Using abstract models of behaviors to automatically generate reinforcement learning hierarchies

Hauptseminar Intelligente Autonome Systeme (WiSe 04/05)

Christian Sosnowski

Betreuer: Freek Stulp
Abgabetermin: 22. November 2004
Contents

1 Introduction 2

2 The example 3

3 Fundamentals 4
   3.1 Markov Decision Process .............................................. 4
   3.2 Q-Learning ................................................................. 4
   3.3 The Curse of dimensionality ............................................. 6

4 Building Task Hierarchies through planning 7
   4.1 Building the plan .......................................................... 7
   4.2 Combining Planning and Learning ....................................... 8

5 The P-HSMQ Algorithm 10
   5.1 Pseudo code of P-HSMQ .................................................. 10
   5.2 Experiment 1 .................................................................... 11

6 Termination Improvement 12
   6.1 Teleo-Reactive Q-learning (TRQ) ....................................... 13
   6.2 Experiment 2 .................................................................... 14

7 Conclusion 15

References 16
1 Introduction

A major problem for an intelligent autonomous system is the learning process, especially reinforced learning in an efficient way (i.e. in a timely acceptable frame). Scaling up general-purpose solutions is nearly irresolvable. Increasing the number of actions per state means the number of possibilities, which have to be explored, will raise exponential (curse of dimensionality).

This leads to the approach of concentrating on special-purpose solutions by providing background knowledge to improve the learning process. One possible way is the hierarchical reinforcement learning (HRL) which proves to be quite effective. This means, the whole task will be divided into numerous simple subtasks. If the agent solves all subtasks, the whole task will be fulfilled. One of the most used approaches is to introduce temporally abstract actions or behaviors with a limited scope (or applicability space) and localized goals. Policies to choose between the possible actions are learnt in regard of these abstract behaviors and not orientated directly on the primitive actions. Providing abstract behaviors is not yet enough to limit the horizon of the learning process in an efficient way. If the agent can choose between many behaviors where only a few lead to the goal, the scope of possibilities might be as big as for the solution with no behaviors. Since behaviors are designed with a purpose, this purpose leads to an abstract model determining for particular states, which behaviors are appropriate and which are not. This saves the agent to explore inappropriate behaviors.

Most existing algorithms achieve this by including some kind of task hierarchies which structure the agent’s decision process, limiting the possible choices to those which might lead to the goal. Basically the agent uses a function which maps the current state to a set of appropriate behaviors. Up to now, the development of the abstract models has to be done manually by the trainer. It is obvious, that with increasing complexity, this becomes more and more difficult.

In the following chapters one possible approach will be demonstrated. The appropriate behaviors will be embedded into a plan. Behaviors will be represented by planning operators. Symbolic descriptions will be used to allow the trainer to specify behaviors in a high-level language.

Based on the provided plan, the agent can automatically build up task-hierarchies through planning. This combination of planning and learning uses the advantages of both sides (background knowledge to limit the policies through planning and reinforcement planning to build up concrete policies for behaviors and optimize choices in the plan).

This paper describes two algorithms introduced by Malcolm R. K. Ryan [Rya02b], demonstrating the advantages of the plan based on reinforcement learning. Both algorithms depend on a Markov-Decision Process and use Q-Learning.
2 The example

The following example will be used to evaluate the provided algorithms. It is based on a grid-world as outlined in Figure 1.

![Figure 1: The grid world](image)

In this case, the learning agent is a household robot in a house with a layout as shown above. The task is to start in the study room (as depicted in the grid), to get the coffee from the kitchen and the book from the Bedroom2 and to carry them to the lounge (for the first experiment, the bump at the exit of the dining room will be ignored).

The robot is able to know its location with the precision of the shown cells of the grid. Its primitive actions are to move from one cell to one of the eight neighbor cells (with a small probability of error). If the robot reaches an object (i.e. the coffee or the book) it can pick it up and carry it from then on. The problem is now to learn the best policy to achieve this task.
3  Fundamentals

What are the possibilities for the robot in the example of chapter 2 to learn its path to achieve the goal? One of the most advanced options in reinforcement learning is Q-learning embedded into a Markov Decision Process.

3.1  Markov Decision Process

As already outlined in the introduction, an algorithm has to guarantee performance at least up to a certain point. Therefore, some theoretical restrictions have to be made onto the model. One restriction is the amount of information necessary which determines the agent’s action. Actions and their results could depend on information hidden from the agent. This could have unpredictable outcomes on the action which complicates the process of learning. To avoid this problem, most reinforcement learning algorithms make a strong assumption about the structure of the environment. They assume that it operates as a Markov Decision Process (MDP). A MDP describes a process that has no hidden state or dependence on history. That means the outcome of an action purely depends on the current state and the performed action. Furthermore if there are any probabilities involved, they are strictly independent with the exception of the current state and the performed action.

3.2  Q-Learning

Q-Learning is an online incremental learning algorithm which improves the policy for a given MDP step by step. With other words, the policy, which influences the primitive action in a certain stage, will be optimized with every step. In this context there will be only a basic explanation of the Q-learning algorithm and the involved equations. The idea is to use a table to store the different Q-values depending on the state \( s_t \) and the action \( a \). For each step, the Q-value will be updated with the following formula:

\[
Q(s_t, a) \leftarrow (1 - \alpha)Q(s_t, a) + \alpha(r_t + \gamma \max_{a \in A} Q(s_{t+1}, a))
\]

\( \alpha \) is the learning rate \((0 \leq \alpha \leq 1)\), \( \gamma \) is the discount factor \((0 \leq \alpha \leq 1)\), \( r_t \) is the real reward and \( \max_{a \in A} Q(s_{t+1}, a) \) is the maximum Q-value for all explored actions at state \( s_{t+1} \). What does the formula mean? With \( \alpha = 0 \), there will be no more learning, the Q-value will remain the same since the second part of the formula will be 0. If \( \alpha \) is greater than 0, the former Q-value will be updated only partially with the old value (first part of the formula) and a correction values derived from the best Q-value of state \( s_{t+1} \) reduced by \( \alpha \) and the discount factor \( \gamma \). To demonstrate the functionality of Q-learning, a small
example will be provided. The start condition is a rectangle of 3x4 positions with the robot in the lower left corner (A4) and the goal is to reach to upper right position (C1). On the final position the reward value $r_t$ is 1, on all other positions 0. The learning factor $\alpha$ is 0.1 and the discount factor is 0.9.

Start configuration

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Now different random paths will be shown which update the Q-values at the respective positions:
Path = A4-A3-A2-B2-C2-C1

In this case only the value at pos C2 will be updated by:
$C2 = 0.9 \times 0 + 0.1 \times (0 + 0.9 \times 1) = 0.0900$

Path = A4-B4-C4-B3-A3-B2-C1
The following value will be updated:
\[ B_2 = 0.9 \times 0 + 0.1 \times (0 + 0.9 \times 1) = 0.09 \]

Path = A4-A3-B3-B2-C1

\[
\begin{array}{ccc}
A & B & C \\
\hline
1 & 0 & 0 \\
2 & 0 & 0.1710 \\
3 & 0 & 0.0081 \\
4 & 0 & 0 \\
\end{array}
\]

The following values will be updated:
\[ B_3 = 0.9 \times 0 + 0.1 \times (0 + 0.9 \times 0.09) = 0.0081 \]
\[ B_2 = 0.9 \times 0.09 + 0.1 \times (0 + 0.9 \times 1) = 0.1710 \]

The Q-learning algorithm is proven to converge to the optimal values. In this case the Q-values in the final grid will look as follows:

\[
\begin{array}{ccc}
A & B & C \\
\hline
1 & 0.81 & 0.9 \\
2 & 0.81 & 0.9 \\
3 & 0.81 & 0.81 \\
4 & 0.729 & 0.729 \\
\end{array}
\]

### 3.3 The Curse of dimensionality

In the primitive example of 3.2, the algorithm of Q-learning converges nicely to the optimal solution. In practice it has a very poor performance. To learn the optimal policy, at least every state-action pair has to be visited. For small problems like this 4 x 3 rectangle the solution is found in a reasonable time. But scaling up to real world problems with multi-dimensional state problems the number of state-action pairs will raise exponentially.

To demonstrate the problem, an example out of the real world will be chosen. Imagine an autopilot in a car which has to bring the car from position A to position B. If there would
be only the different streets, the autopilot would just calculate the best route and needs only some kind of a guidance and recognition system which determines the exact position on the street and then the car could drive to position B. But the world is more complex. Just adding traffic lights into the system, depending on the distance to the traffic light, the auto pilot would have a lot of options (i.e. increase speed to get over the crossing, just continue, stop or take a different route). Then adding for instance other cars the complexity would increase enormous. Especially, the MDP will be violated.

4 Building Task Hierarchies through planning

As seen in chapter 3, reinforcement learning only with the methods of Q-learning is highly inefficient. For the time being there exists no general-purpose solution to solve this problem. But if a problem cannot be solved in the full extend, one possible solution is to divide the problem into smaller subtasks and try to solve these subtasks one after the other. The solution used by Malcolm R. K. Ryan [Rya02b] is to define behaviors which describe basic subtask and then putting the subtasks together to a plan. Now the subtasks could be solved easily by Q-learning, still avoiding the problem of the curse of dimensionality.

For the example of chapter 2, only two different behaviors will be used. The first one is Go(Room1, Room2) which means, the aim is to move from one room to another room. The second behavior is Get(Object, Room), which means obviously that the robot picks up the specified object in the room. As mentioned, the behavior has to be learned by the robot with the means of recursively optimal reinforcement learning.

4.1 Building the plan

To build the task hierarchies (the plan), for convenience purposes the model will be expressed in the high level language of symbolic planning. States, actions and goals are all described symbolically using logic expressions. For the example of chapter 2, only two expression will be used. The first is loc(Object, location), which is true when the Object (i.e. the coffee or the robot) is at the specified location (i.e. a room or even the robot itself). The second expression is door(Room1, Room2), which is true if there is a door between Room1 and Room2.

Using this language, the starting state of the robot will be described as follows:

\[ s = \text{loc}(\text{robot}, \text{study}) \land \text{loc}(\text{coffee}, \text{kitchen}) \land \text{loc}(\text{book}, \text{bedroom}2) \]  

and the goal would be:

\[ G = \text{loc}(\text{robot}, \text{lounge}) \land \text{loc}(\text{coffee}, \text{robot}) \land \text{loc}(\text{book}, \text{robot}) \]
The above mentioned behaviors can now as well be described in this language as so called teleo-operators. Teleo-operators define the effect of a goal-directed behavior \( B \) with a pre- and a postcondition (\( B\.pre \) and \( B\.post \)). They are expressed by conjunctions of states as shown in equation 2. If \( B \) is initiated in the state satisfying \( B\.pre \), then the agent will stay in that condition until it achieves its postcondition \( B\.post \).

Now the \( \text{Go()} \) and \( \text{Get()} \) behaviors can be expressed as follows:

\[
\begin{align*}
\text{Go(Room1, Room2)} \\
\text{Pre: } & \text{loc(robot, Room1)} \land \text{door(Room1, Room2)} \\
\text{Post: } & \text{loc(robot, Room2)}
\end{align*}
\]

\[
\begin{align*}
\text{Get(Object, Room)} \\
\text{Pre: } & \text{loc(Object, Room)} \land \text{dloc(robot, Room)} \\
\text{Post: } & \text{loc(Object, robot)}
\end{align*}
\]

Now the plan can be build which sequences the behaviors to achieve the overall goal. An important item is that the plans have to be complete. This means, not only one path (maybe the shortest path) has to be implemented, but all paths which lead to the goal without a loop. The reinforcement learner can then choose between the different paths. Figure 2 shows a partial plan to solve the problem. The plan is represented in a tree, the states are the nodes and the edges show the actions. For one state there can be more than one active state (node). In this case, the starting state is shown. The active nodes are contrast inverted. Since there are two active nodes, there are presently two possible ways to achieve the goal (first getting the coffee and then the book and vice versa).

### 4.2 Combining Planning and Learning

Now since we have a plan, we have to combine it with the reinforcement learning. This will be done by using the former introduced teleo-operators together with a reward function. The aim should be to reach the postcondition as soon as possible. If the agent executes action \( a \) in the state \( s \), which results in a transition to state \( s' \), the local reward for the behavior \( B \) will be:

\[
B\.r(s, a, s') = \begin{cases} 
1 & \text{if } s' \models B\.post \\
-1 & \text{if } s' \not\models B\.post \text{ and } s' \not\models B\.pre \\
0 & \text{otherwise}
\end{cases}
\]  \( (4) \)

This means, as long as the agent stays in a certain stage (i.e. just moving inside of a room and not picking up anything) the reward will be 0. If the agent would leave the state not according to the plan the reward is -1 (i.e. moving into a room not dictated by the plan) and if the agent acts according to the plan the reward will be 1. Obviously, in this example, the agent has to do a lot of primitive actions which will be rewarded with
0 since moving from one room to another takes several steps. This will be solved with the algorithm of Q-Learning as described in the chapter 3.

Like for the local behavior, the achievement of the overall aim can be expressed as follows:

\[
    r(s, a, s') = \begin{cases} 
    1 & \text{if } s' \models G \\
    0 & \text{otherwise} 
    \end{cases} 
\] (5)
5 The P-HSMQ Algorithm

As a result of chapter 3 and 4, Malcolm R. K. Ryan [Rya02b] came up with the Planned-Hierarchical Semi Markov Q-learning (P-HSMQ) algorithm.

5.1 Pseudo code of P-HSMQ

For simplicity the exact algorithm will be modified to show only the important parts. For details please refer to [Rya02a] or [Rya02b].

Table 1: P-HSMQ Algorithm

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>According to the plan all behaviors, which correlate to the current state and could lead to the goal, will be chosen.</td>
</tr>
<tr>
<td>2.</td>
<td>According to the exploration policy (this can be the behavior with the highest Q-value or, if still learning, a different, maybe random behavior out of the pool of</td>
</tr>
</tbody>
</table>
active behaviors), one behavior will be selected.

3. This is one of the most crucial points in the algorithm. Not like in the normal Q-learning, the Q-values will be learned depending on the current behavior. This means for one position, there can be more than one Q-value. One state/position will be weighted different for different behaviors.

4. k and the totalReward will be set to 0 since the totalReward will be calculated for each execution of a behavior.

5. If the behavior reaches the global goal, the global reward Function value will be weighted with the number of step until it reached this goal.

6. Similar to the Q-values of the primitive actions, the whole behavior will be rewarded with a Q-value. This allows influencing the choice according to the exploration policy.

7. Again like in note 1, all the presently active behaviors will be selected.

5.2 Experiment 1

This P-HSMQ algorithm will now be evaluated with the example from chapter 2 against nearly the same algorithm, but with all behaviors available for the whole time (no plan involved), which is basically global Q-learning as described in chapter 3. For comparison, an additional algorithm with a plan but without HRL (not described more in detail) will be added. The learning parameters are set as follows:

- $\alpha = 0.1$
- $\gamma = 0.95$

The policy dictates the agent with a 10% probability to choose an exploration path (and not the already calculated best path). Each approach was run twenty times, with each run consisting of 1000 consecutive trials. The result was as predicted. The P-HSMQ algorithm converged to the optimal path length with about half of the number of trials (see figure 3).

More precisely, measuring the average number of experiences required before trial length falls below 500, the HSMQ with all behaviors took 72324 primitive actions, the P-HSMQ took only 35151. The reason is obvious. Without the plan a lot of the early trials were used learning behaviors not suitable to reach the final goal. Also the long term performance of the HSMQ with all behaviors was poorer since it explores still behaviors which do not contribute to the goal.
6 Termination Improvement

While the P-HSMQ algorithm is effective, it does not make full use of the information available to it in the plan it builds. It only checks the appropriateness of a behavior when it is initiated, and then always executes it until completion, ignoring effects that might cause the action to be no longer appropriate. The problem will now be demonstrated with a small modification of the example in chapter 2. A bump will be added at the exit of the dining room. With a probability of 10% the coffee will be spilled when the robot carries the coffee over the bump. Using the P-HSMQ algorithm, the robot will finish the current behavior, in this case moving to the exit of the room until the spilled coffee is noticed and only then the robot is able to return to the kitchen to get a new cup of coffee. This could be improved if the robot would turn around and move back to the kitchen right after spilling the coffee.

The solution is obvious. The current behavior has to be stopped as soon as it makes no sense any more. An interruption criterion for each behavior has to be introduced. There is only a small problem. If there are two different nodes of the plan dictating the same behavior with different interruption criteria, the outcome of the executed behavior will depend on the selected node. To maintain the MDP not a behavior but a node of the plan with the associated behavior has to be selected to be executed in the inner loop of the algorithm. For the same reason a behavior should learn the Q-values independent from the execution context. This means, the behavior must be fully allowed to explore its application spaces. In particular, as the reward functions used for the example of chapter 2 only provide non-zero rewards when the behavior terminates, a certain percentage of the executed behaviors must be allowed to proceed until they terminate as planned.
6.1 Teleo-Reactive Q-learning (TRQ)

The TRQ is a modification of the P-HSMQ algorithm. It is complying with the above mentioned requirements. As P-HSMQ assigns Q-values to the behaviors, TRQ executes the behaviors of a selected node and assigns the Q-value to the node. To find the optimal path, the node with the best Q-value has to be selected.

Like for the P-HSMQ algorithm only the more important parts of the TRQ algorithm will be shown.

Function TRQ
Get a plan
Get all active Nodes
While goal is not reached
    Select one node N from all active nodes according to an exploration policy
    While the selected node is still active
        Choose primitive action at
        Execute the primitive action at
        \[ B.Q(s_t, a_t) \leftarrow B.r(s_t, a_t, s_{t+1}) + \gamma \max_{a \in B.A} B.Q(s_{t+1}, a) \]  
    Get all active Nodes
End while
k = 0
totalReward = 0
for each step done
    totalReward \leftarrow totalReward + \gamma^k r(s, a, s')
    k \leftarrow k + 1
end for
\[ Q(s_T, N) \leftarrow totalReward + \gamma^k \max_{N' \in N} Q(s_{T+k}, N') \]
If behavior B of current node N not terminated
    With a probability \( \eta \) do
        PERSIST(B)
End with
End while
Return chosen path
End TRQ

Table 2: TRQ Algorithm

Explanations to the algorithm:

1. According to the plan all active node complying with the current state, which lead to the goal, will be chosen.
2. According to the exploration policy, one node will be selected.

3. Checking if a node is still active allows interrupting a behavior. If the state changes while executing a behavior, the current node will not any longer be part of the active, which results in interrupting the "while" loop.

4. Absolutely similar to the P-HSMQ algorithm, the Q-values will be calculated while executing the behavior B of the current node P.

5. The big difference to the P-HSMQ is this line. Inside the inner while loop, the pool of active nodes will be recalculated to allow the interruption if necessary.

6. k and the totalReward will be set to 0 and recalculated like in the P-HSMQ.

7. Similar to the calculation of Q-values for the behaviors in the P-HSMQ, the Q-values will be calculated for the current node N.

8. As described earlier, with a certain probability, the currently executed behavior has to be allows to finish. The function PERSIST(B) does exactly the same like the P-HSMQ and explores the current behavior until it terminates normally.

### 6.2 Experiment 2

Now the P-HSMQ and the TRQ algorithm will be compared against each other. As described earlier in this chapter, the example of chapter 2 together with the bump will be used. The learning and exploration parameters will be as for the experiment 1. The probability factor for $\eta$ is set to 0.1, the probability to spill the coffee is as well set to 0.1. Twenty independent runs were performed for each approach. Each run consisted of 1000 learning trials. The convergence times for both approaches are comparable. P-HSMQ took on average 55911 experiences to produce a trial length of less than 500 steps, TRQ took 45657 experiences. But dividing the results into those with spill and without spill and looking at the average trial length and standard deviation for the final learnt policy of both algorithms (see table 6.1), there are significant differences.

<table>
<thead>
<tr>
<th></th>
<th>P-HSMQ</th>
<th>TRQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spill</td>
<td>472.25 ± 33.68</td>
<td>347.01 ± 38.26</td>
</tr>
<tr>
<td>No Spill</td>
<td>296.61 ± 23.54</td>
<td>283.44 ± 37.67</td>
</tr>
<tr>
<td>Difference</td>
<td>175.64 ± 23.96</td>
<td>63.57 ± 22.82</td>
</tr>
</tbody>
</table>

Table 3: average trial length and standard deviation

As expected, with no spill, the result is similar, but the P-HSMQ algorithm shows a significant higher trial length where a spill occurred. This demonstrates how termination improvement, applied appropriately according to the plan, can result in more efficient policies.
7 Conclusion

Reinforcement learning is one of the most important tasks an autonomous intelligent system has to accomplish. Q-learning provides one basic mean to solve this problem. Although, in theory, Q-learning converges nicely to the optimal solution, the primitive example of chapter 2 shows already the curse of dimensionality. In real problems the performance of Q-learning is not acceptable.

One possible way to solve this problem is to use abstract models of behavior to automatically generate reinforcement learning hierarchies or with other words to use a plan consisting of primitive behaviors and let the system learn how to solve the primitive behaviors and then how to use the optimal part of the plan. The experiments in chapter 5 and 6 demonstrate precisely the increase in performance.

There is still a lot of room for improvements. Future algorithms could maybe analyze what went wrong if a plan failed or could even invent new behaviors on their own to fit the circumstances that arise.
References
