the next action to be executed by the agent. Initially, the current node of the extended sequence tree is its root node. At each step, the agent performs the following:

Algorithm 5 The ϵ-greedy action selection mechanism modified for the extended sequence tree.

procedure MODIFIED-ϵ-GREEDY(s, T, Q, ϵ) ▷ s is a given state, T is the extended sequence tree and Q is the estimated state-action value function.
  $L = A$ ▷ A is the set of actions.
  for all $a \in A$ **do**
    $value[a] = Q(s, a)$ ▷ $Q(s, a)$ is the estimated value of executing action a at state s.
  end for
  Let $N = \{n_1, \ldots, n_k\}$ be the set of child nodes of the root node of T which contain s in their continuation sets.
  if $N \neq \emptyset$ **then**
    Select n_i from N based on the expected return. ▷ With sufficient exploration.
    Let $\langle a_{n_i}, \cdot \rangle$ be the label of the edge connecting n_i to the root node.
    Let $\langle s, \cdot, R_{s,n_i} \rangle$ be the tuple in n_i corresponding to state s.
    $L = L \cup \{\mathcal{M}_{a_{n_i}}\}$ ▷ $\mathcal{M}$ denotes the meta-action that initiates the extended sequence tree.
    $value[\mathcal{M}] = R_{s,n_i}$ ▷ $\mathcal{M}$ is annotated with action a_{n_i} (see Algorithm 6).
  end if
  Pick a number $p \in [0, 1)$ with uniform probability.
  if $p < \epsilon$ **then**
    Pick *action* from L with uniform probability.
  else
    Pick *action* from L such that $value_{action} = \max_{l \in L} value[l]$.
  end if
  return *action*
end procedure

1. If the meta-action has been initiated, then one of the children of the current node of the extended sequence tree that contains the current state in its continuation set is selected based on the expected return and with sufficient exploration; a_{next} is set to the action specified by the label of the edge connecting the current node to the child node, and the child node becomes the current node. If no such child node exists, then the meta-action terminates and the current node is set to the root node of the extended sequence tree.
2. If the meta-action is not active (or terminated in the previous step), then either one of the actions from the action set or the meta-action (only if it can be initiated at the current state) is selected based on the expected return and the extended sequence tree with sufficient exploration (e.g., using the modified version of the ϵ-greedy strategy). It is assumed that the selection mechanism annotates the meta-action with an action $a_{\mathcal{M}} \in A$ that indicates the actual action to be executed (see Algorithm 5).
   (a) If the meta-action is selected, then the corresponding child node of the root node of the extended sequence tree (i.e., the one which is connected to the root node by an edge with label $\langle a_{\mathcal{M}}, \cdot \rangle$) becomes the current node and a_{next} is set to $a_{\mathcal{M}}$.
   (b) Otherwise, a_{next} is set to the chosen action from the action set.
3. a_{next} is executed; the immediate reward and the next state are observed, and passed to the underlying reinforcement learning algorithm that would process the interaction of the agent with the environment and update its value function estimates accordingly.

Since the underlying reinforcement learning algorithm (in particular updates of the value functions) stays intact, provided that the action selection mechanism allows sufficient exploration (i.e., each state-action pair is visited infinitely often as in the case of the modified

Algorithm 6 Reinforcement learning with extended sequence tree.

T is an extended sequence tree with *root* node only.
A is the set of actions.
$A' = A \cup \{\mathcal{M}\}$ ▷ $\mathcal{M}$ denotes the meta-action.

procedure SELECT-ACTION
  if *active* = *true* **then**
    Let $N = \{n_1, \ldots, n_k\}$ be the set of child nodes of the *current* node which contain s in their continuation sets.
    if $N \neq \emptyset$ **then** ▷ Execution can continue from one of the nodes in N.
      Select n_i from N based on the expected return. ▷ With sufficient exploration.
      Let $\langle a_{n_i}, \cdot \rangle$ be the label of the edge connecting the *current* node to n_i.
      current = n_i ▷ Advance to node n_i.
      return a_{n_i} ▷ Choose action a_{n_i}.
    else
      active = *false* ▷ The meta-option has terminated.
      current = *root* ▷ Reset the *current* node to the root node of T.
    end if
  end if
  Choose $a \in A'$ from s based on the current value function and T. ▷ With sufficient exploration, e.g. using MODIFIED-ϵ-GREEDY procedure; if the meta-action can be initiated at state s and is chosen then it is assumed to be annotated with an action $a_{\mathcal{M}} \in A$ (see Algorithm 5).
  if $a = \mathcal{M}$ **then**
    active = *true*
    Let n be the child node of the *root* node connected by an edge with label $\langle a_{\mathcal{M}}, \cdot \rangle$
    current = n
    return $a_{\mathcal{M}}$
  else
    return a
  end if
end procedure

repeat
  Let *current* denote the active node of T.
  current = *root* ▷ The *current* node is initially set to the root node.
  Let s be the current state.
  $h = s$ ▷ Episode history is initially set to the current state.
  active = *false* ▷ Initially the meta-option is not active.
  repeat ▷ For each time step.
    a_{next} = SELECT-ACTION ▷ Select the next action based on the current value function and T.
    Take action a_{next}, observe r and next state s'
    Update state-action value function using the underlying RL algorithm based on s, r, a_{next}, s'.
    Append r, a_{next}, s' to h. ▷ Update the observed history of events.
    $s = s'$ ▷ Advance to the next state.
  until s is a terminal state
  UPDATE-TREE(T, h) ▷ Update the extended sequence tree by adding the (episode) history h.
until a termination condition holds

ϵ-greedy strategy), the learning model described above preserves many of the theoretical properties, such as convergence to an optimal value function or policy, of the underlying reinforcement learning algorithm. For example, in the Q-learning algorithm, the approximated state-action value function $Q(s,a)$ will converge to the optimal state-value function $Q^*(s,a)$ for discrete MDPs provided that each state-action pair is visited infinitely often and the learning rate has the property of being square summable but not summable; these conditions are still applicable and not invalidated under the proposed model. Although they follow a different heuristic based approach, Bianchi et al. (2008) also employ an analogous model that transparently guides the exploration behavior of an underlying reinforcement learning algorithm for increasing the rate of convergence; we refer interested reader to their work for similar theoretical results under other settings.

6 Experiments

We have applied the method described in Sect. 5.2 to different reinforcement learning algorithms and compared its effect on performance on three benchmark problems: six-room maze problem, various versions of Dietterich's taxi problem (Dietterich 2000), and keep-away sub-task of robotic soccer (Stone et al. 2005). The first problem has bottleneck states, the second problem has repeated sub-tasks, and the last one has a continuous state space and defined in the SMDP setting, i.e., actions take variable amount of time. In this section, we first give the definitions of the studied problems. Then, we present the empirical results that compare the performance of several standard reinforcement learning algorithms with and without the proposed method being applied. We analyze the behavior of the method as the complexity of the problem increases, demonstrate the effects of parameters and also vary the amount of non-determinism in the environment to see the performance of the proposed method under such settings. Furthermore, we discuss how the level of abstraction progresses and how the intermediate abstractions represented by the extended sequence tree perform during the learning process. Finally, we compare the proposed approach with the *acQuire-macros* algorithm of McGovern that inspired our work (McGovern 1998, 2002) on a simple grid world problem for which existing results are available.

In order to determine statistically whether our method or the specific parameters of the learning algorithms affect learning performance and whether the effect of training on performance depends on them or not, we applied the *randomized ANOVA* procedure proposed by Piater et al. for comparing performance curves (Piater et al. 1998). This procedure tests the *algorithm* and the *interaction* effects. The null hypotheses are the following: (a) the mean performances of two or more algorithms (or the same algorithm with different sets of parameters) are the same (no Algorithm effect), and (b) the relation between training and performance does not depend on algorithm (no Interaction effect); these correspond to F tests of a main effect and the interaction effect in a two-way analysis of variance and normally can be calculated using the conventional ANOVA. However, in a given performance (learning) curve the correlation between performance after time steps t_i and t_j (for any i and j) is not constant—points at successive time steps have a tendency of being highly correlated compared to more distant points—which means that the assumption of homogeneity of covariance is violated. In order to handle this problem, Piater et al. repeatedly and randomly redistribute the performance curves to the algorithms; they thus preserve the dependencies of the data points on each curve, and calculate the sample statistics in each shuffling; the distribution of these statistics is then used to find the critical values that must be exceeded in order to reject the null hypotheses with desired level of confidence. In the

experiments described in this paper, we calculated the test results over 1000 shufflings. For more detailed information about this randomized ANOVA procedure, we refer the interested reader to the original paper (Piater et al. 1998).

6.1 Problem set

6.1.1 Six-room maze

In the six-room maze problem (Fig. 6(a)), the agent's task is to move from a randomly chosen position in the top leftmost room of a maze to the grey shaded goal location in the bottom rightmost room; the six rooms of the maze are arranged in a 3×2 configuration and are separated with walls. The primitive actions are movement to a neighboring cell in four directions. Actions are non-deterministic, and each action succeeds with probability 0.9, or moves the agent perpendicular to the desired direction with probability 0.1. The agent receives a reward of $+1$ when it reaches the goal location. For all other cases, it receives a small negative reward of -0.01. If a move action causes the agent to hit a wall the position of the agent does not change. The state space consists of 605 possible positions of the agent. The agent must learn to reach the goal state in shortest possible way to maximize the total discounted reward. As it can be seen from the figure, the agent can pass from one room to another only through the passages connecting the rooms, hence learning how to make use of the passages is crucial to attain an optimal policy quickly. These passages can be considered as the sub-goals of the agent and are also bottleneck states in the solutions.

6.1.2 Taxi domain

Our second domain, Dietterich's Taxi problem, is an episodic task in which a taxi agent moves around on a $n \times n$ grid world, containing obstacles that limit the movement (see Fig. 6(b)). The agent tries to transport one or more passengers located at predefined locations to either the same location or to another one. In order to accomplish this task, the taxi agent must repeat the following sequence of actions for all passengers: (i) go to the location where a passenger is waiting; (ii) pick up the passenger; (iii) go to the destination location; and (iv) drop off the passenger. At each time step, the agent can either move one square in one of four main directions, attempt to *pickup* a passenger, or attempt to *drop-off* the passenger being carried. If a move action causes the agent to hit a wall or an obstacle, the position of the agent does not change. The movement actions are non-deterministic and with a certain

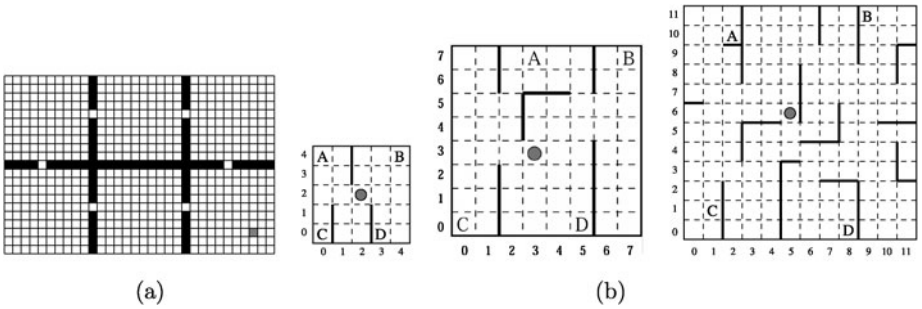


Fig. 6 (**a**) Six-room maze, and (**b**) 5×5, 8×8 and 12×12 taxi problems. In the taxi problems, the four predefined locations are labeled with letters from *A* to *D*

probability, p_{fail}, the agent may move perpendicular (either clockwise or counter-clockwise) in the desired direction. Unless stated otherwise, we fixed p_{fail} to 0.2 in the experiments. Passengers can not be co-located at the same position but their destinations can be the same. An episode ends when all passengers are successfully transported to their destinations. There is an immediate reward of $+20$ for each successful transportation, a high negative reward of -10 if pickup or drop-off actions are executed incorrectly, and -1 for any other action. In order to maximize the overall cumulative reward, the agent must transport the passengers as quickly as possible, i.e., using minimum number of valid actions. Initial position of the taxi agent, locations and destinations of the passengers are selected randomly with uniform probability.

We represent each possible state using a tuple of the form $\langle r, c, l_1, d_1, \ldots, l_k, d_k \rangle$, where r and c denote the row and column of the taxi's position, respectively, k is the number of passengers, and for $1 \leq i \leq k$, l_i denotes the location of the ith passenger (either (i) one of the predefined locations, (ii) picked-up by the taxi, or (iii) transported), and d_i denotes the destination of the passenger (one of the predefined locations). The size of the state space is $RC(L+2)^k L^k$, where $R \times C$ is the size of the grid, L is the number of predefined locations, and k is the number of passengers.[1] Compared to the six-room maze domain, the taxi domain has a larger state space and possesses a hierarchical structure with repeated sub-tasks, such as navigating from one location to another. These sub-tasks are difficult to describe as state based sub-goals because state trajectories are different in each instance of a sub-task. For example, if there is a passenger at location A, the agent must first learn to navigate there, which has the same sub-policy irrespective of the destination of the passenger or other state variables at the time of execution.

6.1.3 Keepaway

Keepaway is a sub-task of simulated robotic soccer. In keepaway, there are two teams of 2–5 players each. Members of the "keepers" team try to keep control of the ball inside a restricted region of a virtual soccer field for as long as possible, whereas the opponent team, "takers", seeks to capture the ball or send it outside of the region (Fig. 7). Typically, the number of takers is (one) less than the number of keepers. The problem is episodic; at the beginning of each episode the keepers are distributed evenly near the corners of the designated region and the ball is placed next to one of them. All takers start at the bottom left corner of the region. Whenever the takers have the possession of the ball or the ball leaves the region, current episode ends and another one begins. Therefore, the aim is to prolong the duration of the episodes. The dynamics of the environment is controlled by

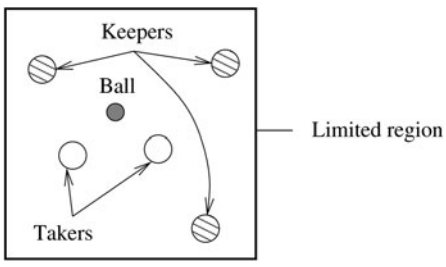


Fig. 7 3×2 keepaway sub-task of robotic soccer

[1] For the single passenger case, since there is only one passenger to be carried, it reduces to $RC(L+1)L$.

the *soccerserver* simulator (Noda et al. 1998). The players have limited and noisy sensors which provide information about various objects in the world (such as relative distance and angle of nearby players and location markers on the field), and the primitive actions, most of them with continuous parameters, are non-deterministic and may fail or succeed partially. In a series of papers, Stone, Sutton and Kuhlmann applied reinforcement learning to learn higher-level decision in keepaway sub-task (Stone and Sutton 2001; Stone et al. 2005). They treat the problem as a *semi-Markov* decision process by using action choices consisting of high level skills instead of primitive actions provided by the simulator, and focus on learning policies for keepers that are within kickable distance to the ball when playing against other keepers and takers with predefined behavior (for example hand-coded or random). Using SMDP version of the SARSA(λ) algorithm with linear tile-coding function approximation and replacing eligibility traces, Stone et al. showed that such players can learn policies that significantly outperform a range of benchmark policies, improving the overall performance of the keepers team. Furthermore, they demonstrated that their approach scales well as the complexity of the task increases.[2]

6.2 Extended sequence tree method applied to standard reinforcement learning algorithms

We first applied the sequence tree method to standard reinforcement learning algorithms on six-room maze and three different versions of the Taxi domain with three different grid sizes, namely 5×5, 8×8 and 12×12. The chosen reinforcement learning algorithms were Q-learning, SARSA(λ) and SMDP Q-learning. While Q-learning is an off-policy algorithm, SARSA(λ) is an on-policy algorithm that uses eligibility traces; in their original form both algorithms consider all actions as primitive actions (i.e., without macro actions). SMDP Q-learning is an extension of Q-learning for learning in SMDPs (Bradtke and Duff 1994) using the update rule given in (3); in SMDP Q-learning, the agent can choose from the primitive or macro actions (either hand-coded or learned). The method is guaranteed to converge when similar conditions as for standard Q-Learning are met (Parr 1998). Q-learning and SARSA(λ) with extended sequence tree are the variants of the learning model described in the previous section, in which the underlying reinforcement algorithm is either Q-learning or SARSA(λ).

By laying a regular grid over the parameter set, we performed a set of systematic initial experiments to determine the optimal values of parameters involved in the learning process. Based on the outcomes of these experiments, the learning rate α is set to 0.125, and the eligibility decay rate λ in the SARSA(λ) algorithm is taken as 0.98 for the six-room maze problem and 0.90 for the taxi problems (see Fig. 8). The p-values for the pairwise comparison of performance curves for different values of λ indicate that in regular SARSA(λ) the difference is significant ($p < 0.001$ except for the case of $\lambda = 0.60$ and 0.70), whereas in SARSA(λ) with sequence tree there is evidence that the hypothesis of no effect of λ cannot be rejected in all cases (Appendix Table 4). The state-action values (Q-values) are initially set to 0. For action selection, the agent followed an ϵ-greedy strategy with $\epsilon = 0.1$. In this strategy, at each state one of the actions having the maximum Q-value is selected with probability $1 - \epsilon$, and a random action is selected uniformly with probability ϵ; as ϵ increases the policy behaves more like a random policy. The effect of ϵ on learning performance is significant (see Fig. 9 and Appendix Table 5). The reward discount factor γ is set to 0.9. For

[2]For more detailed information about simulated robotic soccer one can refer to Noda et al. (1998). Full details of the SMDP approach to keepaway along with extensive empirical results are available in Stone et al. (2005).

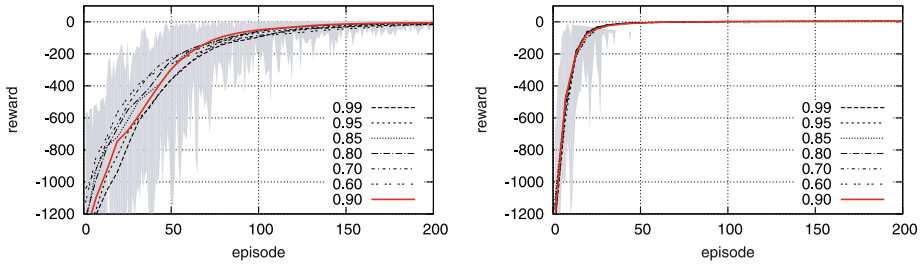


Fig. 8 The effect of λ. The median of total reward per episode over 50 runs. (**a**) SARSA(λ), and (**b**) SARSA(λ) with sequence tree in the 5 × 5 taxi problem; the *shaded areas* show the range of values falling between the first and the third quartiles for $\lambda = 0.90$. The *curves* are smoothed for visual clarity. Note that, although smaller values of λ initially perform better in SARSA(λ), their performances degrade subsequently

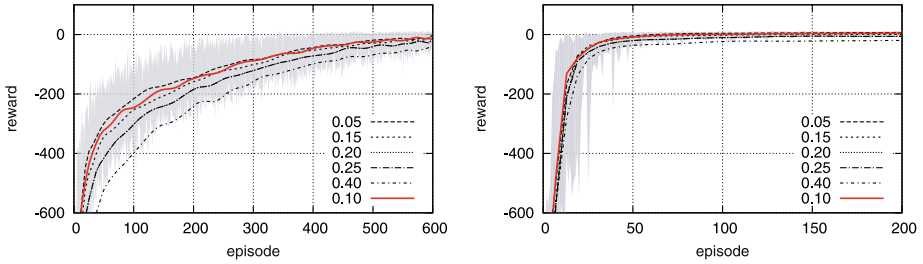


Fig. 9 The effect of ϵ in ϵ-greedy action selection mechanism. The median of total reward per episode over 50 runs. (**a**) Q-learning, and (**b**) Q-learning with sequence tree in the 5 × 5 taxi problem; the *shaded areas* show the range of values falling between the first and the third quartiles for $\epsilon = 0.10$. The *curves* are smoothed for visual clarity

the SMDP Q-learning algorithm (Bradtke and Duff 1994), we implemented problem specific hand-coded options. In the six-room maze problem, these options move the agent from any cell in a room, except the bottom rightmost one which contains the goal location, to one of two doorways that connect to neighboring rooms in minimum number of steps; in the taxi problem, they move the agent from any position to one of predefined locations in shortest possible way.[3] Unless stated otherwise, while building the sequence tree the eligibility decay rates and the eligibility thresholds are taken as 0.95 for ψ_{decay}, 0.99 for ξ_{decay}, and 0.01 for both $\psi_{threshold}$ and $\xi_{threshold}$. Note that, only these last three parameters are related to the proposed method; all the other parameters are general learning parameters. The sequence tree is generated during learning without any prior training session. At decision points, actions are chosen by following an ϵ-greedy strategy based on the expected cumulative reward values associated with the tuples. The complete list of parameters are presented in Table 1; we will analyze the effect of different values of these parameters on learning performance later in

[3]In all versions of the taxi problem, the number of predefined locations was 4. Therefore, there were four such options each corresponding to one of these locations.

Table 1 The list of parameters

General Learning Parameters		
Parameter	Default Value	Description
α	0.125	Learning rate
γ	0.9	Discount factor
λ	0.9 and 0.98	Eligibility decay rate in SARSA(λ) algorithm (see Fig. 8)
ϵ	0.1	Probability of choosing a random action in ϵ-greedy action selection mechanism (see Fig. 9)
Extended Sequence Tree Parameters		
ψ_{decay}	0.95	Eligibility decay rate for edges in the extended sequence tree (see Fig. 17)
ξ_{decay}	0.99	Eligibility decay rate for tuples in the extended sequence tree
$\{\psi, \xi\}_{threshold}$	0.01	Eligibility thresholds (see Fig. 20)

this section. The reported experiments are repeated 30 times for the keepaway problem and 50 times for all other cases.

Figures 10 and 11 show the progression of the total reward obtained per episode for the 5×5 taxi problem with one passenger and the six-room maze problems, respectively. As apparent from the learning curves, both SARSA(λ) and SMDP Q-learning converge faster compared to regular Q-learning. When the sequence tree method is applied to Q-learning and SARSA(λ), the performance of both algorithms improve substantially. In SMDP Q-learning algorithm, the agent receives higher negative rewards when options that move the agent away from the goal are erroneously selected at the beginning of training; SMDP Q-learning needs to explore and learn which options are optimal to execute and under what context. The effect of this situation can be seen in the six-room maze problem, where it causes SMDP Q-learning to converge slower compared to SARSA(λ). On the contrary, algorithms that employ sequence tree start to utilize shorter sub-optimal sequences immediately in the initial stages of learning; this results in more rapid convergence with respect to hand-coded options. The results of pairwise randomized ANOVA procedures between Q-learning with sequence tree and other algorithms indicate that in all cases except SARSA(λ) in the six-room maze problem and SARSA(λ) with sequence tree in the taxi problem, the null hypothesis of no algorithm effect can be rejected ($p < 0.001$); we also observed the same outcome when the first $n \in \{50, 100, 200, 300, 400, 500\}$, data points are omitted and the results are compared for the remaining points.[4] Furthermore, the learning curve of the SMDP Q-learning algorithm in which the extended sequence tree acts as a single online option (top rightmost sub-figure in Fig. 10) indicates that such an approach may suffer from the "forgetting" effects and result in sub-optimal behavior.

[4]Pairwise comparisons of Q-learning with sequence tree with SARSA(λ) in the six-room maze problem and SARSA(λ) with sequence tree in the 5×5 taxi problem result in $p > 0.244$ for $n = 100$ and $p > 0.863$ for $n = 50$.

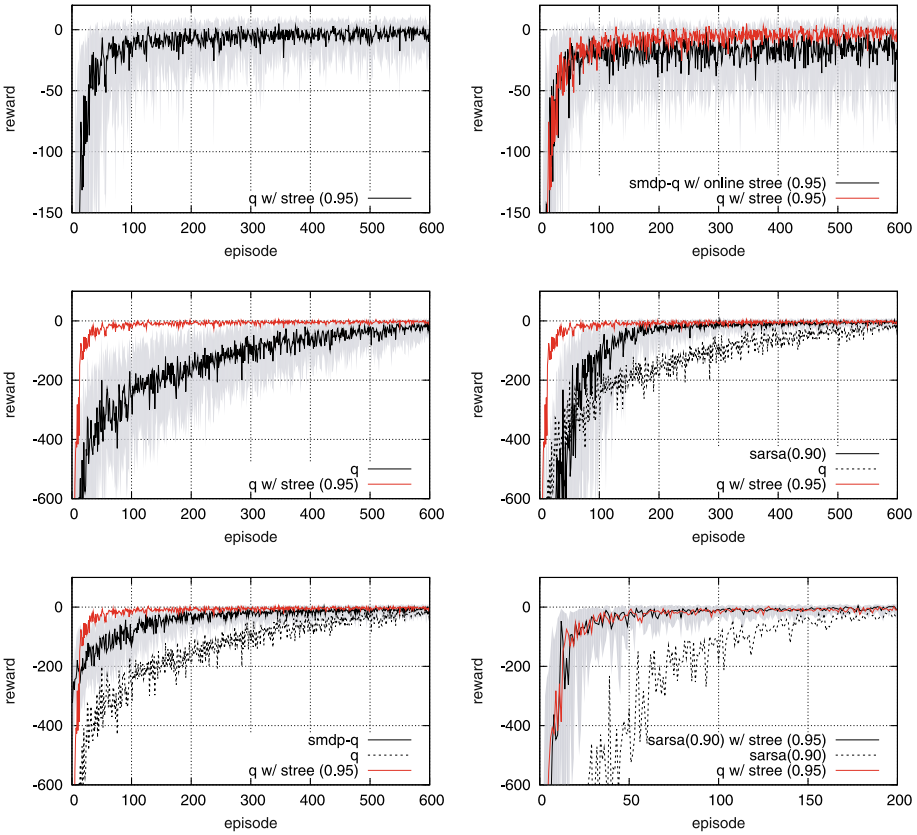


Fig. 10 The median of total reward per episode for the 5×5 taxi problem with one passenger over 50 runs; the *shaded areas* show the range of values falling between the first and the third quartiles for the algorithm denoted by the *solid black line*. The reference *curves* show the performance of the Q-learning algorithm with sequence tree ($\psi_{decay} = 0.95$). The *top rightmost figure* plots the performance of the SMDP Q-learning algorithm when the extended sequence tree is employed as a single option that evolves

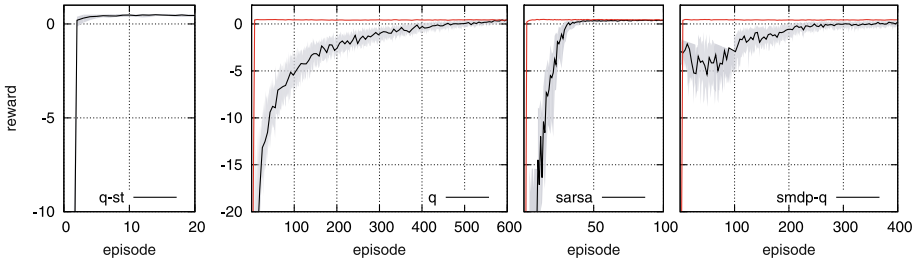


Fig. 11 The median of total reward per episode for the six-room maze problem over 50 runs. The *leftmost figure* and the reference *curves* in other figures show the performance of the Q-learning algorithm with sequence tree ($\psi_{decay} = 0.95$); the *shaded areas* show the range of values falling between the first and the third quartiles for the algorithm denoted by the *solid black line*

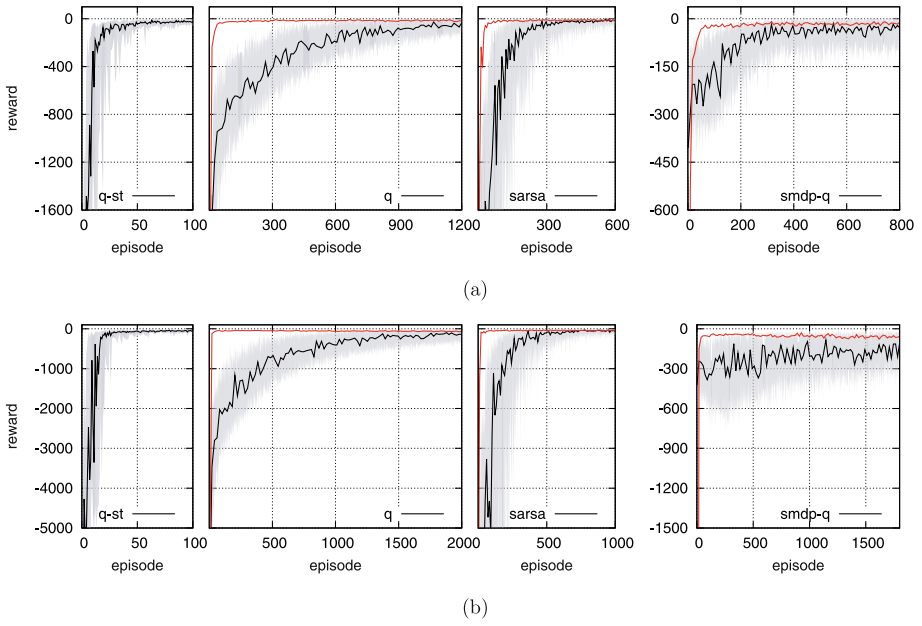


Fig. 12 The median of total reward per episode for (**a**) 8×8, and (**b**) 12×12 taxi problem with one passenger over 50 runs. The *leftmost figures* and the reference *curves* in other figures show the performance of the Q-learning algorithm with sequence tree ($\psi_{decay} = 0.95$); the *shaded areas* show the range of values falling between the first and the third quartiles for the algorithm denoted by the *solid black line*

Figure 12 shows the results for the 8×8 and 12×12 taxi problems with one passenger; these problems have larger state spaces and contain more obstacles compared to the original 5×5 taxi problem. In both cases, we observed similar learning curves as in the 5×5 version, and algorithms that are enhanced with the proposed method learn much faster compared to their regular counterparts. In both cases, the effect of the algorithm is statistically significant ($p < 0.001$ for pairwise randomized ANOVA tests in comparison to Q-learning with sequence tree; $p > 0.136$ after 1000 time-steps between SARSA(λ) and Q-learning with sequence tree in the 8×8 taxi problem).

In order to test how our approach performs in a challenging machine learning task, we implemented the method described in Stone et al. (2005) for the keepaway sub-task of simulated robotic soccer using the publicly available keepaway player framework (Stone et al. 2006). The method used by Stone et al. is an SMDP version of the SARSA(λ) algorithm with linear tile-coding function approximation and replacing eligibility traces. We integrated the extended sequence tree to their algorithm, and conducted the experiments in a 3 keepers vs. 2 takers setting playing within a 20 m $\times$ 20 m region. All players were given noiseless visual sensory information. The state representation used by a learning agent (i.e. a keeper) is a mapping of the available environment information to 13 continuous variables that are computed based on the positions of the players and center of the playing region. In order to approximate the state-action value function, 32 uniformly distributed tilings are overlaid for each variable which together form a feature vector of length 416 that maps continuous space into a finite discrete one. By using an open-addressing hashing technique, the size of the state space is further reduced. More detailed information about the state variables and this particular mapping can be found in Stone et al. (2005). Since the feature vector

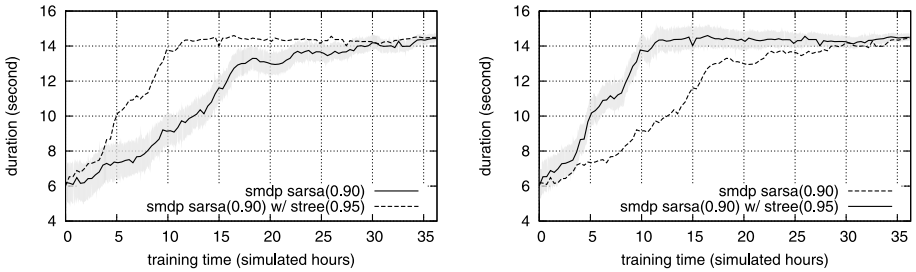


Fig. 13 The progression of the episode duration in the keepaway problem over 30 runs (higher episode duration signifies better behavior); the *shaded areas* show the range of values falling between the first and the third quartiles for the algorithm denoted by the *solid black line*. The base RL algorithm is SMDP SARSA(λ) as described in Stone et al. (2005)

spans an extremely large space and hashing does not preserve locality, i.e., states with similar feature vectors may get mapped to unrelated addresses, an alternative discretization method is needed while generating the sequence tree. For this purpose, we chose 5 most important variables out of 13 available state variables that are shown to display similar results to those obtained when all state variables are utilized (see Stone and Sutton 2001; Stone et al. 2005). Each variable is then discretized into 12 classes and together used to represent states while building and employing the sequence tree. The immediate reward received by each agent after selecting a high level skill is defined as the number of primitive time steps that have elapsed while following the high level skill. The takers use a hand-coded policy implemented in Stone et al. (2006): A taker either tries to (a) catch the ball if it is the closest or second closest taker to the ball or if no other taker can get to the ball faster than it does, or (b) position itself in order to block a pass from the keeper with the largest angle with vertex at the ball that is clear of takers. We used the default values of learning parameters and λ in SARSA(λ) is set to 0.9. The learning curves showing the progression of episode duration with respect to training time have been plotted in Fig. 13. Since the learning agents are trying to keep the ball away from the takers as long as possible, higher episode duration indicates better behavior. We repeated each experiment 30 times. The SMDP SARSA(λ) algorithm with sequence tree converges faster, achieving an episode time of around 14 seconds in almost one third of the time required by its regular counterpart. This data also supports the fact that the proposed sequence tree based method is successful in utilizing useful abstractions and improves the learning performance in more complex domains.

6.2.1 Scaling on the taxi domain

In the taxi problem, the number of situations in which sub-tasks can be applied increases with the number of passengers to be transported. This also applies to other problems; a new parameter added to the state representation leads to a larger (usually exponential) state space. Consequently, the number of instances of sub-tasks that involve only a subset of variables also increase. Therefore, more significant improvement in learning performance is expected when sub-tasks can be utilized effectively. Results for the 5×5 taxi problem with multiple passengers (from two up to four) are presented in Fig. 14. Note that the performance of the algorithms that do not make use of abstractions degrade rapidly as the number of passengers increases, and consequently common sub-tasks become more prominent. The results also demonstrate that the proposed method is effective in identifying solutions in such cases.

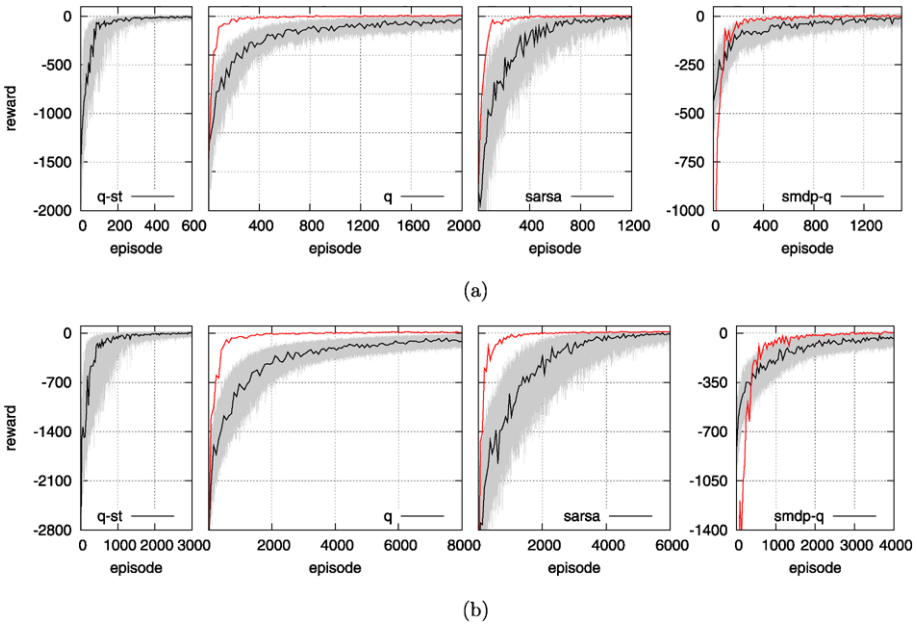


Fig. 14 The median of total reward per episode for the 5×5 taxi problem with (**a**) two, and (**b**) four passengers over 50 runs. The *left-most figures* and the reference *curves* in other figures show the performance of the Q-learning algorithm with sequence tree ($\psi_{decay} = 0.95$); the *shaded areas* show the range of values falling between the first and the third quartiles for the algorithm denoted by the *solid black line*

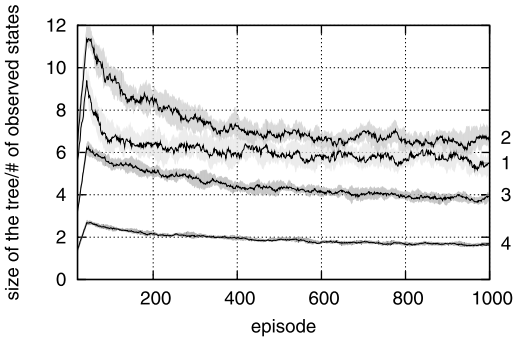


Fig. 15 The median size of the sequence trees with respect to the size of state space for the 5×5 taxi problem with one to four passengers; the *shaded areas* show the range of values falling between the first and the third quartiles. The number of passengers are indicated to the *right* of the corresponding *curves*

Based on pairwise randomized ANOVA tests, in all cases the null hypotheses of no effect of the sequence tree on the learning performance can be rejected ($p < 0.001$).

Figure 15 shows the size of the sequence tree in terms of the number of nodes with respect to the number of observed states in the state space (from the beginning up to the current episode) for different number of passengers in the 5×5 taxi problem. Note that as new passengers are added, the state space increases exponentially with respect to the number of passengers, i.e., multiplies with the number of predefined locations, whereas the relative space requirement decreases indicating that the proposed method scales well in terms of space efficiency.

6.3 Level of abstraction

Apart from the performance, other important factors that need to be considered are the structure of the sequence tree, its involvement in the learning process, and their overall impact. The content of the sequence tree is mostly determined by the π-histories generated based on the interactions of the agent with the environment. Due to the complex structure of the tree, it is not feasible to directly give a snapshot of it to expose what kind of abstractions it contains. Rather, we opt to take a more comprehensible and qualitative approach, and at various stages of the learning process examined how successful the tree is in generating abstractions belonging to similar sub-tasks.

Note that in the taxi problem the agent must first navigate to the location of the passenger to pick him up regardless of the destination of the passenger; for regions of the state space that differ only in the destination of the passenger, the sub-task to be solved is the same and one expects the agent to learn similar action sequences within these regions. This means that states which only differ in the variable corresponding to the destination of the passenger must appear in the tuple lists of the same nodes in the extended sequence tree; during the learning process, the ratio of the actual number of such states in the tuple list of a node to the number all possible such states (which we can easily enumerate for this problem) can be used as an indicator of the level of abstraction attained by the sequence tree. This ratio, which we will call *state abstraction ratio*, must be close to 1 if action sequences that involve a particular state are also applicable to other states having different destinations for the passenger.

In order to analyze how this ratio evolves, we conducted a set of experiments on the 5×5 taxi problem with one passenger. The results of the experiments are presented in Fig. 16. Each of the four columns denotes the case in which the passenger is at a specific predefined location from A to D, and each row shows the abstraction level attained by the sequence tree generated after 50, 100, 150 and 200 episodes. The intensity of shading in each cell indicates the state abstraction ratio corresponding to the state in which the agent is positioned at that cell (regardless of the destination of the passenger); black represents 1, white represents 0 and the intensity of intermediate values decreases linearly. One can observe that after 50 episodes all cells have non-white intensities which get darker with increasing number of episodes and eventually turn into black, i.e., the state abstraction ratios converge to 1; this means that the sequence tree is indeed successful in identifying abstractions that cover multiple instances of the same or similar sub-tasks starting from early stages of the learning.

6.4 The effects of the parameters

Other than the π-histories, the structure of the sequence tree also depends on the eligibility decay rate and threshold parameters that regulate the amount of information to be retained in the sequence tree. We found out that from these parameters the most prominent one that causes the most significant difference is the edge eligibility decay rate, ψ_{decay}. The results for various ψ_{decay} values presented in Fig. 17 show that the size of the sequence tree decreases considerably for both six-room maze and taxi problems as ψ_{decay} gets smaller. This is due to the fact that only more recent and commonly used sequences have the opportunity to be kept in the tree and others get eliminated. Note that since such sequences are more beneficial for the solution of the problem, unless ψ_{decay} is chosen to be small (in this case, less than 0.80) different ψ_{decay} values show near optimal behavior (see Fig. 18 and pairwise randomized ANOVA results of no ψ_{decay} effect on performance presented in Table 2 where each entry gives the p-value testing for difference between the two values of *psi* in the row

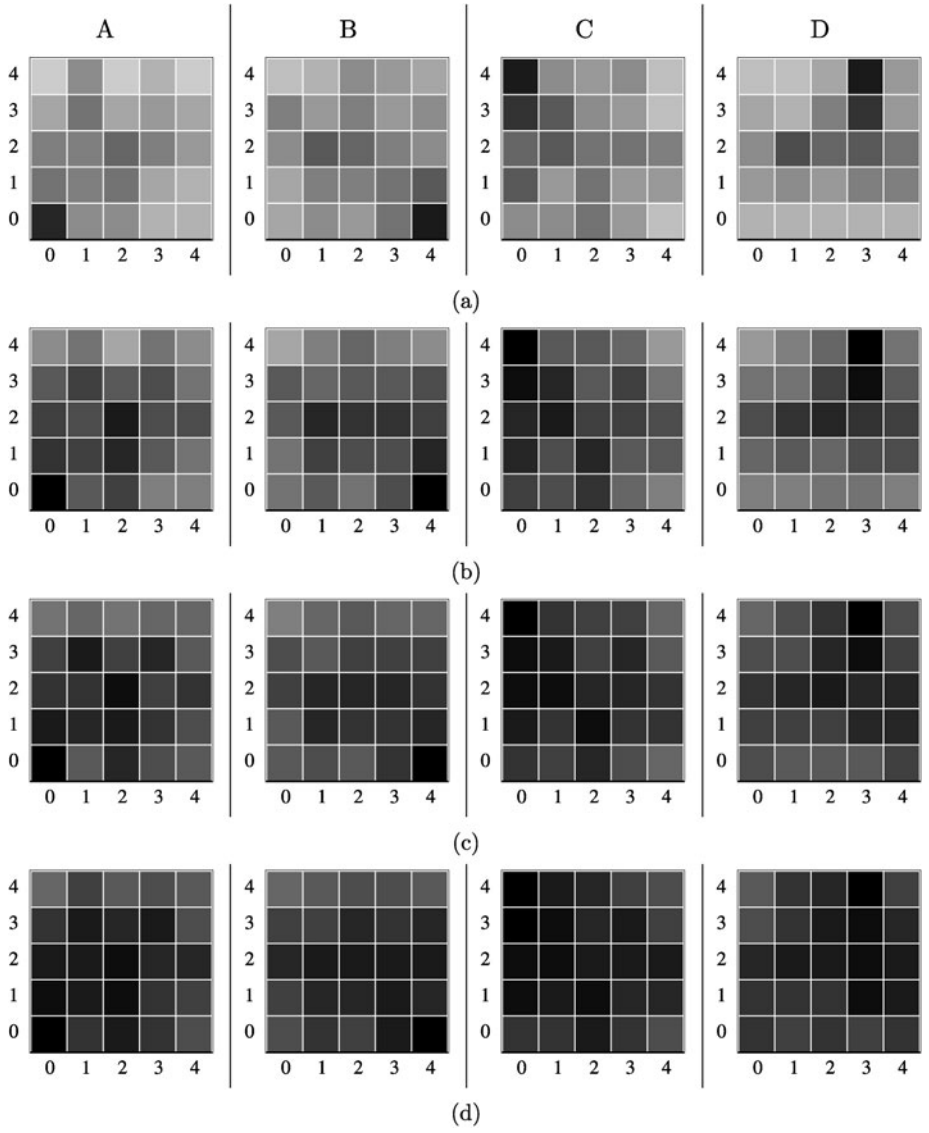


Fig. 16 State abstraction levels for four predefined locations (*A* to *D* from *left* to *right*) in 5×5 taxi problem after 50, 100, 150 and 200 (**a**–**d**) episodes. Darker colors indicate higher abstraction

and column). Hence, by selecting ψ_{decay} parameter appropriately, it is possible to reduce memory requirements without degrading the performance. We observe a similar effect for the SMDP Q-learning algorithm in which the extended sequence tree acts as a single online option (Fig. 19); however, the performance drop is more noticeable and ψ_{decay} needs to be larger in order to attain near optimal behavior. Here, we also would like to stress the inherent relationship between ψ_{decay} and $\psi_{threshold}$. Although a wide range of $\psi_{threshold}$ values perform similar for large ψ_{decay} values, as ψ_{decay} decreases large $\psi_{threshold}$ values lead to early

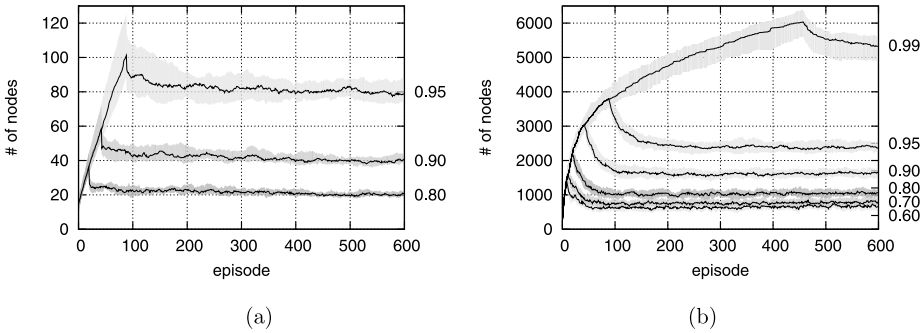


Fig. 17 The median size of sequence trees for different ψ_{decay} values for Q-learning with sequence tree in (**a**) six-room maze, and (**b**) 5×5 taxi problem with one passenger; the *shaded areas* show the range of values falling between the first and the third quartiles

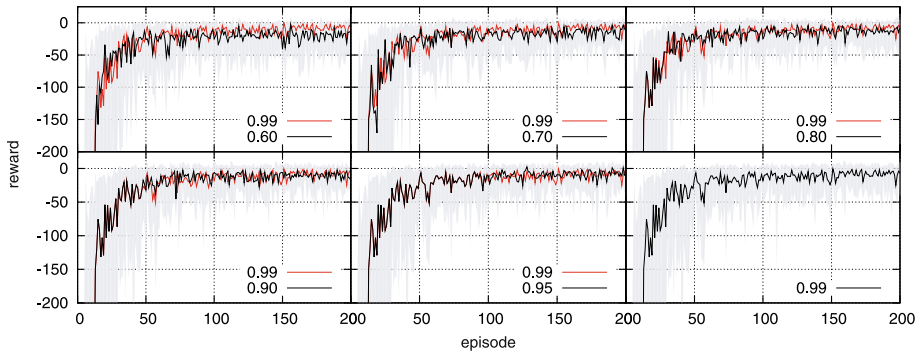


Fig. 18 The median of total reward per episode for different ψ_{decay} values for Q-learning with sequence tree in 5×5 taxi problem with one passenger; the *shaded areas* show the range of values falling between the first and the third quartiles

removal of nodes from the extended sequence tree and has a negative effect on the learning performance (Fig. 20).

6.5 Limiting the length of sequences

Other than lowering the value of the eligibility decay rate ψ_{decay}, another possible way to reduce the size of the extended sequence tree is to limit the length of the probable π^*-histories that are added to it. After generating probable histories using Algorithm 2, instead of the entire π-history $h = s_1 a_1 r_2 \ldots r_t s_t$, we can only process h up to l_{max} steps (i.e., $s_1 a_1 r_2 \ldots r_{l_{max}+1} s_{l_{max}+1}$) in Algorithm 3 by omitting the observations after l_{max} steps. This corresponds to having conditionally terminating sequences of length at most l_{max}, and therefore the maximum depth of the extended sequence tree will be bounded by l_{max}. Since shorter abstractions are prefixes of longer abstractions, and the applicability of abstractions decreases with the increase in length, it is plausible to apply pruning without drastically affecting the performance of learning. The learning curves of Q-learning with sequence tree on the 5×5 taxi problem with one passenger for different l_{max} values are presented in Fig. 21(a) and the size of the corresponding sequence trees as plotted in Fig. 21(b). We observe that

Table 2 The p-values of the sample statistic $F_{\psi_{decay}}$ for the learning performance curves of Q-learning with sequence tree for different ψ_{decay} values in (a) six-room maze and (b) 5×5 taxi problem with one passenger, and (c) SMDP Q-learning with sequence tree acting as a single online option in the 5×5 taxi problem

ψ_{decay}	0.90	0.95
0.80	**0.009**	**0.104**
0.90	1	**0.372**

(a)

ψ_{decay}	0.70	0.80	0.90	0.95	0.99
0.60	**0.305**	**0.01**	0.001	0.001	0.001
0.70	1	**0.176**	**0.039**	**0.099**	**0.321**
0.80		1	**0.456**	**0.776**	**0.735**
0.90			1	**0.683**	**0.318**
0.95				1	**0.57**

(b)

ψ_{decay}	0.95	0.99
0.90	**0.275**	**0.003**
0.95	1	**0.04**

(c)

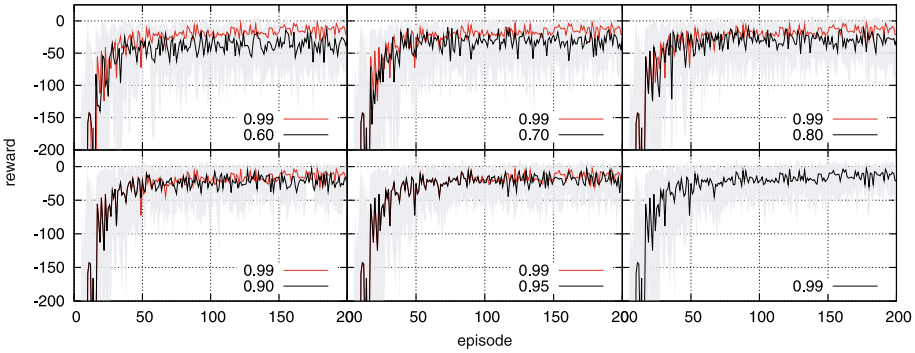


Fig. 19 The median of total reward per episode for different ψ_{decay} values for SMDP Q-learning with sequence tree acting as a single online option in 5×5 taxi problem with one passenger; the *shaded areas* show the range of values falling between the first and the third quartiles

starting from a maximum π-history length of 7, the characteristics of the learning curves are similar to the learning curves that are obtained without limiting the length of the π-histories, conforming to our expectation. The results of pairwise randomized ANOVA tests indicate that the null hypothesis of no effect of l_{max} on the learning performance cannot be rejected. The SMDP Q-learning algorithm in which the extended sequence tree acts as a single online option requires relatively longer history lengths, otherwise the performance deteriorates (Table 3).

6.6 The effect of non-determinism in the environment

In order to examine how the non-determinism of the environment affects the performance, we conducted a set of experiments by changing p_{fail}, i.e., the probability that movement actions fail, in the taxi domain. The results are presented in Fig. 22. Except for increased fluctuation in the received reward due to increased non-determinism, the proposed method preserves its behavior and methods that employ sequence tree consistently learn more rapidly compared to their regular counterparts. Based on the results of randomized ANOVA tests,

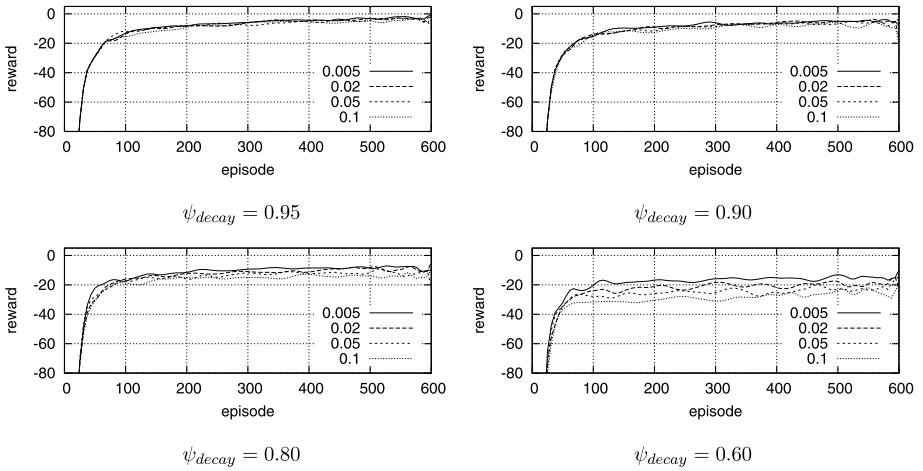


Fig. 20 The relation between the eligibility threshold $\psi_{threshold}$ and decay rate ψ_{decay}. $\psi_{threshold} \in \{0.95, 0.90, 0.80, 0.60\}$ for four different values of ψ_{decay} (0.005, 0.02, 0.05, and 0.1) in the 5×5 taxi problem with one passenger. The *curves* are smoothed for visual clarity

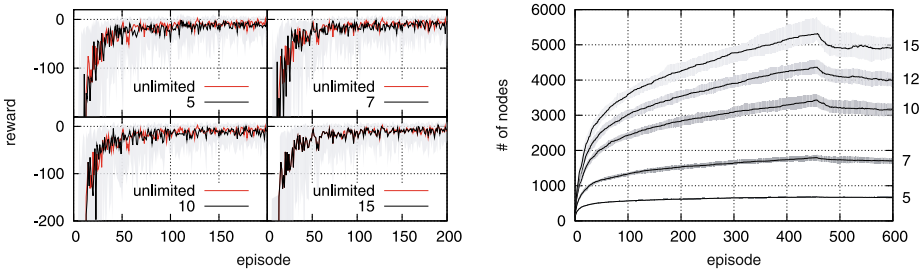


Fig. 21 (**a**) The median of total reward per episode for different maximum history lengths for Q-learning with sequence tree in the 5×5 taxi problem with one passenger, and (**b**) corresponding median size of the sequence trees; the *shaded areas* show the range of values falling between the first and the third quartiles

Table 3 The p-values for the sample statistics $F_{l_{max}}$ and $F_{interaction}$ (left and right values in each cell, respectively) for the learning performance curves of (a) Q-learning with sequence tree, and (b) SMDP Q-learning with sequence tree acting as a single online option for different maximum sequence lengths in the 5×5 taxi problem with one passenger

l_{max}	7	10	12	15	∞
5	**0.051**	0.003	0.001	0.001	0.001
7	1	**0.554**	**0.169**	**0.354**	**0.352**
10		1	**0.398**	**0.712**	**0.709**
12			1	**0.652**	**0.589**
15				1	**0.969**

(a)

l_{max}	7	10	12	15	∞
5	0.001	0.001	0.001	0.001	0.001
7	1	0.001	0.001	0.001	0.001
10		1	**0.151**	**0.013**	**0.1**
12			1	**0.253**	**0.319**
15				1	**0.873**

(b)

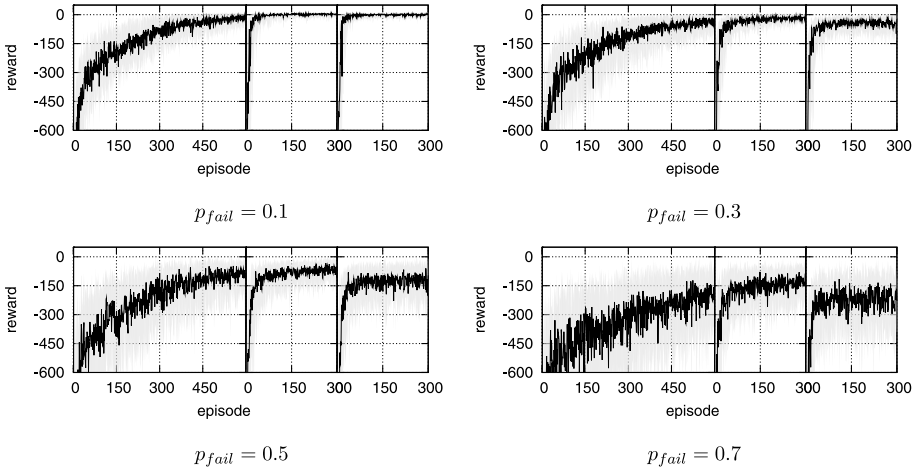


Fig. 22 Results with different levels of non-determinism (i.e. different values of p_{fail}—the probability that the movement actions fail) in the 5×5 taxi problem with one passenger. In each figure, the sub-figures show the performance of Q-learning (*left*), Q-learning with sequence tree ($\psi_{decay} = 0.95$, *middle*), and SMDP Q-learning with sequence tree acting as a single online option ($\psi_{decay} = 0.95$, *right*); the *shaded areas* show the range of values falling between the first and the third quartiles

the null hypothesis of no algorithm effect can be rejected ($p < 0.001$). The SMDP Q-learning algorithm in which the extended sequence tree acts as a single online option is more sensitive to the level of non-determinism in the environment.

6.7 The performance of the intermediate abstractions represented by the extended sequence tree

The results given so far demonstrate the on-line performance of the method, i.e., while the extended sequence tree is continuously evolving and abstractions that it represents change dynamically. As our main goal is to solve the given problem and learn an optimal behavior, this is a more efficient approach since it does not require an off-line pre-learning stage to determine the options. However, this also brings a certain weakness: it is not straight-forward to assess the quality of the evolving abstractions represented by the sequence tree in the sense that whether the abstractions at a given instant of the learning process are really meaningful and beneficial or not. In order to accomplish this, we employed the following procedure: we applied our method to the standard Q-learning algorithm, and let the extended sequence tree evolve for different number of episodes. Then, we converted each of the obtained trees into a single pseudo option, and run a separate instance of the SMDP Q-learning algorithm for each of them with an action set that includes the primitive actions and the corresponding pseudo option. This allows us to isolate and observe the effect of the discovered abstractions in a controlled manner.

The results of the experiments for the 5×5 taxi problem with one passenger are presented in Fig. 23. The learning curves of the SMDP Q-learning algorithm demonstrate that even sequence trees in their early stage of training accommodate useful abstractions. The sequence tree generated after 20 episodes is quite effective and leads to a substantial improvement. The performance of SMDP Q-learning increases with the number of episodes

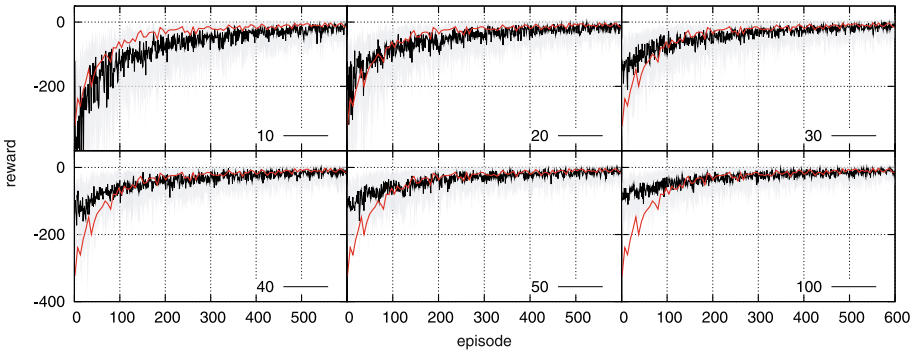


Fig. 23 SMDP Q-learning algorithm when the previously generated sequence tree is employed as a single online option; the *shaded areas* show the range of values falling between the first and the third quartiles. Each number in the key denotes the number of episodes used to generate the tree in the prior training phase. The reference *curves* show the performance of the SMDP Q-learning algorithm with predefined options

used to generate the sequence tree and then saturates. This indicates that the sequence tree based approach is successful in finding meaningful abstractions, improving them and preserving the most useful ones.

6.8 Running time

Our focus in this work and the proposed sequence tree approach is to allow agent learn more efficiently in terms of its interactions with the environment, that is to learn successful policies with *less experiences*. The results presented so far aim to demonstrate this goal under different settings and provide insight into the effects of various parameters involved. Nonetheless, in real-world applications the running time of an algorithm is an important factor; we therefore analyzed the running time of our approach in comparison to the standard reinforcement learning algorithms. As discussed in Sects. 6.4 and 6.5, the eligibility decay rate (ψ_{decay}) and the maximum history length have a direct effect on the size of the resulting sequence tree which consequently affects the running time. Hence, in the experiments we opted to test different values for these parameters as well. All algorithms and sample problems are implemented in C++ language and compiled using the GCC compiler suite on x86 architecture with maximum level of optimizations.

The running times of Q-learning with sequence tree and other algorithms for the 5×5 taxi problem with one passenger are presented in Fig. 24. For each case, we measured the total running time of 50 consecutive runs where the duration of each run is 600 episodes; we computed the average over 10 such trials and normalized the results by the average total running time of the Q-learning algorithm. We can observe that although standard reinforcement learning algorithms are more efficient in terms of execution time, unless ψ_{decay} and maximum history length are chosen to be large, the running time of the proposed approach also stays within acceptable limits outperforming SARSA(λ) when the maximum length is small because the number of updates on the sequence tree could be less than the number of updates required to propagate the eligibility trace. The results for the two and three passenger versions of the same problem, in which the duration of each run is taken as 2000 and 6000 episodes respectively, indicate that despite the relative increase in the running time the behavior of the proposed approach is consistent (Fig. 25); we would like to note that the state space increases exponentially with the number of passengers. Over all trials,

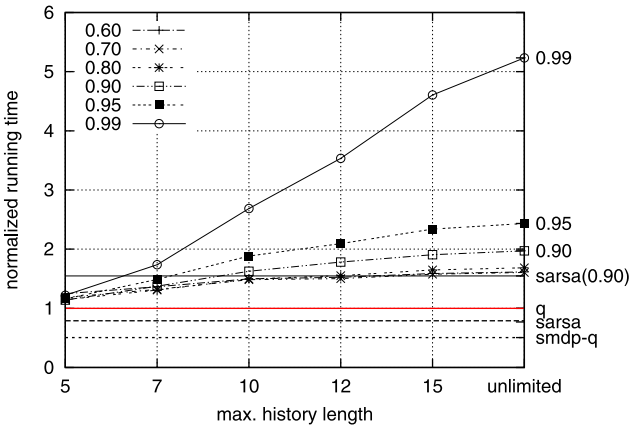


Fig. 24 The average running times of Q-learning with sequence tree for different maximum history lengths and ψ_{decay} values in the 5×5 taxi problem with one passenger; the results show the total running time of 50 consecutive independent runs repeated 10 times and are normalized by the running time of Q-learning

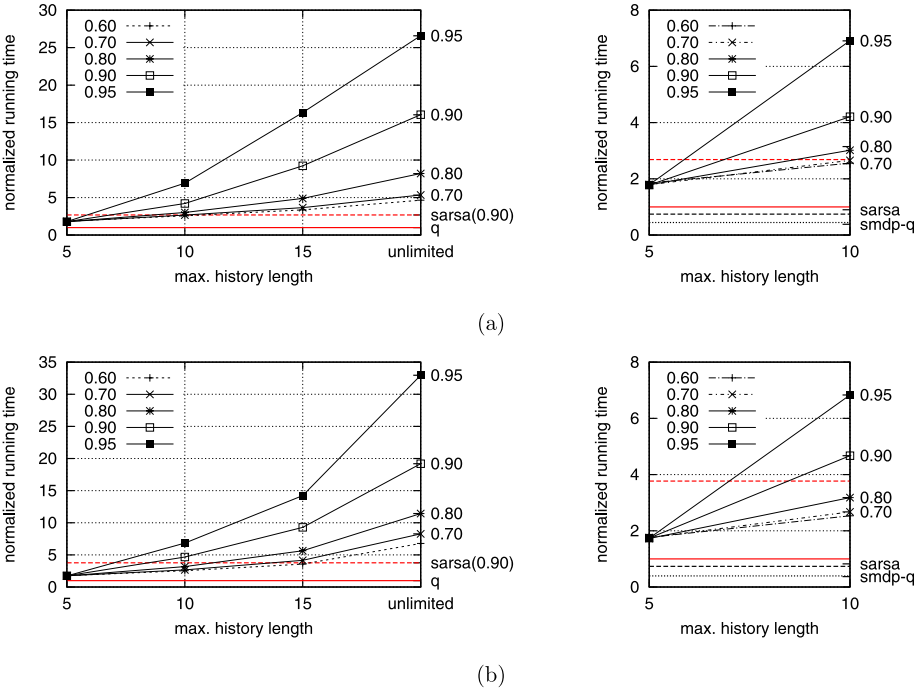


Fig. 25 The average running times of Q-learning with sequence tree for different maximum history lengths and ψ_{decay} values in the 5×5 taxi problem with (**a**) two and (**b**) three passengers; the reported results (see *right hand side*) show the total running time of 50 consecutive independent runs repeated 10 times. The running times of Q-learning and SARSA(λ) are *highlighted*

the variances of total running times were between 0.2% to 1.8% and are omitted in the figures.

6.9 Comparison to acQuire-macros algorithm

Finally, we compared the performance of our method with the *acQuire-macros* algorithm of McGovern (1998, 2002), which formed the starting point of the work presented in this manuscript and which is also based on conditionally terminating sequences. In order to compare the published results with the proposed method, the experiments are conducted on a problem already studied by McGovern, the 20×20 empty grid world problem, i.e., one room maze without any obstacles. In this problem, the agent is initially positioned at the lower left corner of the grid and tries to reach the upper right corner. The action set and dynamics of the environment are the same as in the six-room maze problem. The agent receives an immediate reward of 1 when it reaches the goal cell, and 0 otherwise. The discount rate γ is set to 0.9. In order to comply with the existing work, a learning rate of $\alpha = 0.05$ and ϵ-greedy action selection with $\epsilon = 0.05$ are used. Note that, in this problem several abstractions, such as moving diagonally as exemplified in Sect. 5.1, are quite useful to fulfill the specified task; they help to reduce the distance to the goal state which the agent must reach in as few steps as possible. Although instances of these abstractions are abundant in the policy space, shorter abstractions are subsumed by longer abstractions (moving n cells away compared to $n + 1$) and when treated as sub-goals to be found and solved must be learned independently leading to the aforementioned drawbacks.

In order to determine best parameter setting, we applied acQuire-macros algorithm using various minimum support, minimum eligibility value and minimum sequence length values. The results of the experiments show that as the minimum eligibility value gets smaller the acQuire-macros algorithm converges to optimal policy faster irrespective of the minimum support. A moderate minimum support of 0.6 performs better than a fairly high value of 0.9 that filters out most of the sequences. Lower minimum support values lead to high number of options, and in the extreme fails to converge to optimal behavior. Best result is achieved with minimum sequence length of 4. More detailed analysis of the effects of the parameters in acQuire-macros algorithm can be found in the Appendix (see Figs. 28 and 29).

Results for various learning algorithms are presented in Fig. 26. Although acQuire-macros performs better than regular Q-learning, it falls behind SARSA(λ). This is due to the fact that options are not created until sufficient number of instances are observed and there is no discrimination between options based on their expected total discounted rewards. The

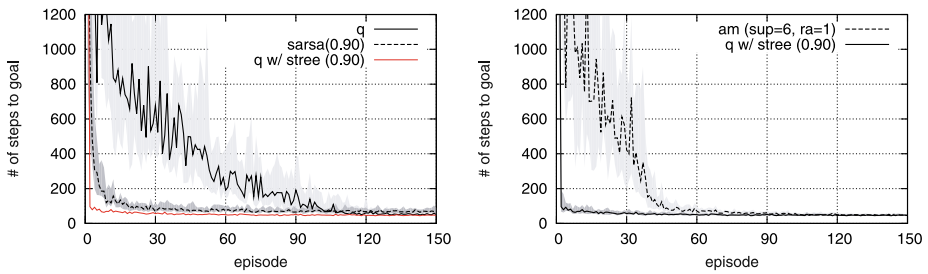


Fig. 26 Results for the 20×20 grid world problem

sequence tree based Q-learning algorithm does not suffer from such problems and shows a fast convergence by making efficient use of the abstractions in the problem.

7 Conclusions

In this paper, we proposed and analyzed the interesting and useful characteristics of a tree-based learning approach that utilizes stochastic conditionally terminating sequences. We showed how such an approach can be utilized for better representation of temporal abstractions. First, we emphasized the usefulness of discovering Semi-Markov options automatically. Then, we demonstrated the importance of constructing a dynamic and compact sequence-tree from histories. This helps identify and compactly represent frequently used sub-sequences of actions together with states that are visited during their execution. As learning progresses, this tree is constantly updated and used to implicitly locate and run the appropriately represented options. Experiments conducted on three well-known domains—with bottleneck states, repeated sub-tasks and continuous state space with macro-actions, respectively—highlighted the applicability and effectiveness of utilizing such a tree structure in the learning process. The reported test results demonstrate the advantages of the proposed tree-based learning approach over the other learning approaches described in the literature. Our future work will examine the adaptation of the method to larger domains using function approximation, such as neural networks, to store the eligibility values of states in the tuples of the nodes of the sequence tree. In addition, we want to investigate the computational cost of the proposed algorithm on theoretical basis.

Appendix

A.1 Conditionally terminating sequences and semi-Markov options

Lemma A.1 *For every conditionally terminating sequence σ, one can define a corresponding Semi-Markov option o_σ.*

Proof Let $\sigma = \langle I_1, a_1\rangle \ldots \langle I_n, a_n\rangle$ be a conditionally terminating sequence. A history $h_{t\tau} = s'_t, a'_t, r'_{t+1}, s'_{t+1}, a'_{t+1}, \ldots, r'_\tau, s'_\tau$ is said to be compatible with σ if and only if its length is less than the length of σ, for $i = t, \ldots, \tau - 1$, $s'_i \in I_{i-t+1} \wedge a'_i = a_{i-t+1}$, and $s'_\tau \in I_{\tau-t+1}$, i.e., observed states were consecutively in the continuation sets of σ starting from I_1 and at each step actions determined by σ were executed. Let H_σ denote the set of possible histories in Ω that are compatible with σ. We can construct a Semi-Markov option $o_\sigma = \langle I, \pi, \beta\rangle$ as follows:

$$I = I_{\sigma,1}$$

$$\pi(h_{t\tau}, a) = \begin{cases} 1, & \text{if } h_{t\tau} \in H_\sigma \wedge a = a_{\sigma,\tau-t+1} \\ 0, & \text{otherwise} \end{cases}$$

$$\beta(h_{t\tau}) = \begin{cases} 0, & \text{if } h_{t\tau} \in H_\sigma \\ 1, & \text{otherwise} \end{cases}$$

o_σ can only be initiated at states where σ can be initiated. When initiated at time t, the execution of o_σ continues if and only if the state observed at time $t+k$, $0 \leq k < n$, is in I_{k+1}. At time $t+k$, action a_{k+1} is selected, for every other possible action $a \neq a_{k+1}, \pi(\cdot, a) = 0$. Therefore, o_σ behaves exactly as σ. □

A.2 Stochastic conditionally terminating sequences

Stochastic conditionally terminating sequences (S-CTS) extend conditionally terminating sequences to allow alternative action sequences be followed depending on the history of events starting from its execution. They make it possible to define a broader class of abstractions in a compact form.

Definition A.1 (Stochastic conditionally terminating sequence) Let $init_\varsigma$ denote the set of states at which a stochastic conditionally terminating sequence ς can be initiated, and *first-act*$_\varsigma$ be the set of possible first actions that can be selected by ς. A stochastic conditionally terminating sequence (S-CTS) is defined inductively as:

1. A conditionally terminating sequence σ is a S-CTS; its initiation set and first action set are $init_\sigma$ and $\{\textit{first-act}_\sigma\}$, respectively.
2. Given a conditionally terminating sequence u and a S-CTS v, their concatenation $u \circ v$, defined as executing u followed by v is a S-CTS. $init_{u \circ v}$ is equal to $init_u$ and *first-act*$_{u \circ v}$ is equal to *first-act*$_u$.
3. For a given set of S-CTSs $\Sigma = \{\varsigma_1, \varsigma_2, \ldots, \varsigma_n\}$ such that each ς_i conforms to either rule (1) or rule (2) and for any two ς_i and $\varsigma_j \in \Sigma$, *first-act*$_{\varsigma_i} \cap$ *first-act*$_{\varsigma_j} = \emptyset$, i.e., their first action sets are disjoint, then $\odot_t \Sigma$ defined as defined as:

$$\odot_t \Sigma \begin{cases} \varsigma_i, & \text{if } s \in init_{\varsigma_i} \setminus \bigcup_{j \neq i} init_{\varsigma_j} \\ \mu_{\Sigma,t}, & \text{otherwise} \end{cases}$$

   is a S-CTS. In this definition, s denotes the current state, and $\mu_{\Sigma,t} : \Omega \times \Sigma \to [0, 1]$ is a branching function which selects and executes one of $\varsigma_1, \ldots, \varsigma_n$ according to a probability distribution based on the observed history of the last t steps. $\odot_t \Sigma$ behaves like ς_i if no other $\varsigma_j \in \Sigma$ is applicable at state s. $init_{\odot_t \Sigma} = init_{\varsigma_1} \cup \cdots \cup init_{\varsigma_n}$ and *first-act*$_{\odot_t \Sigma}$ = *first-act*$_{\varsigma_1} \cup \cdots \cup$ *first-act*$_{\varsigma_n}$. Note that, since they are of the form (1) or (2), the first action set of all S-CTSs in Σ has a single element. $\odot_t \Sigma$ in effect allows conditional branching of action selection and corresponds to a decision point of order $n = |\Sigma|$.
4. Nothing generated by rules other than 1–3 is a S-CTS.

Given a conditionally terminating sequence $\sigma = \langle C_1, a_1 \rangle \ldots \langle C_n, a_n \rangle$, let $\sigma^{[i:j]} = \langle C_{\sigma,i}, a_{\sigma,i} \rangle \ldots \langle C_{\sigma,j}, a_{\sigma,j} \rangle$ be the conditionally terminating sequence obtained from σ by taking continuation sets and action tuples starting from i up to and including j; let $\sigma^{[i:]}$ denote the suffix of σ which starts from the ith position (i.e. $\sigma^{[i:|\sigma|]}$).

The action pattern that combines σ_{ee}, σ_{enn} and σ_{enen} as described in Sect. 5.2 can now be represented by the S-CTS:

$$\varsigma_{\sigma_{ee},\sigma_{enn},\sigma_{enen}} = (\sigma_{ee}^{[1:1]} \cup \sigma_{enn}^{[1:1]} \cup \sigma_{enen}^{[1:1]}) \circ \odot_1 \{\sigma_{ee}^{[2:]}, (\sigma_{enn}^{[2:2]} \cup \sigma_{enen}^{[2:2]}) \circ \odot_2 \{\sigma_{enn}^{[3:]}, \sigma_{enen}^{[3:]}\}\}$$

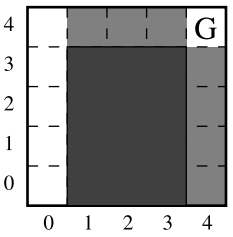


Fig. 27 After first step, $\varsigma_{\sigma_{en},\sigma_{ee}}$ behaves like σ_{en} if the current state is in the *right light shaded area*, behaves like σ_{ee} if it is in top light shaded area, and either as σ_{en} or σ_{ee} if it is in *dark shaded area*

Algorithm 7 Algorithm to construct the sequence tree corresponding to a given S-CTS.

```
1: function CREATE-SEQ-TREE(ς)                    ▷ Returns the sequence tree of S-CTS ς
2:     Create a new node root
3:     BUILD(root, ς)
4:     return root
5: end function

6: procedure BUILD(parent,u)
7:     if u = ⟨I, a⟩ then
8:         Create a new node child with init_child = I
9:         Connect parent to child by an edge with label a
10:    else if u = σ ∘ ς where σ is a CTS then
11:        Create a new node child with init_child = init_σ
12:        Connect parent to child by an edge with label first-act_σ
13:        if |σ| = 1 then
14:            BUILD(child, ς)
15:        else
16:            BUILD(child, σ^2 ∘ ς)
17:        end if
18:    else
19:        u is of the form ⊙_t{ς_1, ..., ς_n}
20:        for i = 1 to n do
21:            BUILD(parent, ς_i)
22:        end for
23:    end if
24: end procedure
```

which is by definition equivalent to (expanding σs)

$$= \langle C_{\sigma_{ee,1}} \cup C_{\sigma_{enn,1}} \cup C_{\sigma_{enen,1}}, e\rangle \circ \odot_1 \left\{ \begin{array}{l} \langle C_{\sigma_{ee,2}}, e\rangle, \\ \langle C_{\sigma_{enn,2}} \cup C_{\sigma_{enen,2}}, n\rangle \end{array} \circ \odot_2 \left\{ \begin{array}{l} \langle C_{\sigma_{enn,3}}, n\rangle, \\ \langle C_{\sigma_{enen,3}}, e\rangle\langle C_{\sigma_{enen,4}}, n\rangle \end{array} \right\} \right\}$$

Note that $\varsigma_{\sigma_{ee},\sigma_{enn},\sigma_{enen}}$, and in general a S-CTS, also favors abstractions which last for longer duration by executing $\langle C_{\sigma_{ee},2}, e\rangle$ directly if the state s observed after the termination of $\langle C_{\sigma_{ee,1}} \cup C_{\sigma_{enn,1}} \cup C_{\sigma_{enen,1}}, e\rangle$ is in $C_{\sigma_{ee},2}$ but not in $C_{\sigma_{enn,2}} \cup C_{\sigma_{enen,2}}$. Similarly, the S-CTS corresponding to the other branch is initiated at once if s is in $C_{\sigma_{enn,2}} \cup C_{\sigma_{enen,2}}$, but not in $C_{\sigma_{ee},2}$.

Given a S-CTS ς, its corresponding sequence tree can be constructed by using Algorithm 7, given next. The main function CREATE-SEQ-TREE creates a root node and then calls the auxiliary BUILD procedure to recursively construct the sequence tree representing ς. BUILD takes two parameters, a *parent* node and a S-CTS u. If u is a conditionally terminating sequence of length one, then a new node with continuation set $init_u$ is created and connected to the parent by an edge with label *first-act*$_u$. If u is of the form $\sigma \circ \varsigma$, where σ is a conditionally terminating sequence, then BUILD creates a new node, *child*, with continuation set $init_\sigma$, connects *parent* to *child* by an edge with label *first-act*$_\sigma$; *child* is connected to the sequence tree of ς if $|u| = 1$ or else to the sequence tree of $\varsigma^{[2:]} \circ \varsigma$. Otherwise, u is of the form $u = \odot_t\{\varsigma_1, \ldots, \varsigma_n\}$; for each ς_i BUILD calls itself recursively to connect *parent* to sequence tree of ς_i.

Note that if a S-CTS u is of the form $\odot_t\{\varsigma_1, \ldots, \varsigma_n\}$, then by definition each ς_i is either a conditionally terminating sequence or of the form $\sigma_i \circ \upsilon_i$, where σ_i is a conditionally terminating sequence. Therefore, at every call to BUILD, a new node representing an action choice is created either directly (lines 8 and 11) or indirectly (line 21). As a result, CREATE-SEQ-TREE requires linear time with respect to the total number of action sequences that can be generated by the S-CTS ς to construct the corresponding sequence tree.

Instead of creating more functional and complex S-CTSs from scratch, one can extend the union operation defined in Definition 5.2 for conditionally terminating sequences to combine behaviors of a conditional terminating sequence and a S-CTS. As we will show later, this also makes it possible to represent a set of conditionally terminating sequences as a single S-CTS. The extension is not trivial since one needs to consider the branching structure of a S-CTS. For this purpose we define a time dependent operator $\otimes_t$.

Definition A.2 (Combination operator) Let u be a conditionally terminating sequence and v be a S-CTS.[5] The binary operator $\otimes_t$, when applied to u and v, constructs a new syntactically valid S-CTS $u \otimes_t v$ that behaves both like u and v, and is defined recursively as follows, depending on the form of v:

1. If v is a conditionally terminating sequence, then
   - If action sequence of u is a prefix of action sequence of v (or vice versa), then $u \otimes_t v = u \cup v$ (or $v \cup u$).
   - If first actions of u and v are different from each other, then $u \otimes_t v = \odot_t\{u, v\}$.
   - Otherwise, action sequences of u and v have a maximal common prefix of length $k-1$, and $u \otimes_t v = (u^{[1:k-1]} \cup v^{[1:k-1]}) \circ (\odot_{t+k}\{u^{[k:]}, v^{[k:]}\})$.
2. If $v = \sigma \circ \varsigma$, where σ is a conditionally terminating sequence, then,
   - If the action sequence of u is a prefix of action sequence of σ, then $u \otimes_t v = (\sigma \cup u) \circ \varsigma$.
   - If action sequence of σ is a prefix of action sequence of u, then $u \otimes_t v = (\sigma \cup u^{[1:|\sigma|]}) \circ (u^{[|\sigma|+1:]} \otimes_{t+|\sigma|+1} \varsigma)$.
   - if first actions of u and σ are different from each other, then $u \otimes_t v = \odot_t\{u, v\}$.
   - Otherwise, action sequences of u and σ differ at a position $k \leq |\sigma|$, and $u \otimes_t v = (\sigma^{[1:k-1]} \cup u^{[1:k-1]}) \circ (\odot_{t+k}\{u^{[k:]}, \sigma^{[k:]} \circ \varsigma\})$.

[5] It is also possible to define a more general combination operator that acts on two S-CTS. However the definition is more complicated and the operator is not required for the algorithms in this paper, therefore we preferred not to include it.

3. if $v = \odot_\cdot\{\varsigma_1, \ldots, \varsigma_n\}$, then

$$u \otimes_t v = \begin{cases} \odot_t\{\varsigma_1, \ldots, \varsigma_{i-1}, u \otimes_t \varsigma_i, \varsigma_{i+1}, \ldots, \varsigma_n\} & \text{if } \textit{first-act}_u \in \textit{first-act}_{\sigma_i} \\ \odot_t\{\varsigma_1, \ldots, \varsigma_n, u\} & \text{otherwise.} \end{cases}$$

The operator $\otimes_t$ combines u and v by either directly unifying u with a prefix of v, or by creating a new branching condition or updating an existing one depending on the action sequence of u and the structure of v. When v is represented using a sequence tree T, it can easily be extended to represent $u \otimes_t v$ by starting from the root node of the tree and following edges that match the action sequence of u. Let *current* denote the active node of T, which is initially the root node. At step k, if there exists an edge with label $a_{u,k}$ connecting *current* to node n, then the kth continuation set of u is added to the continuation set of n and *current* is set to n. Otherwise, there are three possible cases depending on the number of out-going edges of *current*. In all cases, a new sequence tree for $u^{[k:]}$ is created and connected to *current* by unifying the root node of the created tree with *current*. If *current* has a single out-going edge, then it becomes a decision point of order 2. If *current* is already a decision point, then its order increases by one. The construction of the sequence tree of $u \otimes_t v$ from the sequence tree of v is linear in the length of u and completes at most after $|u|$ steps.[6]

One important application of the $\otimes_t$ operator, as we show next, is that given a set of conditionally terminating sequences to be used in a reinforcement learning problem, by iteratively applying $\otimes_t$ one can obtain a single S-CTS which represents the given conditionally terminating sequences and extend their overall behavior to allow different action sequences be followed depending on the history of observed events.

Definition A.3 (Combination of a set of CTSs) Let $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ be a set of conditionally terminating sequences and assume that the sequence to be initiated at state s is chosen on the basis of the probability distribution $P(s, \cdot)$ determined by a given function $P : S \times \Sigma \to [0, 1]$. The S-CTS $\prod \Sigma$ defined as

$$\prod \Sigma = \begin{cases} \sigma & \text{if } \Sigma = \{\sigma\} \\ \sigma_1 \otimes_0 \prod\{\sigma_2, \ldots, \sigma_n\} & \text{otherwise} \end{cases}$$

such that the branching function $\mu_{\{\varsigma_1,\ldots,\varsigma_k\},t}$ at decision point $\odot_t\{\varsigma_1, \ldots, \varsigma_k\}$ satisfies

$$\mu_{\{\varsigma_1,\ldots,\varsigma_k\},t}(\eta, \varsigma) = \max\{P(s, \sigma_i) | \sigma_i \in \Sigma, A_{\sigma_i}^{1,t-1} = \Gamma_{\eta,t-1} \text{ and } a_{\sigma_i,t} \in \textit{first-act}_\varsigma\}$$

where $\Gamma_{\eta,t}$ is the sequence of actions taken during the last t steps of history $\eta \in \Omega$, is called the combination of CTSs in Σ.

Suppose that the conditionally terminating sequence to be initiated at state s is chosen on $\prod \Sigma$ combines sequences in Σ one by one, and $\mu_{\cdot,t}$ selects a branch based on the initiation probability of conditionally terminating sequences that are compatible with the

[6] Proof is by induction on u.

sequence of actions observed until time t. Suppose that $\prod \Sigma$ is initiated at state s, and let $s_1 = s, s_2, \ldots, s_k$ and $a_1, \ldots, a_{k-1}$ be the sequence of observed states and actions selected by $\prod \Sigma$ until termination, respectively. Then, by construction of $\prod \Sigma$, for each $i = 1, \ldots, k-1$ there exists a conditionally terminating sequence $\sigma_i \in \Sigma$ such that $s_i \in C_{\sigma_i,i}$ and the action sequence of σ_i starts with $a_1 \ldots a_i$ (i.e., $A_{\sigma_i}^{1,i} = a_1 \ldots a_i$). Furthermore, one can prove that if $\sigma_\tau \in \Sigma$ is selected by P at state s and executed successfully $|\sigma_\tau|$ steps until termination, then initiated at s, $\prod \Sigma$ takes exactly the same actions as σ_τ, and exhibits the same behavior as we show next.

Theorem A.1 *If $\tau \in \Sigma$ is selected by P at state s and executed successfully $|\tau|$ steps until termination, then initiated at s and given the same observations, $\prod \Sigma$ takes exactly the same actions as τ, and exhibits the same behavior.*

Proof Let $s = s_1, s_2, \ldots, s_{|\tau|}$ be the sequence of observed states during the execution of τ. By definition, these states are members of the initiation sets of tuples in τ, i.e., for all $i = 1..|\tau|$, $s_i \in C_{\tau,i}$. Let u be a S-CTS, and a be an action in $\mathit{first\text{-}act}_u$. The behavior of u after selecting action a can be represented by a S-CTS, $u \to a$, defined as follows:

- If u is a conditionally terminating sequence then $u \to a = u^{[2:]}$.
- If $u = \sigma \circ \varsigma$ where σ is a conditionally terminating sequence then

$$u \to a = \begin{cases} \sigma^{[2:]} \circ \varsigma & \text{if } |\sigma| > 1 \\ \varsigma & \text{otherwise} \end{cases}$$

- If $u = \odot\{\varsigma_1, \ldots, \varsigma_n\}$, then there exists a unique σ_i such that $a \in \mathit{first\text{-}act}_{\varsigma_i}$ and $u \to a = \varsigma_i \to a$.

Suppose that $\prod \Sigma$ chose actions $a_{\tau,1}, \ldots, a_{\tau,k-1}$ followed by $a' \neq a_{\tau,k}$. Let $\prod \Sigma^i$ denote the resulting S-CTS after selecting actions $a_{\tau,1}, \ldots, a_{\tau,i}$, i.e., $\prod \Sigma^i = \prod \Sigma \to a_{\tau,1} \to \cdots \to a_{\tau,i}$. By construction of $\prod \Sigma$, $s_k \in \mathit{init}_{\prod \Sigma^{k-1}}$ and $a_{\tau,k} \in \mathit{first\text{-}act}_{\prod \Sigma^{k-1}}$. Depending on the form of $\prod \Sigma^{k-1}$, we have the following cases:

- $\prod \Sigma^{k-1} = \sigma \circ \varsigma$, where σ is a conditionally terminating sequence. Hence, $s_k \in \mathit{init}_\sigma$ and $a' = a_{\sigma,1} = a_{\sigma_\tau,k}.\bot$
- $\prod \Sigma^{k-1} = \odot_k\{\varsigma_1, \ldots, \varsigma_n\}$. Since $a_{\tau,k} \in \mathit{first\text{-}act}_{\prod \Sigma^{k-1}}$, by definition, there exists a S-CTS ς_ψ which contains $a_{\tau,k}$ in its first action set, i.e., $a_{\tau,k} \in \mathit{fist\text{-}act}_{\varsigma_\psi}$, and therefore s_k is in the initiation set of ς_ψ. Let X be the set of S-CTSs $\{\varsigma_1, \ldots, \varsigma_n\}$, which can continue from state s_k, i.e., $X = \{\varsigma_i : s_k \in \mathit{init}_{\varsigma_i}\}$. If $|X| = 1$, then $a' \in \mathit{first\text{-}act}_{\varsigma_\psi}$; but by the construction of a S-CTS $\mathit{first\text{-}act}_{\varsigma_\psi} = \{a_{\tau,k}\}$, and consequently $a' = a_{\tau,k}$. Otherwise, by definition, we have

$$\mu_{X,k}(\eta, \varsigma_i) = \max\{P(s, \sigma_j) : \sigma_j^{1,k-1} = \tau^{1,k-1} \text{ and } a_{\sigma_j,k} \in \mathit{first\text{-}act}_{\varsigma_i}\}$$

  But, for all $\varsigma_i \in X$ other than ς_τ, we have $\mu_{X,k}(\eta, \varsigma_i) < \mu_{X,k}(\eta, \varsigma_\psi) = P(s, \sigma_\tau)$, since σ_τ is selected by P, and thus $a' \in \mathit{first\text{-}act}_{\varsigma_\psi} = \{a_{\sigma_\tau,k}\}.\bot$

Both cases lead to a contradiction, completing the proof. □

Note that, the total number of action sequences in $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ is $\sum_{i=1}^n |\sigma_i|$ and hence it is possible to build the corresponding sequence tree for $\prod \Sigma$ in linear time.

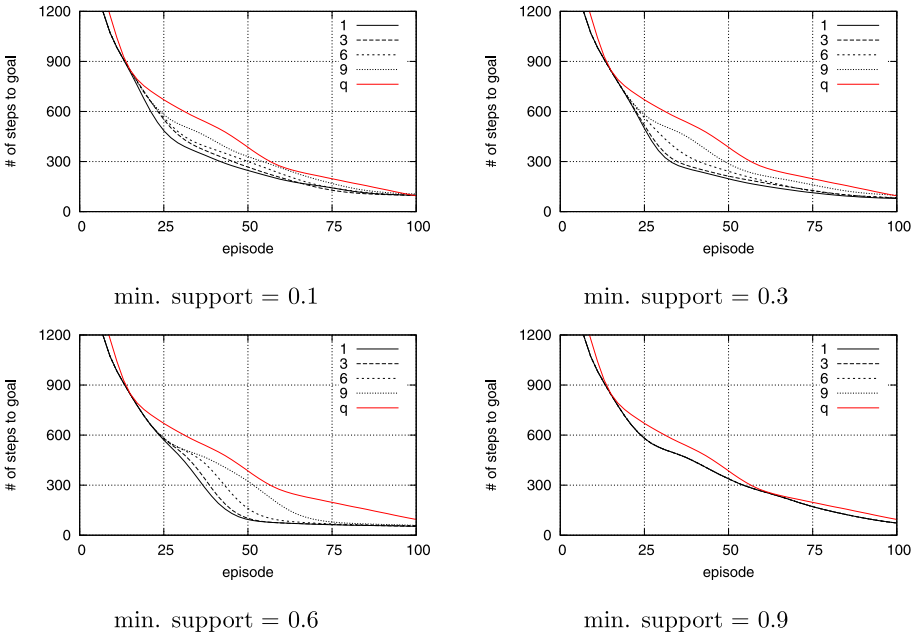
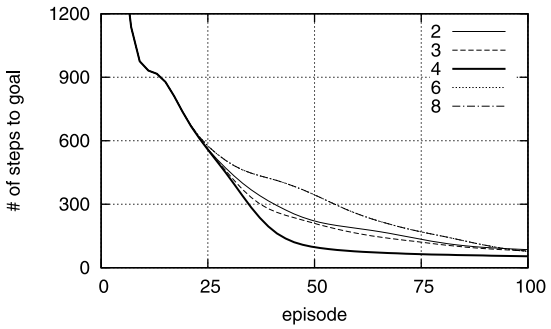


Fig. 28 Results for the acQuire-macros algorithm using different values of minimum support on the 20×20 grid world problem

Fig. 29 Results for the acQuire-macros algorithm using different minimum sequence lengths on 20×20 grid world problem. Minimum support and minimum eligibility values are taken as 0.6 and 3, respectively; the *curves* are smoothed for visual clarity



A.3 The effect of parameters in acQuire-macros algorithm

Figures 28 and 29 show plots justifying the choice of parameters for the acQuire-macros algorithm. In Fig. 28 the learning curves for different minimum eligibility values of 1, 3, 6 and 9 are plotted and compared with regular Q-learning; the curves are smoothed for visual clarity; minimum sequence length is taken as 4.

A.4 Auxiliary results

Each entry in Table 4 gives the p-value testing for difference between the two values of λ in the row and column. On the other hand, each entry in Table 5 gives the p-value testing for difference between the two values of ϵ in the row and column.

Table 4 The p-values of the sample statistics F_λ and $F_{interaction}$ (left and right values in each cell, respectively) for the pairwise comparison of different values of λ. (a) SARSA(λ), and (b) SARSA(λ) with sequence tree ($\psi_{decay} = 0.95$)

λ	0.70		0.80		0.85		0.90		0.95		0.99	
0.60	**0.131**	**0.193**	**0.004**	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
0.70	1	1	0.001	**0.006**	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
0.80			1	1	0.001	**0.054**	0.001	0.001	0.001	0.001	0.001	0.001
0.85					1	1	0.001	0.001	0.001	0.001	0.001	0.001
0.90							1	1	0.001	**0.007**	0.001	0.001
0.95									1	1	0.001	**0.021**

(a)

λ	0.70		0.80		0.85		0.90		0.95		0.99	
0.60	**0.011**	**0.61**	**0.003**	**0.207**	0.001	**0.116**	0.001	**0.024**	0.001	**0.003**	0.001	**0.006**
0.70	1	1	**0.901**	**0.446**	**0.052**	**0.699**	**0.003**	**0.843**	0.001	**0.25**	0.001	**0.06**
0.80			1	1	**0.043**	**0.472**	**0.002**	**0.245**	0.001	**0.025**	0.001	**0.074**
0.85					1	1	**0.162**	**0.16**	0.001	**0.354**	0.001	**0.029**
0.90							1	1	**0.019**	**0.049**	0.001	**0.028**
0.95									1	1	**0.022**	**0.009**

(b)

Table 5 The p-values of the sample statistics F_ϵ and $F_{interaction}$ (left and right values in each cell, respectively) for the pairwise comparison of different values of ϵ. (a) Q-learning, and (b) Q-learning with sequence tree ($\psi_{decay} = 0.95$)

λ	0.10		0.15		0.20		0.25		0.40	
0.05	0.001	**0.872**	0.001	**0.104**	0.001	0.001	0.001	0.001	0.001	0.001
0.10	1	1	0.001	**0.762**	0.001	**0.179**	0.001	0.001	0.001	0.001
0.15			1	1	0.001	**0.487**	0.001	**0.003**	0.001	0.001
0.20					1	1	0.001	**0.539**	0.001	0.001
0.25							1	1	0.001	0.001

(a)

λ	0.10		0.15		0.20		0.25		0.40	
0.05	0.001	**0.186**	0.001	**0.005**	0.001	**0.043**	0.001	0.001	0.001	0.001
0.10	1	1	0.001	**0.251**	0.001	**0.932**	0.001	**0.018**	0.001	0.001
0.15			1	1	0.001	**0.099**	0.001	**0.4**	0.001	0.001
0.20					1	1	0.001	**0.136**	0.001	0.001
0.25							1	1	0.001	**0.048**

(b)

References

Asadi, M., & Huber, M. (2005). Autonomous subgoal discovery and hierarchical abstraction for reinforcement learning using Monte Carlo method. In M. M. Veloso, & S. Kambhampati (Eds.), *AAAI* (pp. 1588–1589). Menlo Park/Cambridge: AAAI Press/MIT Press.

Barto, A. G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, *13*(4), 341–379.

Bellman, R. (1957). *Dynamic programming*. Princeton: Princeton University Press.

Bianchi, R. A., Ribeiro, C. H., & Costa, A. H. (2008). Accelerating autonomous learning by using heuristic selection of actions. *Journal of Heuristics*, *14*(2), 135–168.

Bradtke, S. J., & Duff, M. O. (1994). Reinforcement learning methods for continuous-time Markov decision problems. In G. Tesauro, D. Touretzky, & T. Leen (Eds.), *Advances in neural information processing systems* (Vol. 7, pp. 393–400). Cambridge: MIT Press.

Chen, F., Gao, Y., Chen, S., & Ma, Z. (2007). Connect-based subgoal discovery for options in hierarchical reinforcement learning. In *ICNC '07: Proceedings of the third international conference on natural computation* (pp. 698–702). Los Alamitos: IEEE Computer Society.

Degris, T., Sigaud, O., & Wuillemin, P.-H. (2006). Learning the structure of factored Markov decision processes in reinforcement learning problems. In *ICML '06: Proceedings of the 23rd international conference on machine learning* (pp. 257–264). New York: ACM.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Digney, B. (1998). Learning hierarchical control structure for multiple tasks and changing environments. In *Proceedings of the fifth conference on the simulation of adaptive behavior: SAB 98*.

Girgin, S., Polat, F., & Alhajj, R. (2006a). Effectiveness of considering state similarity for reinforcement learning. In *LNCS*. *The international conference on intelligent data engineering and automated learning*. Berlin: Springer.

Girgin, S., Polat, F., & Alhajj, R. (2006b). Learning by automatic option discovery from conditionally terminating sequences. In *The 17th European conference on artificial intelligence*. Amsterdam: IOS Press.

Girgin, S., Polat, F., & Alhajj, R. (2007). State similarity based approach for improving performance in RL. In *LNCS*. *The international joint conference on artificial intelligent*. Berlin: Springer.

Goel, S., & Huber, M. (2003). Subgoal discovery for hierarchical reinforcement learning using learned policies. In I. Russell, & S. M. Haller (Eds.), *FLAIRS conference* (pp. 346–350). Menlo Park: AAAI Press.

Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T., & Boutilier, C. (1998). Hierarchical solution of Markov decision processes using macro-actions. In *Uncertainty in artificial intelligence* (pp. 220–229).

Hengst, B. (2002). Discovering hierarchy in reinforcement learning with HEXQ. In *the International conference on machine learning*. San Mateo: Morgan Kaufman.

Jonsson, A., & Barto, A. G. (2001). Automated state abstraction for options using the u-tree algorithm. In T. K. Leen, T. G. Dietterich, & V. Tresp (Eds.), *Advances in neural information processing systems 13* (pp. 1054–1060). Cambridge: MIT Press.

Kazemitabar, S. J., & Beigy, H. (2009). Automatic discovery of subgoals in reinforcement learning using strongly connected components. In M. Köppen, N. K. Kasabov, & G. G. Coghill (Eds.), *Lecture notes in computer science: Vol. 5506*. *ICONIP (1)* (pp. 829–834). Berlin: Springer.

Kozlova, O., Sigaud, O., & Meyer, C. (2009). Automated discovery of options in factored reinforcement learning. In *Proceedings of the ICML/UAI/COLT workshop on abstraction in reinforcement learning* (pp. 24–29), Montreal, Canada.

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

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, *8*(3–4), 293–321.

Mahadevan, S., Marchallek, N., Das, T. K., & Gosavi, A. (1997). Self-improving factory simulation using continuous-time average-reward reinforcement learning. In *Proceedings of the 14th international conference on machine learning* (pp. 202–210). San Mateo: Morgan Kaufmann.

Mannor, S., Menache, I., Hoze, A., & Klein, U. (2004). Dynamic abstraction in reinforcement learning via clustering. In *ICML '04: Proceedings of the 21st international conference on machine learning* (pp. 71–78). New York: ACM.

McGovern, A. (1998). Acquire-macros: an algorithm for automatically learning macro-actions. In *The neural information processing systems conference (NIPS'98) workshop on abstraction and hierarchy in reinforcement learning*.

McGovern, A. (2002). *Autonomous discovery of temporal abstractions from interactions with an environment*. Ph.D. thesis, University of Massachusetts Amherts, May 2002.

McGovern, A., & Barto, A. G. (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. In *ICML '01: Proceedings of the 18th international conference on machine learning* (pp. 361–368). San Mateo: Morgan Kaufmann.

McGovern, A., & Sutton, R. S. (1998). *Macro-actions in reinforcement learning: an empirical analysis*. Technical Report 98-79, University of Massachusetts, Department of Computer Science.

Menache, I., Mannor, S., & Shimkin, N. (2002). Q-cut—dynamic discovery of sub-goals in reinforcement learning. In *ECML '02: Proceedings of the 13th European conference on machine learning* (pp. 295–306). London: Springer.

Noda, I., Matsubara, H., Hiraki, K., & Frank, I. (1998). Soccer server: a tool for research on multiagent systems. *Applied Artificial Intelligence*, *12*(2–3), 233–250.

Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. In *NIPS '97: Proceedings of the 1997 conference on advances in neural information processing systems 10* (pp. 1043–1049). Cambridge: MIT Press.

Parr, R. E. (1998). *Hierarchical control and learning for Markov decision processes*. Ph.D. thesis, University of California at Berkeley.

Piater, J. H., Cohen, P. R., Zhang, X., & Atighetchi, M. (1998). A randomized ANOVA procedure for comparing performance curves. In *ICML '98: Proceedings of the fifteenth international conference on machine learning* (pp. 430–438). San Mateo: Morgan Kaufmann.

Precup, D., Sutton, R. S., & Singh, S. P. (1998). Theoretical results on reinforcement learning with temporally abstract options. In *European conference on machine learning* (pp. 382–393).

Simsek, O., & Barto, A. G. (2004). Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *ICML '04: Proceedings of the 21st international conference on machine learning*. Banff, Canada.

Simsek, O., Wolfe, A. P., & Barto, A. G. (2005). Identifying useful subgoals in reinforcement learning by local graph partitioning. In *ICML '05: Proceedings of the 22nd international conference on machine learning*.

Stolle, M., & Precup, D. (2002). Learning options in reinforcement learning. In *Proceedings of the 5th international symposium on abstraction, reformulation and approximation* (pp. 212–223). London: Springer.

Stone, P., & Sutton, R. S. (2001). Scaling reinforcement learning toward RoboCup soccer. In *Proceedings of the eighteenth international conference on machine learning* (pp. 537–544). San Mateo: Morgan Kaufmann.

Stone, P., Sutton, R. S., & Kuhlmann, G. (2005). Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, *13*(3), 165–188.

Stone, P., Kuhlmann, G., Taylor, M. E., & Liu, Y. (2006). Keepaway soccer: from machine learning testbed to benchmark. In I. Noda, A. Jacoff, A. Bredenfeld, & Y. Takahashi (Eds.), *RoboCup-2005: Robot Soccer World Cup IX* (Vol. 4020, pp. 93–105). Berlin: Springer.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: an introduction*. Cambridge: MIT Press. A Bradford Book.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: a framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*(1–2), 181–211.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3/4), 279–292.

Zang, P., Zhou, P., Minnen, D., & Isbell, C. (2009). Discovering options from example trajectories. In *ICML '09: Proceedings of the 26th annual international conference on machine learning* (pp. 1217–1224). New York: ACM.