Hierarchical Assignment of Behaviours to Subpolicies

Wilco Moerman[1], Bram Bakker[2] & Marco Wiering[3]
[1]Cognitive Artificial Intelligence, Utrecht University, wilco.moerman@gmail.com
[2]Intelligent Autonomous Systems Group, University of Amsterdam, bram@science.uva.nl
[3]Intelligent Systems Group, Utrecht University, marco@cs.uu.nl

Abstract

Task decompositions are central in Hierarchical Reinforcement Learning, but in most approaches they need to be *designed* a priori, and the agent only needs to fill in the details in the fixed structure. In contrast, the algorithm presented here autonomously identifies behaviours in an *abstract* higher level state space. Subpolicies self-organise to specialize for the high level behaviours that are identified.

1 Introduction

The use of hierarchies in Reinforcement Learning (RL) is one of the strategies for dealing with large state spaces. The idea is to improve normal, flat Reinforcement Learning by giving it the possibility to execute actions that are *temporally extended.* The two most common ways to achieve this are *the introduction of multiple layers* (prime example: MAXQ [1]) and the use of the *options* framework [2].

In the options framework, extended actions (options) are added to the flat Reinforcement Learning algorithm, directly augmenting the action space. This allows taking larger steps, but does not really introduce layers of abstraction.

Introducing layers, on the other hand, allows for the use of more abstract representations or states (although not every layered approach uses abstractions). For approaches like MAXQ, the designer needs to define a task decomposition, determining which task is done by which (sub)policy. The agent only needs to fill in the values in the value functions, because the structure of the task, and which subtasks are done by which policies, is already largely fixed, and only the (sub)policies need to be learned.

The algorithm presented here takes a different approach. Instead of thinking in terms of detailed task decompositions, a suitable geometric *abstract state space* (a higher level representation of the normal state space) is used for the higher level(s), and subpolicies self-organise to cover (i.e. specialize for) the needed behaviours identified in an abstract *Behavior Space.*

2 Behaviour Space and Abstract State Space

Our method is based on having/identifying an abstract, high-level, geometric state space which captures important properties of the underlying task and which is used for taking higher-level actions to be executed by specialist lower-level subpolicies.

For such an abstract state space (or any state space, for that matter) we define a *Behaviour Space* as the set of all possible difference vectors in that state space (see fig. 1). This means that the *Behaviour Space* consists of all possible vectors that are confined within the dimensions of the state space. The actions that actually occur in the state space (because they are transitions from one state to another) form a subset of all possible behaviour vectors.

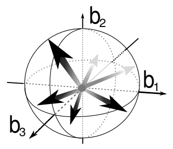


***Figure 1: Behaviour Space**: the space of all possible difference vectors in a state space (with dimensions b_1, b_2, b_3).*

A suitable *abstract representation* of the problem space has the following properties: states that are close together in the original state space need to be mapped to abstract states near each other (or the same), and neighbouring abstract states need to be mapped to states in the state space that are near each other (fig. 2). Also, a translation (difference vector) in the abstract state space should correspond to a meaningful change in the original state space. Furthermore, a *useful* abstract state space needs to be significantly smaller than the original state space. Finally, the actually occuring transitions between states in the abstract state space need to be distributed *non-uniformly* in the abstract Behaviour Space.

The last requirement is needed to ensure that many transitions are roughly the same, meaning there are "pockets" or "clusters" of actual transitions in the Behaviour Space.

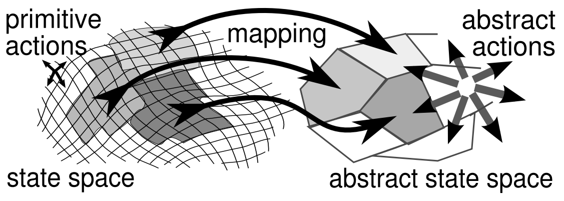


***Figure 2: mapping** of a state space to an abstract state space. The small arrows on the left are primitive actions, the arrows on the right depict actions that can be taken in the abstract space.*

Creating such an abstract space is easiest when the environment has a certain *inherent geometry.* This is the case, e.g., in spatial navigation tasks; but this work is definitely not constrained to pure navigation tasks, as we will demonstrate in the experiments.

3 HABS

HABS (Hierarchical Assignment of Behaviours to Subpolicies) has one high level policy that controls a limited number of low level *subpolicies.* All these policies learn with standard Reinforcement Learning techniques like Q-Learning or Sarsa [3].

HABS uses the original state space and the primitive actions for the subpolicies (as in normal Reinforcement Learning). For the higher level policy it uses a suitable abstract state space and the *subpolicies* are used as its higher level actions.

These subpolicies perform behaviours that are vectors in the *abstract* Behaviour Space defined by the *abstract* state space. They are temporally (and spatially) extended in the original, normal state space. In this way the relation *abstract state space* $\rightsquigarrow$ *abstract Behaviour Space* $\rightsquigarrow$ *behaviours* mirrors that of *state space* $\rightsquigarrow$ *behaviour space* $\rightsquigarrow$ *primitive actions.*

Because a suitable abstract state space has its actually occuring transitions distributed *non-uniformly*, there are pockets or clusters of behaviours that are roughly the same, meaning they can be performed by one subpolicy. These dense areas in the abstract Behaviour Space allow HABS to learn to distribute its subpolicies to cover most – and if enough subpolicies are available: *all* – of the needed high level actions. This mimics the way primitive actions are applicable everywhere in the state space: a primitive action like *"go left"* matches with transitions between every state and its left neighbour, so only one action *"go left"* is needed.

The high level policy selects one of its actions, which means one of its subpolicies takes control until a new (i.e. adjacent) abstract state is reached or timeout occurs. After that, control is returned and a new subpolicy is selected by the high level policy. This mirrors the way primitive actions are used: they terminate in an adjacent state or where they started (because the action failed).

The high level policy gets high level immediate rewards which correspond to the real, original rewards accumulated between one high level state and the next high level state. The update rule for the high level policy is:

$$\begin{aligned} Q_h(hs_t, b_t) \quad &\leftarrow \quad (1-\alpha)\, Q_h(hs_t, b_t) \quad + \\ &\alpha\, (hr_{t+\Delta t} + \gamma^{\Delta t} \max_b Q_h(hs_{t+\Delta t}, b)) \end{aligned}$$

where the Q_h-value denotes the value-function for selecting subpolicy b in high level state hs, where the duration of the behavior is denoted as Δt and the high level reward is denoted hr. Note that the discounting of the value of the next high level state is similar to what happens in Semi-Markov Decision Processes [1, 2].

The subpolicies need to be rewarded for good behaviour. To accomplish this, a *characteristic behaviour* vector[1] is assigned to each subpolicy. This characteristic behaviour represents the kind of behaviour that the subpolicy should accomplish, and it acts as a measure for the training of a subpolicy. If the subpolicy accomplishes a behaviour that is closest to its own characteristic behaviour, it is rewarded, but if it acted more like the characteristic behaviour of another subpolicy or if the subpolicy didn't get the agent out of the abstract state at all, it is punished. At all other times during execution of the subpolicy, the reward is 0. Updating is done using Q-Learning:

$$\begin{aligned} Q_l(s_t, a_t) \quad &\leftarrow \quad (1-\alpha)\, Q_l(s_t, a_t) \quad + \\ &\alpha\, (r_{t+1} + \gamma \max_a Q_l(s_{t+1}, a)) \end{aligned}$$

When the characteristic behaviour of *another* subpolicy is the closest match for the actually experienced behaviour, the selected action *at the high level* can be replaced by the action that was the closest match. After that the Q_h-value can be updated as usual. This allows for efficient use of all data and speeds up learning.

The characteristic behaviour vector (*char*) is moved towards the actually executed behaviour vector ($act_{t\rightarrow t+\Delta t}$) if and only if the accomplished behaviour is closer to its own characteristic behaviour vector than to those of all other subpolicies. This update can be done using the following simple rule:

$$char_{t+\Delta t} \leftarrow (1-\alpha) \cdot char_t + \alpha \cdot act_{t\rightarrow t+\Delta t}$$

HABS is suitable for use with function approximators (needed when the abstract state space becomes too large for a tabular representation) on the high level because its higher level has the same structure as normal state spaces where function approximators are used. It can also be extended to more layers, because a new (more) abstract state space can be made out of an abstract state space.

[1]It does not need to be a *vector*, but could be any way of defining a change in the abstract state space.

3.1 Self-organizing Structure

The subpolicies need to cover the commonly used parts of the abstract Behaviour Space, if they are to be useful as actions for the high level policy. But since there is no *a priori* knowledge about what behaviours are needed, the subpolicies start without meaningful behaviour and with randomly initialized characteristic behaviour. This is a crucial difference with other hierarchical RL approaches, where the structure of desired behaviors (macro-actions, options, sub-tasks) is typically predefined.

This means that in the beginning (much) exploration is needed on both levels. The high level selects actions (subpolicies) to execute and the subpolicy (while exploring) might stumble upon a new high level state and when the translation is closest to its characteristic behaviour, it is rewarded (and its behaviour reinforced). The characteristic behaviour is then moved in the direction of the just executed successful behaviour.

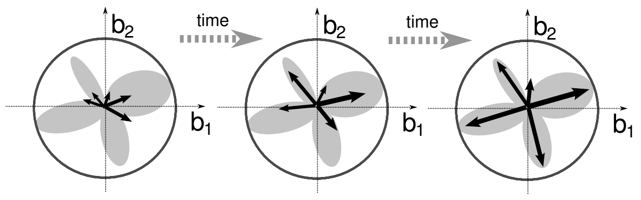


Figure 3: Changing characteristic behaviour vectors*. The grey areas depict pockets of behaviours that actually occur in the abstract state space. The arrows are the characteristic behaviours organizing themselves to cover the needed behaviours.*

Subpolicies are punished when they don't succeed in leaving a high level state, so they are *forced outward*. This prevents the subpolicies from specializing in *standing still*. Assuming that the actually occurring behaviours are not distributed evenly in the Behaviour Space (because of a suitable abstract state space), but cluttered together, the characteristic behaviours move away from each other. This is because each time a subpolicy acts close to its characteristic behaviour, the characteristic behaviour moves towards what the subpolicy actually did. Each characteristic behaviour therefore gravitates towards one of the centers where the density of actually occuring behaviours is high and it moves away from other centers, leaving them free for other characteristic behaviours. In this way, the different characteristic behaviour vectors distribute themselves over the Behaviour Space (fig. 3). This is analogous, in an important sense, to competitive learning and self-organising maps, which similarly attempt to self-organize into a set of "characteristic behaviour vectors" for clusters of vectors.

3.2 Related Work

HABS was inspired by an existing algorithm, HASSLE (Hierarchical Assignment of *Subgoals* to Subpolicies LEarning) [4]. Both are based on abstract states, and have a set of initially non-committed subpolicies which self-organize to specialize for subtasks indicated by a high level policy. However, in contrast to HABS, HASSLE uses abstract states (subgoals) as its *actions* for the higher level. This means that the Q-values table for the higher level grows quadratically with the size of the abstract state space, because each HASSLE-subgoal is only used one time in the entire state space. This is the reason why HABS-subpolicies have behaviours assigned to them instead of subgoals.[2]

Approaches like MAXQ [1] and HEXQ [5] can exploit the fact that a subtask may need only a selection of features or states, which is somewhat similar to the notion of abstract states; but they do not have the same concept of *behaviors as changes in a high level state space*. Another important feature of HABS is that the rewards for the subpolicies are *independent* of the global reward, in contrast with MAXQ, HEXQ and HAM [6].[3]

Approaches like HAM depend on discrete representations. HAM works by designing a hierarchy of finite state machines, which leaves little room for function approximators. MAXQ requires that a task decomposition graph is designed, and it is not immediately obvious how function approximators can be used in the policies. The same holds for HEXQ which automatically decomposes the problem by looking at variables in the state space that change more or less frequently resulting in a MAXQ-like subtask graph.

When using the options approach [2], generalization can become a problem because all the behaviours/options are added to the flat Reinforcement Learning policy and a function approximator would grow in the number of outputs and become very large.

All these approaches have a unique subpolicy for each unique task. HASSLE and HABS, on the other hand, dynamically assign subpolicies to subtasks. They require designing a suitable abstract state space, whereas most other algorithms need complete task decompositions to be designed. In a sense, this means there is a transfer of the problem of designing task decompositions to the problem of finding a suitable abstract state space. We argue, however, that this new problem can in some ways be easier to solve and provide a way in which much of the hierarchical structure can be learned, an issue which is hard given the standard, task decomposition way of thinking. Our novel approach allows automatic development of large parts of the hierarchical structure and the exploitation of useful

[2]Furthermore, this makes HASSLE unsuitable for function approximators or more than 2 layers.

[3]Even though MAXQ-Q uses such independent rewards to some extent in the form of pseudo-rewards.

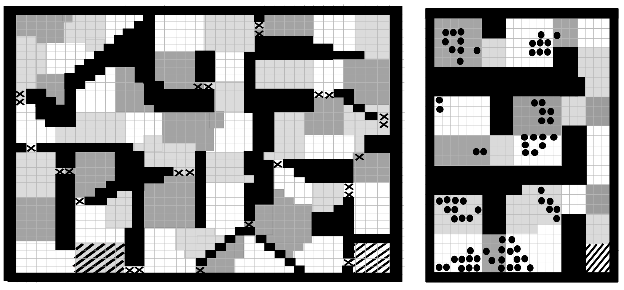
properties of geometric state spaces and corresponding techniques developed for geometric state spaces, such as self-organising vector quantisation. On the other hand, this approach does not have the same theoretical convergence guarantees as MAXQ and the options framework (yet); it is intended to provide a different, complementary way of thinking about hierarchical RL that may eventually lead to algorithms which do have such convergence guarantees.

4 Experiments

The first experiment illustrates that HABS handles large environments well, the second that HABS can use a function approximator, in this case a neural network, for the higher level Q-function on a task where the state space is too big to use discrete states. Furthermore, the first experiment compares HABS to flat RL that uses a table-based representation; the second experiment compares HABS to flat RL using a neural network as its function approximator.

Both experiments were done in grid worlds (see fig. 4) with some cells marked as *walls*, *doors* (only providing extra observational clues about small passages) or *drop areas*. The agent has the primitive actions *North, East, South, West*, *Pickup* and *Drop* (similar to Dietterich's well-known taxi task [1]). In the first experiment one object was present (at a random location) which needed to be picked up and dropped at a drop area. In the second experiment, the agent needed to clean an environment with many objects. The agent received a non-zero high level reward only when an object was delivered at a drop area.

Both experiments illustrate that HABS can handle problems whose state space have some inherent geometric elements (spatial navigation), but also elements which are not geometric: objects which can be picked up and dropped and which can either be or not be in the agent's possession.



(a) big maze (b) cleaner

***Figure 4: environments**: (a) **Big Maze** with $39 \times 36 \approx 1.4 \cdot 10^3$ cells and 50 clusters (colored areas). Black cells are walls, crosses are doors and the striped cells are drop zones. (b) **Cleaning Task**: black dots are examples of the randomly scattered objects.*

4.1 Big Maze Task

For the first experiment, a big maze is used (fig. 4(a)). The lower level states correspond to observation vectors, part of which is generated by a sensor grid (fig. 5). Each of the 32 areas is represented by its average wall/drop area/door/object[4] density, resulting in 4×32 values. To this, features are added for being at a drop area, for being in the same cell as the object, and for carrying the object. HABS uses (linear) function approximators for the lower level subpolicies because they need to generalize.

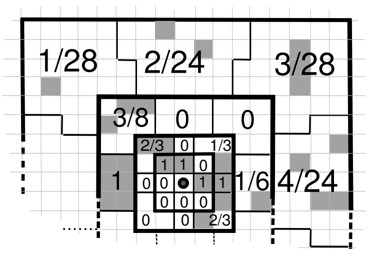


***Figure 5: sensor grid**. The density of each of the 24 areas (8 per ring) is a value in the observation vector, in this example: $\langle \frac{1}{28}, \frac{2}{24}, \frac{3}{28}, \frac{4}{24}, \ldots, \frac{3}{8}, 0, 0, \frac{1}{6}, \ldots, 1, \frac{2}{3}, 0, \frac{1}{3}, 1, \frac{2}{3}, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0 \rangle$.*

The flat learner uses a Q-values table, and has states defined by its coordinates. It needs $(39 \times 36)^2 \approx 2 \cdot 10^6$ states to describe the problem, resulting in $1 \cdot 10^7$ Q-Values. HABS needed to store only $n \times 50 \times 50 \times 2 \approx 5 \cdot 10^3 \times n$ Q-Values for the higher level and n small linear function approximators for the lower level, where n is the number of low-level policies.

The abstract states are tuples consisting of the area (i.e. cluster of low level states) the agent and the object are in, and a boolean for having the object, for example $\langle 3, 1, false \rangle$. The high level Q-function is represented by a table.

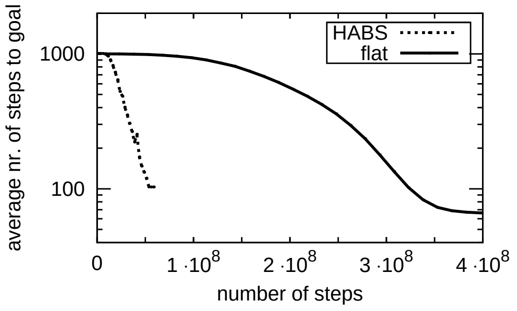


***Figure 6: big maze experiment** results, showing best performances for HABS and the flat learner after tuning a range of parameters.*

[4]Actually, in this first experiment (but not the second) a value of 1 or 0 was given instead of the object density, because the density would make the object nearly "invisible" in the outer rings since there was only one object

The convergence time for the flat learner is approximately $3.5 \cdot 10^8$ steps (see fig. 6). HABS solved the problem in approximately 4 to $5 \cdot 10^7$ steps for a range of parameters, using 25 subpolicies.

4.2 Cleaning Up Task

The second experiment involved "cleaning" an environment (fig. 4(b)) of objects scattered everywhere. The goal was to collect objects ("trash") and drop them in the drop zone. The agent received a reward for every object deposited on a drop area (making it a continuous task). Lower level actions were the same as in the first experiment. Essentially the same sensor grid was used as in the first experiment. However, the sensor grid was used without the outermost areas and there were no doors (so only 3×24 inputs for the subpolicies) and the object *density* was used. Information about the number of objects carried by the agent (maximum capacity was 10), was also included.

For the higher level a coarser version of the sensor grid was used, with each square in fig. 5 representing 5×5 cells, and only 2 rings were used (giving 3×16 inputs). For the higher level Q-values, a multi-layer feedforward neural network with 5 hidden units was used, and the usual linear networks for the 10 subpolicies.

The flat learner used the same neural network architecture as the HABS high level policy. It received the same state information as HABS, but all combined in one vector, instead of separate vectors for the lower and higher level.

Neural networks were used for both HABS and the flat system because the state space was very large and generalization was needed (see [3] for examples of this approach); this allowed us to compare HABS and flat RL when both use function approximators.

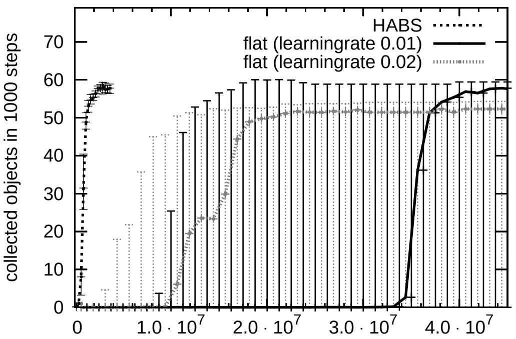


Figure 7: cleaning experiment *with flat learners (5 hidden units) and* HABS *(5 hidden units in higher level) showing performance over a range of parameters (discount, selection temperature).*

The flat learner took longer to reach a good performance, but it could (after extensive tuning) reach a higher performance than HABS (fig. 7). It did so only rarely, however, as can be seen from the wide spread of the results for the flat learner. HABS not reaching the same maximum performance was probably due to the rather simple abstract state space and/or characteristic behaviour vector that was used, leading to suboptimal results.

5 Discussion

HABS outperformed flat Reinforcement Learning by a large factor, especially for large problems where tabular value functions are used. HABS is also suitable for the use of function approximators, such as neural networks, to approximate the value functions at all levels in its hierarchy. It was faster – both in time per step and in steps until convergence – but suboptimal. This is probably because the characteristic behaviours are too simple, or the abstract state space doesn't allow optimal policies. In fact, this is a common characteristic of hierarchical approaches: optimal behavior given the hierarchy may be near optimal, but slightly suboptimal given the space of all policies. However, often this is a price worth paying for more efficient learning in general, and the ability to learn in cases where flat RL is completely infeasible.

In future work we want to study how well HABS performs on larger problems, where more layers (together with function approximators) are needed. It is also interesting to find methods for improving the identification of characteristic behaviors.

References

[1] Thomas G. Dietterich, *"Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition"* (Journal of Artificial Intelligence Research, 2000)

[2] Richard S. Sutton, Doina Precup & Satinder Singh *"Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning"* (Artificial Intelligence, 1999)

[3] Richard S. Sutton & Andrew G. Barto *"Reinforcement Learning: An Introduction". MIT Press, 1998*

[4] Bram Bakker & Jürgen Schmidhuber, *"Hierarchical Reinforcement Learning with Subpolicies Specializing for Learned Subgoals"* Proceedings of the 2nd IASTED International Conference on Neural Networks and Computational Intelligence, 2004)

[5] Bernard Hengst, *"Discovering Hierarchy in Reinforcement Learning with HEXQ"* (In Maching Learning: Proceedings of the Nineteenth International Conference on Machine Learning, 2002)

[6] Ronald Parr & Stuart Russell, *"Reinforcement learning with hierarchies of machines"* (In Proceedings of Advances in Neural Information Processing Systems 10. MIT Press, 1997)