

Generating Hierarchical Structure in Reinforcement Learning from State Variables

Bernhard Hengst

School of Computer Science and Engineering
University of New South Wales, UNSW Sydney 2052 AUSTRALIA
bernhardh@cse.unsw.edu.au

Abstract. This paper presents the CQ algorithm which decomposes and solves a Markov Decision Process (MDP) by automatically generating a hierarchy of smaller MDPs using state variables. The CQ algorithm uses a heuristic which is applicable for problems that can be modelled by a set of state variables that conform to a special ordering, defined in this paper as a “nested Markov ordering”. The benefits of this approach are: (1) the automatic generation of actions and termination conditions at all levels in the hierarchy, and (2) linear scaling with the number of variables under certain conditions. This approach draws heavily on Dietterich's MAXQ value function decomposition and Hauskrecht, Meuleau, Kaelbling, Dean, Boutillier's and others region based decomposition of MDPs. The CQ algorithm is described and its functionality illustrated using a four room example. Different solutions are generated with different numbers of hierarchical levels to solve Dietterich's taxi tasks.

1 Introduction

Reinforcement learning (RL) is known not to scale well as the number of state variables increases in the state description of a problem.

Many tasks tackled by agents involve repeatable sub-tasks. If a reinforcement learner could learn these sub-tasks separately and then invoke them at a higher level as a composite or abstract action, the learner could save itself the effort of relearning each sub-task for every situation in which it was required and help scale up RL problems.

An example will help to clarify the issue. Sutton, Precup and Singh [10] discuss a traveller journeying to a distant city who needs to decide whether to fly, drive or take a taxi. Each of these possible actions is a sub-task that requires still smaller steps for its execution. Calling a taxi may involve finding a telephone, dialling each digit, etc. Which city the traveller has in mind for the destination is generally not relevant for calling-a-taxi. In other words, the traveller must still find a telephone and dial the number, etc if the taxi action is chosen, no matter which destination city is in mind.

There are advantages in learning the calling-a-cab sub-task only once. A reinforcement learner that treats each city as a special case will relearn the calling-a-taxi sub-task for each city. By contrast a hierarchical learner need only learn the sub-task once and *reused* it for each city.

Conceptually, this is much like a computer programmer calling the same subroutine many times throughout the main program, rather than repeating the same code over and over again throughout the program.

Several researchers have used a hierarchical approach in reinforcement learning to tackle related issues ([3],[4],[5],[6],[9],[10],[11]). More recently Dietterich [2] showed how some problems, decomposed by a MAXQ graph, benefit from sub-task reuse. Dietterich requires a designer to specify the hierarchical structure of the sub-tasks. For each sub-task, the designer is required to define (1) active and termination states, (2) allowable actions and (3) safe state abstractions, (4) pseudo-reward functions and (5) hierarchical credit assignment, to complete the specification of the hierarchical reinforcement learner. This process requires considerable effort and is error prone.

A hierarchical reinforcement learning algorithm called CQ will be presented which can, for a certain class of problems, *automatically* generate a hierarchy of sub-tasks. The designer is *only* required to provide an ordered set of state variables as discussed in section 5.

This paper will now introduce a simple four room example problem which will be used to help explain the operation of the CQ algorithm. The CQ algorithm will then be demonstrated with the taxi domain used by Dietterich [2] and results compared.

2 The Four Room Problem

Imagine a situation where a robot is started in one of three connected rooms and required to navigate to a fourth room. This example is similar to the one in Hauskrecht, et al. [4] and will be used here to help explain the operation of the CQ algorithm. The four room problem is shown diagrammatically in figure 1.

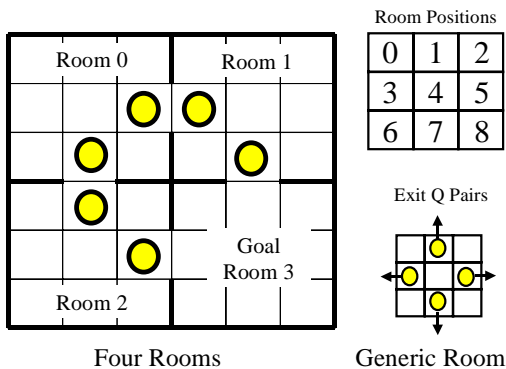


Fig. 1. Four Room Example

Each room is connected via a doorway to the adjacent room. The rooms are labelled 0,1,2 & 3. The nine internal room positions are labelled 0 to 8 as illustrated

in figure 1. The robot is started at random in any position in any one of the rooms 0,1 or 2 and required to learn to move to the goal, which is room 3. The deterministic primitive actions available are moves one step in any of the compass directions. A reward of -1 is administered for each step the robot takes, except when entering the goal room 3, in which case the reward is 20.

The robot does not have a model of the rooms and must therefore learn by observing only its location and the reward after taking an action.

3 Reinforcement Learning

Reinforcement learning addresses situations, such as these, where an autonomous agent uses sensory information to learn to take appropriate actions in an environment in order to achieve its goal.

The reinforcement learning framework is usually modelled as a Markov Decision Process $M: \langle S, A, P, R \rangle$, where: S is a set of finite states; A is a finite set of primitive actions; $P : S \times A \times S \rightarrow [0,1]$ is a probabilistic state transition function; and $R : S \times A \rightarrow \mathfrak{R}$ is a reward function. The agent's objective is to learn an action policy to maximise a measure of expected future reward.

In the four room example, the state is defined by the individual cells in the grid. There are 27 possible states in this problem, 3 rooms with 9 positions in each room. The state could be represented by one variable with 27 values.

This MDP is usually referred to as "flat" in comparison to one structured hierarchically.

4 Hierarchical Approach and Exit Q Pairs

Each of the rooms in the above example present a similar environment to the robot except that the doorways are located in different directions.

Nevertheless, if the reinforcement learner could learn the sub-tasks of exiting a room in any one of the four directions, this skill could be reused for each room and would not have to be re-learned. For example, the skill of leaving room 0 to the east is the same as that required to leave room 2 to the east.

To achieve this, the problem is reformulated. The 27 states are partitioned into three regions. Each room becomes a region. The state is now defined by two state variables, instead of one with 27 values. The room number becomes one variable and the position in each room the second variable. Hence, the overall state is defined by the designer as $s = (s^o, s^l)$ where s^o represents the room number and s^l the location in each room. For example, the middle position in the top left room is (0,4).

In looking for reusable sub-tasks it can be seen that the skill required to navigate inside each room is not dependent on the room number. "Room Navigation" is therefore a good candidate for a reusable sub-task.

There is, however, one problem. The rooms differ in their doorway locations. It may seem that this would prevent reusable sub-tasks to be defined. The way around this difficulty is to identify all the potential ways to exit a room and model them

explicitly. Exit information is extracted so that it can be used when navigating from room to room at a higher level and make the rest of the intra-room navigation common to all rooms. Internal room navigation can now be learnt as a common sub-task.

The exit conditions are identified by finding all potential boundary state/exit action pairs for a generic room. These are the states and actions in a generic room that can potentially lead the agent to leave that room. These boundary state/exit action pairs will be called *exit Q pairs* in this paper. The exit Q pairs for the generic room are shown in figure 1. The circles represent the boundary states for each of the rooms, the arrows the exit actions. The four exit Q pairs are: position 1/move north; position 5/move east; position 7/move south; position 3/move west.

Four sub-MDPs are now created to learn the skill of leading the robot out of a generic room, one for each of the four exit Q pairs. This treatment differs from Hauskrecht et al. [4] in that, in this paper, similar regions (rooms) are treated as *generic* or *aliased*, considerably scaling down the state space.

These sub-MDPs have their state defined by just the position in a room. The room number is irrelevant because internally all rooms have similar reward and transition functions. The actions are the primitive actions and the reward and state transition functions at each step are those defined for the overall problem. The termination condition for each sub-MDP is defined by its associated exit Q pair.

To complete the hierarchical reformulation a higher level or abstract MDP is now defined, but this time using the room number as the state variable. The actions become the 4 sub-MDPs. These actions are called composite or abstract because they do not invoke primitive actions directly, but cause a more complex sequence of primitive actions to be performed as the sub-MDP executes its policy. A composite action works much like a subroutine in a program.

At the abstract level the reinforcement learner is therefore faced with the question of which composite action to use in which room. If the robot is in room 0 and explores the composite action represented by the sub-MDP with termination condition "position 1/move north", it will navigate successfully inside room 0 but hit the wall in the middle of the room on the north side while trying to exit the room. Eventually it will learn to take the composite action which leads it out of room 0 either to by the eastern or southern doorway.

The robot thus learns how to navigate at the higher room level to reach the goal by invoking composite "leaving room" actions.

5 Nested Markov Ordering of State Variables

If the state space of a larger MDP can be represented by two state variables it can be partitioned into regions by using the values of one of the variables to designate each region. In the four room example the room number is used to partition the state space into room regions.

For the following CQ algorithm to be applicable, a designer is required to specify the variable to use for the partitioning so that *all the regions have a similar reward and state transition function*. It is then possible to "navigate" within a region without reference to the value of the variable for that region. In other words the same action policy can be used inside each region knowing that similar state transition and

rewards will produce the same outcome. Regions are thus aliased and can be modelled generically.

This means that an ordering of the state variables is required (supplied by the designer) to exploit and define the reusable sub-tasks within the problem. The order of the two state variables specifies how the overall MDP is to be partitioned. In this paper, when the state is represented by $s = (s^o, s^i)$. The first variable, s^o , will be taken to be the variable whose value identifies the regions and s^i will define the state inside a region.

This ordering of state variables will be referred to as nested Markov because all the regions are defined to have similar Markov properties internally (ie reward and state transition functions). s^i is said to be nested within s^o . The higher level MDP associated with state variable s^o will invoke composite actions made up of sub-MDPs associated with state variable s^i . An analogy is that the sub-MDPs are subroutines of the higher level MDP. They are *nested* in this sense.

This concept can be generalised to any number of levels by considering regions of regions, and so on. For example, if the four rooms were located in a building with multiple floors, where each floor had the same room layout, then a third variable could be introduced to represent the floor number. The floor number would now be chosen as the basis for the first partitioning. Next, the generic floor region is partitioned by the rooms. The overall state would be represented by an ordered Markov set of state variables $s = (s^o, s^i, s^2)$, where the variables are floor number, room number and room location respectively.

Given a nested Markov ordering of state variables, the CQ algorithm generates one hierarchical level per variable and attempts to solve the overall problem along the lines described in section 4 for the four room example.

Interestingly, any problem that can be modelled by n state variables with a nested Markov ordering can generate $2^{n-1}-1$ different hierarchical decompositions (excluding the flat formulation), simply by grouping and combining the variables into various ordered sets. Combining means making one variable out of several that are next to each other. For example the room and location variable can be combined into one variable with 27 values¹.

6 The CQ Algorithm

The CQ hierarchical reinforcement learning algorithm takes a nested Markov ordering of state variable as its input state and outputs primitive actions, as it learns a hierarchical policy. The current CQ algorithm proceeds through two phases, (1) hierarchy construction and (2) solving the total decomposed problem.

¹States in the goal room need not be counted as the problem is restarted at random each time the robot enters the goal room.

6.1 Hierarchy Construction

A MDP with state $s = (s^0, s^1, \dots, s^{i-1}, \dots, s^n)$, where the state variables conform to the nested Markov ordering, will generate one hierarchical level per state variable. It is assumed that any grouping of original state variables has already taken place. The CQ algorithm iterates through all the state variables, starting with the inner most nested MDP corresponding to the variable s^n in s (level n) and concludes with variable s^1 (level 1). At the top level, 0, there is only one MDP to solve the overall task and hierarchy construction is not required.

The steps followed during construction at a level i are:

1. Choose an action a at random at level i .
2. Execute(action a at level $i+1$). Note that action a is composite other than for level n and this function means execute the policy for MDP a at level $i+1$. The Execute function is outlined in the next section.
3. if any of the variables s^0, s^1, \dots, s^{i-1} has changed, add a new MDP at level i with termination condition (ie exit Q pair) a /value of s^i .
4. else update the CQ value function for all MDPs already created at level i .

These four steps are repeated for as many times it is estimated to take to discover all the exit Q pairs.

The CQ algorithm constructs sub-MDPs at each level by using the values of one state variable as the state space. At the lowest level it creates sub-MDPs which uses s^n as the state space.

The algorithm executes a purely random exploration strategy searching for exit Q pairs that are recognised when any higher level variable changes value. Each of these exit Q pairs that can potentially create a region exit condition, generates a separate MDP with the exit Q pair as its termination condition. The number of MDPs generated at each level, therefore, depends on the number of exit Q pairs.

MDPs generated at one level become the available (composite) actions at the next level.

Having generated MDPs for all possible exit Q pairs at the lowest level n , the CQ algorithm repeats this procedure at the next level up, $n-1$, and so on. From now on only composite actions corresponding to the next level down MDPs are used in the random exploration of potential exit conditions for this level.

Learning the value function starts in the construction phase to save time during exploration at higher levels.

CQ employs off-policy [8] updating to learn the value functions for all the MDPs at a particular level simultaneously. This counterbalances, to some degree, the potentially large number of MDPs that may be generated. It should also be noted that solving MDPs during the construction phase will help substantially in speeding up the effort to learn the overall task in the next phase.

6.2 Solving the Hierarchical Problem

Once the hierarchy of MDPs has been constructed the MDP (there is only one) at level 0 is run to solve the overall task by simply calling `Execute(MDP 0, level 0)`. The procedure to `Execute MDP m at level i` follows:

Execute(MDP m at level i)

1. if $i = n+1$ take primitive action m in the domain.
2. else repeat (until RETURN)
 - choose an action a according to the exploration policy for MDP m at level i .
 - `Execute(a at level $i+1$)` & observe state s and reward r .
 - if any of the variables s^0, s^1, \dots, s^{i-1} has changed or if MDP m has terminated, RETURN.
 - Update CQ values for all MDPs at level i . Note that this update actually uses the number of steps taken by levels below to discount future value.

The CQ algorithm defines a CQ function along similar lines to Dietterich's [2] completion function but differs in one very important respect. The values stored to represent the value function are $CQ(i,m,s,a)$ and defined as the expected future reward at level i of executing the current MDP m after taking action a in state s , but *including* the *primitive reward* generated after completing the action. The inclusion of primitive reward automatically assigns credit hierarchically. The CQ algorithm is named after this function which is a type of hybrid completion and Q function. At the lowest level the CQ function is in fact the standard Q function, because the immediate reward for a primitive actions is included in the CQ value. The CQ function for an exit Q pair is zero.

In the four rooms, for example, after finding a hierarchical solution, the value of the middle position in room 0, that is $s=(0,4)$, is 16. This is composed of a value of -1 representing the level-1 CQ value of position 4 for leaving the room either to the south or east and a level-0 CQ value of 17 for exiting from either boundary state. The CQ value of 17 represents a reward of -1 for exiting room 0, -2 for navigating to the entry of goal room 3 and 20 for entering room 3. (An episodic or undiscounted value function is used here)

7 Simple Taxi Task

Dietterich [2] used a simple taxi task to illustrate the MAXQ algorithm. The same example will be used here to demonstrate how the CQ algorithm generates hierarchical structure and then solves the overall task. The taxi domain description is reproduced here for the reader's convenience.

Figure 2 shows Dietterich's 5-by-5 grid in which a taxi can move in any one of the four compass directions at each step. The overall objective of the task is for the taxi to pick up a passenger from one of the four specially designated locations (Red, Green,

Yellow or Blue) and drop the passenger off at one of these designated locations, at which point the task (or trial) terminates. The taxi starting position can be any one of the 25 grid locations.

The taxi starting position, passenger pick-up location and destination location are chosen at random for each trial. There are six deterministic primitive actions, namely, move north, move east, move south, move west, pick-up passenger and put-down passenger. Each action incurs a reward of -1. If the taxi moves into a wall at the boundary of the grid or at a barrier between the cells, its position remains the same, but it still receives a reward of -1. If it attempts a pick-up action anywhere other than with the passenger waiting at the pickup point or a put-down action without the passenger in the taxi and the taxi located at the destination point, it receives a reward of -10. If the taxi delivers the passenger successfully it receives a reward of 20 and this completes one trial.

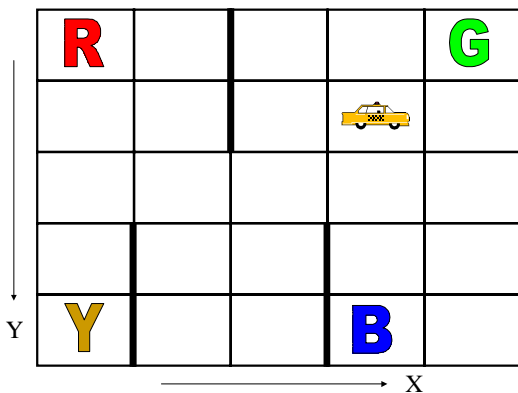


Fig. 2. The Taxi Domain

The state variables for this task are the co-ordinates of the taxi location in the grid, the location of the passenger and the location of the pickup point. The total number of states, $|S|$, is 500: 4 possible destinations, 5 possible passenger locations (4 for the designated locations and in the taxi itself), 5 for the x position and 5 for the y position.

8 CQ Hierarchical Decomposition

The 4 state variables can be arranged into a nested Markov order, namely: destination location, passenger location, x-grid position, y-grid position. The order of x and y is dictated by the fact that there are no barriers between cells moving north or south. The vertical regions, defined by x, have identical intra-region transition and reward functions. The four variables can generate $2^{4-1} = 7$ different hierarchical decompositions by grouping and combining them as discussed. Three of these seven

state representations were tried separately with the CQ algorithm and results compared to flat Q and MAXQ.

Three different state representations for the taxi task were:

Case 1: Two State Variables, $S=(\text{Destination \& Passenger, XY Grid Position})$

Case 2: Three state variables, $S=(\text{Destination, Passenger, XY Grid Position})$

Case 3: Four state variables, $S=(\text{Destination, Passenger, X, Y})$

In case 2, for example, the three variables generate a three level hierarchy, as shown in figure 3.

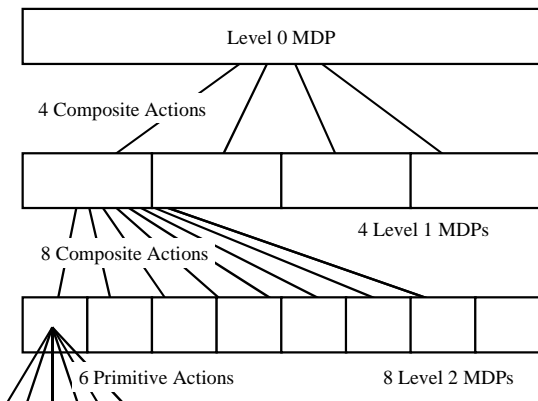


Fig. 3. Hierarchies of MDPs generated by the CQ algorithm for Case 2 in the simple taxi task

Eight exit Q pairs are generated by the CQ algorithm at the lowest level. These are the four specially designated locations together with a pick-up or put-down action. The middle level then uses these 8 composite actions to generate 4 possible exit Q pairs at this level. These are the boundary state where the passenger is in the taxi and the four composite actions that can navigate to a special location and perform a put-down action.

Results comparing the performance of the three different state value representations, MAXQ and flat learning are show in figure 4. In each case CQ learns in fewer trials than the flat Q reinforcement learner but not as well as MAXQ. Of course, handcrafting a MAXQ decomposition means that the designer has provided additional background knowledge which is not provided to CQ. For example, with MAXQ, the primitive actions to pick-up and put-down the passenger are not specified for the navigation task because the designer knows a priori they are not required. The CQ algorithm is not given this information and needs to learn this for itself.

CQ also solves the more complex “Fuel Taxi Problem” problem [2] with similar performance characteristics compared to MAXQ and flat Q reinforcement learning. In this problem the taxi uses 1 unit of fuel for each step taken and can decide to refuel at a filling station to complete its mission. If the taxi runs out of fuel it receives a reward of -20. An extra Fillup action is introduced which fills the fuel tank to 12 units of fuel

at the filling station. The taxi is started with a random fuel level between 5 and 12. This problem is more complex with 7000 states and 7 actions.

Employing MAXQ requires the designer to assign credit hierarchically by decomposing the primitive reward function manually amongst the various sub-tasks. CQ's value function specification and nested Markov ordering of state variables ensures that credit is assigned automatically amongst the hierarchical levels.

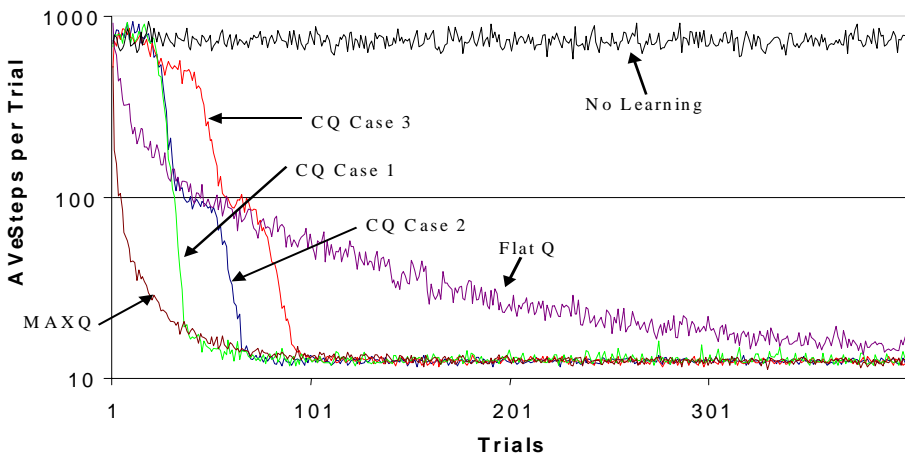


Fig. 4. Performance comparison between Flat RL, MAXQ and the three CQ Algorithm hierarchies for the simple taxi task

9 Limitations and Extensions

One limitation is the potentially large number of sub-task Markov decision problems that are generated, one for each exit Q pair of a region. In a stochastic situation or where there is a large interface between regions, this could become problematic. Further research will explore ways in which exit Q pairs can be grouped into common classes with only one sub-MPD for each class.

The application to continuous action and state spaces and to hidden state is left to further research.

A shortcoming of course is that the CQ algorithm cannot be said to *learn* the hierarchy, because a designer is still required to specify the special form of the state variables. Further research is intended to find ways of bridging the gap between an arbitrarily designated sensor and a nested Markov ordering of state variables.

10 Conclusion

For MDPs for which sub-tasks can be identified via a set of state variables, the CQ algorithm can solve the problem by automatically generating a set of sub-MDPs. The CQ algorithm does *not* require a designer to specify sub-task termination conditions, allowable actions, state abstractions, pseudo rewards or any hierarchical credit assignment.

References

1. Dean, T., Lin, S-H.: Decomposition Techniques for Planning in Stochastic Domains. (Technical Report CS-95-10). Department of Computer Science, Brown University, Providence, RI (1995)
2. Dietterich, T. G.: Hierarchical Reinforcement Learning with MAXQ Value Function Decomposition. Department of Computer Science, Oregon State University, Corvallis, OR (1999)
3. Digney, B. L.: Emergent Hierarchical Control Structures: Learning Reactive / Hierarchical Relationships in Reinforcement Environments. In: Maes, P., et al (eds.): From animals to animats 4: Proceedings of the fourth international conference on simulation of adaptive behaviour, MIT Press, Cambridge(MA) London (1996) 363-372
4. Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T., Boutilier, C.: Hierarchical Solution of Markov Decision Processes using Macro-actions. (Technical Report). Department of Computer Science, Brown University, Providence, RI (1998)
5. Parr, R. E.: Hierarchical Control and Learning for Markov Decision Processes. Doctoral dissertation, Computer Science, University of California, Berkley (1998)
6. Parr, R, Russell, S.: Reinforcement Learning with Hierarchies of Machines, Advances in Neural Information Processing Systems 10. MIT Press (1998)
7. Singh S.: Reinforcement Learning with a Hierarchy of Abstract Models. Proceedings of the Tenth National Conference on Artificial Intelligence, Menlo Park: AAAI Press (1992)
8. Sutton, S., Barto, A. G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
9. Sutton, R. S., Singh, S., Precup, D., Ravindran, B.: Improved switching among temporally abstract actions. Advances in Neural Information Processing Systems 11 (Proceedings of the 1998 conference), MIT Press (1999) 1066-1072
10. Sutton, R. S., Precup, D., Singh, S.: Between MDPs and Semi-MDPs: Learning, Planning, and Representating Knowledge at Multiple Temporal Scales. (Technical Report) Department of Computer and Information Sciences, University of Massachusetts, Amherst, MA (1998)
11. Thrun, S., O'Sullivan, J.: Discovering Structure in Multiple Learning Tasks: The TC Algorithm. Proceedings of the Thirteenth International Conference on Machine Learning. Morgan Kaufmann, San Mateo (1996)
12. Thrun, S., Schwartz, A.: Finding Structure in Reinforcement Learning. Advances in Neural Information Processing Systems 7, Morgan Kaufmann, San Mateo (1995)