

Concurrent Discovery of Task Hierarchies

Duncan Potts

Computer Science and Engineering
University of New South Wales
Australia
duncanp@cse.unsw.edu.au

Bernhard Hengst

Computer Science and Engineering
University of New South Wales
National ICT Australia
bernhardh@cse.unsw.edu.au

Abstract

Task hierarchies can be used to decompose an intractable problem into smaller more manageable tasks. This paper explores how task hierarchies can model a domain for control purposes, and examines an existing algorithm (HEXQ) that automatically discovers a task hierarchy through interaction with the environment. The initial performance of the algorithm can be limited because it must adequately explore each level of the hierarchy before starting construction of the next, and it cannot adapt to a dynamic environment. The contribution of this paper is to present an algorithm that avoids any protracted period of initial exploration by discovering multiple levels of the hierarchy simultaneously. This can significantly improve initial performance as the agent takes advantage of all hierarchical levels early on in its development. Robustness is also improved because undiscovered features and environment changes can be incorporated later into the hierarchy. Empirical results show the new algorithm to significantly outperform HEXQ.

Introduction

Hierarchical decomposition is the only tractable way of managing many complex systems in the real world. Engineers, software programmers, architects, and indeed anyone working on large difficult problems will naturally attempt to break their work up into smaller more manageable components. Often there is a degree of regularity in the problem which cleverly designed components can exploit through reuse. This is the basis of many CAD packages, planning systems, and even computer languages.

When the complex problem is performing a task or reaching a goal then we can refer to the decomposition as a task hierarchy. Each abstract task in the hierarchy can be completed by acting out a sequence of sub-tasks. At the bottom of the hierarchy the tasks are modular and cannot be decomposed. These primitive tasks often involve basic interactions with the environment. An autonomous system can therefore use a task hierarchy for *control*. A model of the environment can often aid in the construction of a task hierarchy. Indeed, as we shall see, the HEXQ (Hengst 2002) algorithm discovers a specific hierarchical model of the environment that can be directly translated into a task hierarchy.

Defining a task hierarchy in advance requires detailed knowledge of the environment, and a high level understanding of how a problem can be decomposed effectively. As well as requiring significant time and effort, a predefined hierarchy is inflexible and may need to be completely re-built when tackling other problems. The most common solution to these limitations is to predefine a set of actions that are abstract enough to remove the low level uncertainties when dealing with sensor noise and small actuator movements, but general enough to be reused. When also provided with a set of pre- and post-conditions and perhaps other information, these actions can be used by a planner to create a task hierarchy.

In this paper we take a different approach that is more familiar to the reinforcement learning community where the agent relies on far less domain knowledge. The agent only knows the primitive actions that can be taken at any time, but has no prior knowledge regarding the effects of these actions. In addition the agent has no initial information regarding its goal, it simply receives a scalar reward value after each primitive step. The agent must determine for itself the actions that maximise this reward. Although this makes the problem a lot harder, the additional flexibility and the possibility of discovering a more suitable hierarchy than one obtained by a planner makes this a potentially fruitful area of research.

The HEXQ algorithm exploits the fact that state information is often provided by a number of predefined state variables, for example the input from different sensors. It discovers low level tasks that manipulate a small subset of these state variables. Increasingly complex tasks are then formed from lower level tasks until the agent has a full task hierarchy that can control all state variables.

Completely defining the bottom level before tackling the next may, however, be disadvantageous. For example, if there are sparse goals in a large environment the agent must perform enough exploration to find all of these goals before constructing the lowest level. An agent may benefit from making quick inductive leaps and later correcting, backtracking or augmenting its knowledge if necessary. The contribution of this paper is to show that there can be significant advantage in starting to form higher level concepts before the lower level ones have been fully defined. The emphasis is on concurrent construction of the multiple layers compris-

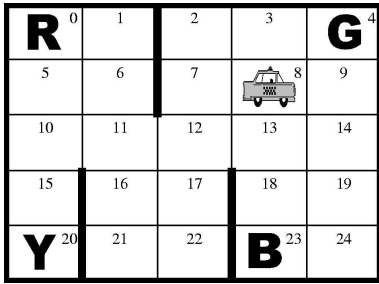


Figure 1: The taxi task.

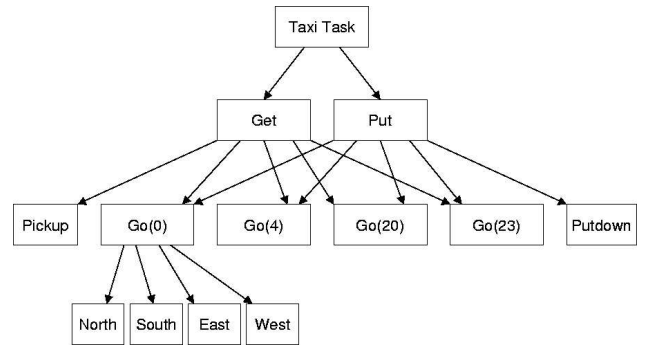


Figure 2: The MAXQ taxi task hierarchy.

ing a task hierarchy, and not on the concurrent learning at multiple levels of a policy constrained by this hierarchy.

The next section describes how the HEXQ algorithm automatically constructs a task hierarchy. We argue that a number of limitations of the algorithm can be fixed by constructing multiple levels of the hierarchy simultaneously, and describe the concurrent HEXQ algorithm which is based on a self-repairing hierarchy. We provide empirical results to demonstrate the improvement with concurrency and conclude with a discussion and future work.

Automatic Discovery of Task Hierarchies

The Taxi Task

Dietterich (2000) introduced the taxi task to motivate his MAXQ hierarchical reinforcement learning framework, and the same example will be used throughout this paper.

In the 5×5 grid world shown in figure 1, a taxi started at a random location must collect a passenger from one of the specially designated locations R(ed), G(reen), Y(ellow) or B(lue). For successful completion of the task the passenger must be dropped off at their destination (also one of R, G, Y or B). The taxi navigates around the world using four primitive actions: North, South, East and West, which move the taxi one square in the intended direction with 80% probability, and in a perpendicular direction with 20% probability. Once at the starting location the taxi must perform the Pickup action, then navigate to the destination and perform the Putdown action, thereby completing the trial. A reward of +20 is given for a successful delivery, a penalty of -10 for performing Pickup or Putdown at the wrong location, and -1 for all other actions.

The taxi problem can be formulated as an episodic Markov decision process (MDP) with the 3 state variables: Taxi Location $\in \{0, \dots, 24\}$, Passenger Location $\in \{R, G, Y, B, \text{and Taxi}\}$, and Destination $\in \{R, G, Y \text{ and } B\}$. It is clear that the navigation policy to each starting and destination location can be the same whether the taxi intends to collect or drop off the passenger. The usual flat formulation of the MDP will solve the navigation sub-task as many times as it reoccurs in the different contexts of passenger location and destination.

The task can be solved efficiently using the task hierarchy in Figure 2. For example Go(20) can be used by Get to collect the passenger from Y, and also by Put to drop

off the passenger at Y. MAXQ uses this hierarchy to obtain considerable savings in both storage and learning time over a non-hierarchical learner (Dietterich 2000). However MAXQ places the burden of defining the hierarchy on the programmer, who must specify the range of states for each sub-task (active states), the termination states for each sub-task classified into undesirable (non-goal) and desirable (goal) termination states, and the set of primitive and abstract actions applicable in each sub-task.

HEXQ

The HEXQ algorithm automates the decomposition of such a problem from the bottom up by finding repetitive regions of states and actions. These regions, and the different ways the agent can move between them, form higher level states and tasks. The next level is constructed by finding repetitive regions in this higher level representation. In this way each level is built upon the preceding level until an entire task hierarchy is constructed.

Representation of a primitive state s by a set of n arbitrarily numbered state variables, $s = \{sv_i | i = 1, \dots, n\}$, plays a significant role in grounding the task hierarchy. HEXQ decomposes the problem variable by variable. The individual state variables are used to construct the different levels of the hierarchy.

Initially the variables are sorted by frequency of change. The motivation behind this heuristic is to use the faster changing variables to construct repetitive regions and associate variables that change value slowly with the durative context. In the taxi domain the passenger location and destination change less frequently. The algorithm therefore explores the behaviour of the taxi location variable first, and makes this the lowest level state variable $level_1.svs = \{\text{Taxi Location}\}$.

The agent explores the state space projected onto the taxi location variable using primitive actions. The region shown in Figure 3 is formed using transitions that are found to be invariant with respect to the context of the higher level variables, the passenger location and destination.

Some transitions are discovered not to be invariant with respect to this context. For example, a pick up from taxi location 0 may or may not succeed in picking up the passenger, depending on whether the passenger is also at that

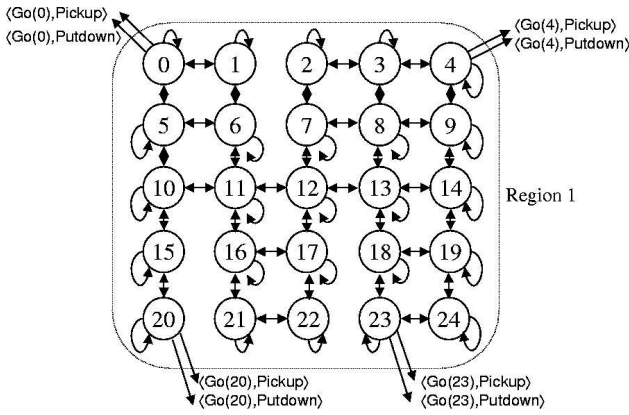


Figure 3: The state transitions for the taxi location, showing the 8 exits.

location. These unpredictable transitions are declared *exits* and the corresponding states *exit states*. In this example exits correspond to potential ways of picking up and putting down the passenger, but they may have a more abstract interpretation in other problems.

The lowest level of tasks in the hierarchy consist of reaching these exit states and performing the corresponding exit action. The policies for these tasks represent temporally extended or abstract actions. From the viewpoint of an agent that can only sense the passenger location and destination, performing these abstract actions is all that is necessary for it to solve the overall problem.

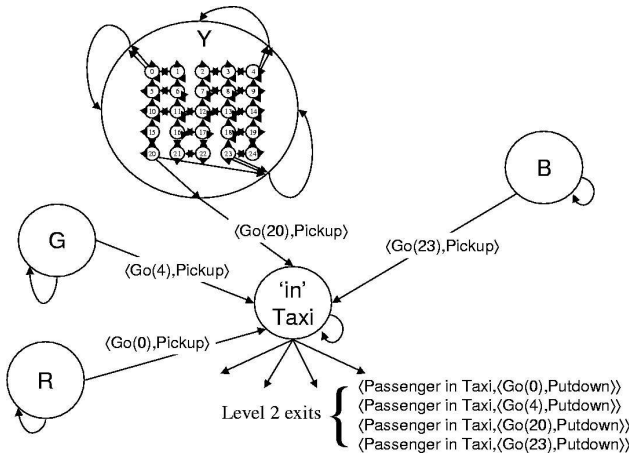


Figure 4: State transitions for the passenger location at level 2 in the hierarchy, showing the 4 exits. State Y is expanded to show the lower level state detail.

The algorithm will now use the next most frequently changing variable, the passenger location, to define level 2 in the hierarchy, $level_2.svs = \{\text{Passenger Location}\}$. Regions are formed in a similar fashion to before, but using the level 1 tasks as abstract actions. Figure 4 shows the region formed by the abstract transitions invariant with respect to the one remaining state variable, the destination. Each state

at this second level of the hierarchy is an abstraction of the entire level 1 navigation region. The four exits represent abstract actions that pick up the passenger (wherever they are) and put them down at one of the four possible destinations.

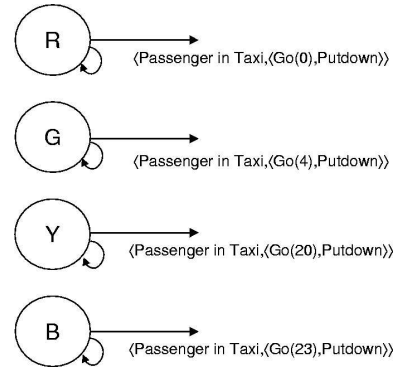


Figure 5: The top level sub-MDP for the taxi domain showing the abstract actions leading to the goal.

The final level states are defined by the destination, $level_3.svs = \{\text{Destination}\}$, and the policies to leave the four exits at the second level form the abstract actions, which can be used to solve the entire problem (Figure 5).

The constructed task hierarchy is detailed in Figure 6. To illustrate its execution assume that the taxi is initially located randomly at location 5, the passenger is at G and wishes to go to Y. At the top level the agent perceives the destination as Y and takes the abstract action $\langle \text{Passenger in Taxi}, \langle \text{Go}(20), \text{Putdown} \rangle \rangle$. This sets the subgoal state at level 2 to passenger location 'in' Taxi. At level 2, the taxi agent perceives the passenger location as G, and therefore executes abstract action $\langle \text{Go}(4), \text{Pickup} \rangle$. This abstract action sets the subgoal state at level 1 to taxi location 4. The level 1 policy is now executed using primitive actions to move the taxi from location 5 to location 4 and the Pickup action is executed to exit. Level 1 returns control to level 2 where the state has transitioned to Taxi. Level 2 now completes its instruction and takes abstract action $\langle \text{Go}(20), \text{Putdown} \rangle$. This again invokes a level 1 policy

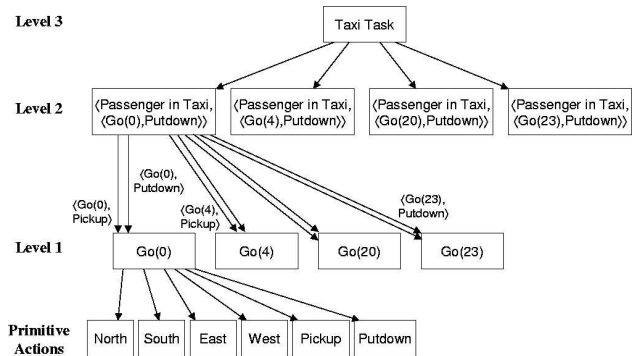


Figure 6: The HEXQ graph showing the hierarchical structure automatically generated by the HEXQ algorithm.

to move the taxi from location 4 to 20 and then executes **Put-down** to exit. Control is returned back up the hierarchy and the trial ends with the passenger delivered correctly.

Although the HEXQ algorithm is able to automatically generate a task hierarchy from its own experience, it is unable to update the hierarchy if it later receives contradictory evidence. It therefore requires a domain-specific amount of initial exploration to determine a valid hierarchy with high probability. The user must specify in advance the amount of exploration to perform at each level of the hierarchy. If the algorithm does not gather enough experience to determine all possible exits, then performance may suffer to the extent that the agent cannot reach the required goal. It is also unable to track a non-stationary environment.

The remainder of this paper describes and evaluates a mechanism that allows an agent to concurrently construct all levels of a task hierarchy. This concurrency is possible when the agent has the ability to repair its task hierarchy in the light of later contradictory evidence, therefore significantly reducing initial construction time, and improving robustness. The new algorithm is referred to as ‘concurrent HEXQ’.

Self-repairing Task Hierarchy

In order to self-repair an agent must compare the response of the world around it with its own internal model. If actual experiences contradict the model, then adjustments must be made so that the model more accurately reflects the real world. The same is true of a task hierarchy; indeed a task hierarchy can be viewed as a hierarchical model of the environment that is specifically tailored for control.

Therefore whenever the agent takes an action and observes its effect upon the world, this transition must be analysed. In the lowest level of the hierarchy this will be a primitive action, however at higher levels the action will be abstract and may comprise many primitive actions and state transitions. This ability to analyse its own transitions is what gives the agent the ability to repair its task hierarchy.

The problem of building an initial hierarchy will be discussed later, for now we assume that an incomplete task hierarchy has already been constructed by the agent. As an example we will use the same taxi problem described earlier for which the lowest level of the hierarchy is shown in Figure 3. The 8 region exits represent 8 abstract actions, e.g. go to location ‘0’ and pick up the passenger.

Figure 7 shows an agent’s partial knowledge of this level, where it has only discovered 4 of the 8 exits. These exits directly correspond to the concept of an abstract action at the next level up in the task hierarchy (see Figure 4). The agent also incorrectly believes there are two separate regions. In general the missing information comprises:

1. Entire states that the agent has not yet visited (e.g. states 2 to 4).
2. Transitions that have not been traversed (e.g. between states 18 and 19).
3. An incorrect number of regions (in this case the agent thinks there are two separate regions because it has never experienced the transitions between states 12 and 13).

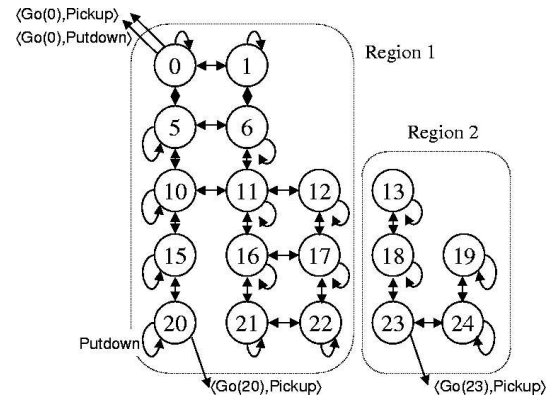


Figure 7: Partial knowledge of level 1.

4. Exit transitions for which not enough experience has been gathered to determine with high probability that they are an exit (e.g. the put down action in state 20).

As a result the agent may discover new transitions, states, regions and exits as it obtains more experience, and each of these may happen at any level of the hierarchy.

States

As the agent moves around the state spaces at each level of the hierarchy, new states can be discovered. If the agent transitions from state s_k to a new state s'_k , then state s'_k is simply added to the same region that s_k belongs to. New states do not change the action hierarchy in any way; it is only new exits or a different number of regions that have an impact.

A state s_k at level k is uniquely identified by the state variables of that level and, if $k > 1$, the region in the level below, $region_{k-1}$ (e.g. see line 5 in Table 1).

Transitions

At level k in the hierarchy, when the agent takes an action a_k in state s_k and transitions to state s'_k with reward r (the last primitive reward received - previous rewards received during a temporally extended abstract action are analysed by lower levels of the hierarchy), the experience can be expressed as the tuple $\langle s_k, a_k, s'_k, r \rangle$. If this transition has previously been declared an exit then no further action is taken (it would be possible to store recent exit statistics and close up exits that disappear over time in a non-stationary environment, but this has not been implemented). The two states each have an associated context c_k and c'_k consisting of the state variables sv_i at all higher levels in the hierarchy (see Table 2 lines 8 and 13). If $c_k \neq c'_k$ then a higher level state variable has changed and the transition is an exit.

If the transition has not been declared an exit, then the experience takes place in the context c_k . If this experience can be abstracted and form part of a region then the transition probabilities do not depend on the context, and $P(s'_k | s_k, a_k, c_k) = P(s'_k | s_k, a_k)$ and $P(r | s_k, a_k, s'_k, c_k) = P(r | s_k, a_k, s'_k)$. If either of these equalities can be shown by statistical tests to be false with high probability then the

Table 1: The BuildHierarchy algorithm.

```

function BuildHierarchy(no. state variables  $n$ )
1  level  $k \leftarrow 1$ 
2  state variables  $level_k.svs \leftarrow \{1, \dots, n\}$ 
3  frequency of change  $svf_i \leftarrow 0, i = 1, \dots, n$ 
4  repeat forever
5     $s_k \leftarrow \langle \{sv_i | i \in level_k.svs\}, region_{k-1} \rangle$ 
6    choose action  $a_k$  using current policy
7     $a_k \leftarrow \text{PerformAction}(a_k, k)$ 
8    increment  $svf_i$  for all changed variables  $sv_i$ 
9     $s'_k \leftarrow \langle \{sv_i | i \in level_k.svs\}, region_{k-1} \rangle$ 
10   add  $\langle s_k, a_k, s'_k, r \rangle$  to transitions
11    $j \leftarrow \text{argmax}_{i \in level_k.svs} svf_i$ 
12   if  $svf_j = d$ 
13      $level_{k+1}.svs \leftarrow level_k.svs - sv_j$ 
14      $level_k.svs \leftarrow sv_j$ 
15      $level_k.regions \leftarrow \text{FormRegions}(s_j,$ 
                                      $transitions)$ 
16      $k \leftarrow k+1$ 
17   end if
18 end repeat
end BuildHierarchy

```

transition cannot be abstracted and it is therefore declared an exit. The χ^2 test is used for the discrete transition distribution (Chrisman 1992), and the Kolmogorov-Smirnov test for the continuous reward distribution (McCallum 1995).

All of the above analysis is performed by the ‘AnalyseTransition’ function called in line 14 of Table 2.

Regions

Regions consist of sets of states that are strongly connected (there is a policy by which every state can be reached from every other state without leaving the region). Therefore it is possible to guarantee leaving a region by any specified exit.

If a transition is found between two regions in a level (e.g. between states 12 and 13 in Figure 7) then it may be possible to merge the regions. Also whenever a new exit is declared, it may affect the strongly connected property of the region and therefore the regions must be re-formed. These region checks are also performed by the ‘AnalyseTransition’ function.

When the number of regions change at a level k , the number of states at level $k+1$ must change accordingly (each state s_{k+1} can only map to a single region at level k). This renders the existing task hierarchy at levels greater than k useless, so it is removed and re-grown. For clarity this mechanism is not detailed in the algorithms. In the domains considered this drastic measure only happened early on in the hierarchy construction, and proved not to be an issue. The connectedness of the lower levels and the number of regions was quickly established. The effort of merging regions pays off as it leads to a reduction in the number of higher level states, faster learning, and less memory usage due to the more compact representation.

Table 2: The PerformAction function.

```

function PerformAction(action  $a_{k'}$ , level  $k'$ )
1  if  $k' = 1$ 
2    take primitive action  $a_{k'}$ 
3    return  $a_{k'}$ 
4  end if
5  do
6     $k \leftarrow k' - 1$ 
7     $s_k \leftarrow \langle \{sv_i | i \in level_k.svs\}, region_{k-1} \rangle$ 
8     $c_k \leftarrow \{sv_i | i \in \bigcup_{m>k} level_m.svs\}$ 
9    while  $s_k \neq a_{k'}.exit\_state$ 
10   choose action  $a_k$  using current policy
11    $a_k = \text{PerformAction}(a_k, k)$ 
12    $s'_k \leftarrow \langle \{sv_i | i \in level_k.svs\}, region_{k-1} \rangle$ 
13    $c'_k \leftarrow \{sv_i | i \in \bigcup_{m>k} level_m.svs\}$ 
14   AnalyseTransition( $\langle s_k, a_k, s'_k, r \rangle, c_k, c'_k$ )
15   if  $\langle s_k, a_k \rangle$  is an exit
16     return  $a_{k'}$  for this exit
17   end if
18    $s_k \leftarrow s'_k$ 
19   end while
20    $a_k \leftarrow \text{PerformAction}(a_{k'}.exit\_action, k)$ 
21   while  $a_k \neq a_{k'}.exit\_action$ 
22   return  $a_{k'}$ 
end PerformAction

```

Exits

Finding a new exit in region $region_k$ corresponds to the discovery of an abstract action a_{k+1} at the level above, or equivalently the task of leaving the region by that particular exit. As discussed above this can change the number of regions and render all higher levels in the hierarchy invalid. Usually, however, the new exit does not alter the number of regions and can be easily incorporated into the hierarchy. The exit requires the addition of the abstract action a_{k+1} to all abstract states s_{k+1} in level $k+1$ that are abstractions of the region $region_k$.

Initial Task Hierarchy Construction

When a fixed hierarchy is built without the ability to self-repair, the agent must perform enough exploration at each level of the hierarchy to enable it to correctly identify all region exits before building the next level of the hierarchy. This exploration is not required when the hierarchy can self-repair. However in order to construct an initial hierarchy, the agent must determine an order over the state variables. HEXQ used a heuristic that placed higher changing variables at lower levels of the hierarchy, and the variable with lowest frequency of change at the top. Experimentally this worked well for discrete environments, and the same technique has been applied here.

At each level of the hierarchy actions are taken (using primitive actions at the lowest level, or abstract actions at higher levels) until the most frequently changing state variable reaches a value d (Table 1 line 11). For our experiments $d = 30$, but the results are not sensitive to this value. For example in the taxi task this means that the taxi moves 30 times

before building level 1. Then the algorithm waits for the passenger location to change 30 times before creating level 2. In this way an entire task hierarchy is quickly established that can be altered as required.

Table 1 shows the algorithm for building an initial hierarchy. Concurrent HEXQ does not require the domain-specific exploration constants that need to be tuned for each level in HEXQ.

Control

The discovered task hierarchy can be used in several ways to determine the best behaviour for the agent to take. The agent can learn a model corresponding to each region, and use this model to plan the best route to the specified exit. Because the problem has been broken down hierarchically, the size of the model could be exponentially smaller than the size of the model for the flat problem.

It is also possible to use model-free learning, and in particular learn an action-value function that decomposes across the different hierarchical levels, as in the original HEXQ algorithm. A decomposed value function has been shown to significantly improve learning times (Dietterich 2000).

Exploration Actions

Care must be taken when the agent is simultaneously learning a policy, whilst concurrently discovering the task hierarchy. It is possible for the agent to become trapped because what it thinks is the best action for a particular state (indeed it may be the only action), leads nowhere. For example if the agent is in region 2 of Figure 7 then there is only one exit to take. If the passenger is at undiscovered state 4, then performing the pick up action at state 23 will not change the state, and although the algorithm exits to the next level up, it will immediately re-enter the bottom level and take the same exit again, as this is the only exit it is aware of. Even commonly used exploration policies will not be effective here, as the agent must take a large number of off-policy steps in order to discover an alternative exit (in the example it needs to find its way to state 4).

To prevent this behaviour, special exploration actions have been incorporated in each region. When an exploration action is executed the agent will take random actions within the region until it leaves by a known exit, or discovers a new one. These exploration actions are treated identically to all other actions and selected according to the exploration policy.

Results

The concurrent HEXQ algorithm will be illustrated in two domains. The first domain is the taxi task, and the second is a large maze with only a single goal. For both experiments and all algorithms there was no discounting of rewards, the learning rate $\alpha = 0.25$ and the policies were greedy. The stochasticity in the movement of the agent and optimistic initialisation of the value function resulted in all algorithms converging to the optimal policy. All results show the average of 20 runs, with error bars indicating one standard deviation.

Table 3: Algorithm differences.

	Uses a task hierarchy	Discovers a task hierarchy
Q-learning	No	No
MAXQ	Yes	No
HEXQ	Yes	Level by level
Concurrent HEXQ	Yes	Concurrently

The Taxi Task

In this domain concurrent HEXQ is compared with the original HEXQ algorithm, Dietterich's MAXQ algorithm and Q-learning (Watkins 1989). Table 3 highlights the differences between these algorithms, and Figure 8 shows the different performances. Figure 9 shows the same data with different scales to highlight the convergence characteristics.

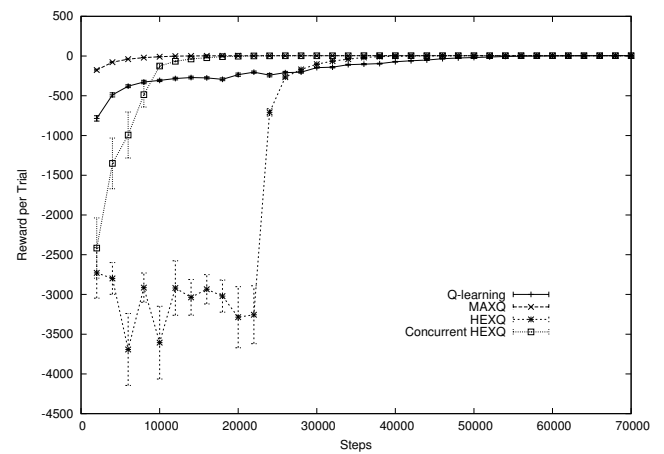


Figure 8: Performance for the taxi task.

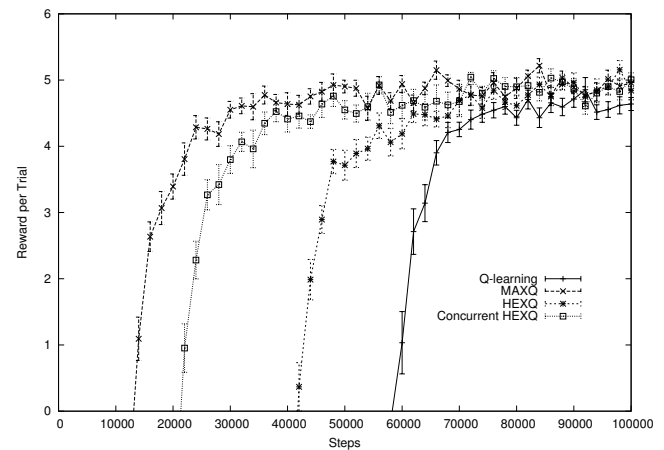


Figure 9: Convergence for the taxi task.

MAXQ has good initial performance due to its prior knowledge of the task hierarchy, which it uses to rapidly learn the optimal policy. HEXQ has no such initial knowledge, but is able to discover the hierarchy level by level. The effort of discovering the task hierarchy pays off when it overtakes the performance of Q-learning. Concurrent HEXQ is able to form an initial task hierarchy much quicker, and then repair any errors at each level. This results in rapid initial learning, and its convergence is not significantly slower than MAXQ, illustrating the high efficiency with which it constructs its hierarchy.

Discovering Sparse Goals

This experiment was designed to illustrate a task that HEXQ should be able to solve efficiently, but at which it is unable to obtain satisfactory performance, due to the nature of the goal. The task is a large maze of 49 rooms in a 7×7 grid. The agent may occupy one of 7×7 locations within each room. In the middle of each internal wall is a door allowing the agent to pass to another room. The agent can move north, south, east and west with a 20% chance of moving perpendicular to its intended direction. The goal is to reach one corner of the world and perform a special fifth action.

HEXQ should be able to quickly form the concept of a room, so that it can leverage this information and move efficiently around the maze. However in order to correctly specify this bottom level of the hierarchy, it must perform enough exploration to find the goal exit.

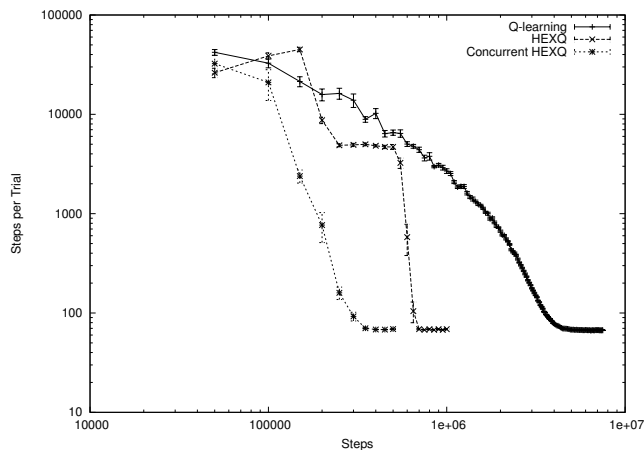


Figure 10: Performance in the large grid world.

Figure 10 shows the performance of Q-learning, HEXQ and concurrent HEXQ at this task. All algorithms have a similar initial performance while they do little more than a random walk. HEXQ must adequately explore each level of the hierarchy before constructing the next, and this results in distinctive steps in its performance that hinder its learning rate. Concurrent HEXQ, however, is able to quickly form the concept of a room and use this to obtain faster convergence. The log-log graph also shows that all 3 algorithms eventually converge to a similar number of steps per trial, with concurrent HEXQ reaching this optimal perfor-

mance twice as quickly as HEXQ and an order of magnitude quicker than Q-learning.

Computational Performance

The ‘BuildHierarchy’ algorithm itself is very fast. The major calculations are the statistical tests in the ‘AnalyseTransition’, however these can only be performed when enough data has been collected. In addition when the existence or non-existence of an exit has been established to the required degree of confidence, it is no longer necessary to perform these tests or store transition statistics. Therefore in very large state spaces the memory usage can be limited.

In the experiments performance was limited by the policy learning algorithm, and not by the building of the hierarchy.

Discussion and Related Work

Discovering and constructing a task hierarchy and learning a hierarchical policy seem to be separate but related activities. Often designers provide the task hierarchy beforehand as background knowledge and then learn the policy.

Stone (2000) advocates a layered learning paradigm to complex multi-agent systems in which learning a mapping from an agent’s sensors to effectors is intractable. The principles advocated include problem decomposition into multiple layers of abstraction and learning tasks from the lowest level to the highest in a hierarchy where the output of learning from one layer feeds into the next layer. This paper highlights the distinction between the discovery of a task hierarchy and learning an optimal policy over the hierarchy. It appears that when the value function is decomposed over multiple levels, learning too early at higher levels may slow down convergence because the agent must unlearn initially sub-optimal policies incorporated in higher level strategies (Dietterich 2000; Hengst 2000). When the sub-goals are not costed into the higher level strategies and simply have to be achieved, learning at multiple levels may be advantageous (Whiteson & Stone 2003).

Utgoff & Stracuzzi (2002) point to the compression inherent in the progression of learning from simple to more complex tasks. They suggest a building block approach, designed to eliminate replication of knowledge structures. Agents are seen to advance their knowledge by moving their “frontier of receptivity” as they acquire new concepts by building on earlier ones from the bottom up. Their conclusion is that the “learning of complex structures can be guided successfully by assuming that local learning methods are limited to simple tasks, and that the resulting building blocks are available for subsequent learning.”

Digney (1998) uses high reward gradient and bottleneck states to distinguish features worth abstracting in reinforcement learning. However this method relies heavily on the form of the reward function, and bottleneck states can only be distinguished after the agent has gained some competence at the task in hand.

In a practical implementation, Drummond (2002) detects and reuses metric sub-spaces in reinforcement learning problems. He finds that an agent can learn in a similar situation much faster after piecing together value function fragments from previous experience.

The idea of refining a learnt model based on unexpected behaviour is also developed by Reid & Ryan (2000). Here a hierarchical model, RL-TOPs, is specified using a hybrid of planning and MDP models. Planning is used at the abstract level and invokes reactive planning operators, extended in time, based on teleo-reactive programs (Nilsson 1994).

De Jong & Oates (2002) use co-evolution and genetic algorithms to discover common building blocks that can be employed to represent larger assemblies. The modularity, repetitive modularity and hierarchical modularity bias of their learner is closely related to the state space repetition and abstraction used by HEXQ and concurrent HEXQ. This work suggests that some concurrency in hierarchical task construction is useful. In this case, the building blocks at a lower level are evaluated on the extent to which they are useful to build higher level assemblies. The evaluation is not possible until there has been some attempt at discovering higher level structures.

Conclusions and Future Work

In this paper we have presented an automatic approach to task hierarchy creation that is applicable when there is only minimal domain knowledge available. The reinforcement learning setting requires only a scalar reward signal to be fed back to the agent which is typically positive when a goal is reached, and negative or zero otherwise.

The concurrent construction of multiple layers in a task hierarchy can give better results than building the hierarchy level by level. The initial guidance given to the agent by the higher levels can significantly improve initial performance, even before the lower levels have been fully defined. Discovery of a task hierarchy is distinct from learning a behavioural policy constrained by this hierarchy, however learning techniques that decompose the value function over the hierarchy can be applied effectively to learn a control policy, even while the task hierarchy is still being built.

Complex actions may be represented in a factored form. For example, speaking, walking and head movements may be represented by three independent action variables. The action space is defined by the Cartesian product of each of the variables. Decomposing a problem by factoring over states and actions simultaneously results in parallel hierarchical decompositions, where the MDP is broken down into a set of sub-MDPs that are “run in parallel” (Boutilier, Dean, & Hanks 1999). Factoring over actions alone has been considered by Pineau, Roy, & Thrun (2001).

We conclude that while the discovery of a task hierarchy will proceed from the bottom up, there can also be a significant advantage in concurrently building higher layers in the hierarchy before the lower layers are complete. The ability to improve or repair the lower layers also provides for a more robust solution.

References

- Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research* 11:1–94.
- Chrisman, L. 1992. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 183–188.
- de Jong, E., and Oates, T. 2002. A coevolutionary approach to representation development. *Proceedings of the ICML-2002 Workshop on Development of Representations*.
- Dietterich, T. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research* 13:227–303.
- Digney, B. 1998. Learning hierarchical control structures for multiple tasks and changing environments. In *From Animals to Animats 5: The Fifth Conference on the Simulation of Adaptive Behavior*.
- Drummond, C. 2002. Accelerating reinforcement learning by composing solutions of automatically identified sub-tasks. *Journal of Artificial Intelligence Research* 16:59–104.
- Hengst, B. 2000. Generating hierarchical structure in reinforcement learning from state variables. In *PRICAI-2000 Topics in Artificial Intelligence*, 533–543.
- Hengst, B. 2002. Discovering hierarchy in reinforcement learning with HEXQ. In *Proceedings of the Nineteenth International Conference on Machine Learning*, 243–250.
- McCallum, A. 1995. *Reinforcement learning with selective perception and hidden state*. Ph.D. Dissertation, University of Rochester.
- Nilsson, N. 1994. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research* 1:139–158.
- Pineau, J.; Roy, N.; and Thrun, S. 2001. A hierarchical approach to POMDP planning and execution. In *Proceedings of the ICML-2001 Workshop on Hierarchy and Memory in Reinforcement Learning*.
- Reid, M., and Ryan, M. 2000. Using ILP to improve planning in hierarchical reinforcement learning. In *Proceedings of the Tenth International Conference on Inductive Logic Programming*, 174–190.
- Stone, P. 2000. *Layered learning in multi-agent systems*. Ph.D. Dissertation, Carnegie Mellon University.
- Utgoff, P., and Stracuzzi, D. 2002. Many-layered learning. *Neural Computation* 14:2497–2529.
- Watkins, C. 1989. *Learning from delayed rewards*. Ph.D. Dissertation, King’s College, Cambridge University.
- Whiteson, S., and Stone, P. 2003. Concurrent layered learning. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*.