Vietnam J Comput Sci (2015) 2:213–226
DOI 10.1007/s40595-015-0045-x

REGULAR PAPER

A multi-agent cooperative reinforcement learning model using a hierarchy of consultants, tutors and workers

Bilal H. Abed-alguni[1] · **Stephan K. Chalup**[1] · **Frans A. Henskens**[1] · **David J. Paul**[2]

Received: 16 February 2015 / Accepted: 22 June 2015 / Published online: 5 July 2015

Abstract The hierarchical organisation of distributed systems can provide an efficient decomposition for machine learning. This paper proposes an algorithm for cooperative policy construction for independent learners, named Q-learning with aggregation (QA-learning). The algorithm is based on a distributed hierarchical learning model and utilises three specialisations of agents: workers, tutors and consultants. The consultant agent incorporates the entire system in its problem space, which it decomposes into sub-problems that are assigned to the tutor and worker agents. The QA-learning algorithm aggregates the Q-tables of worker agents into a central repository managed by their tutor agent. Each tutor's Q-table is then incorporated into the consultant's Q-table, resulting in a Q-table for the entire problem. The algorithm was tested using a distributed hunter prey problem, and experimental results show that QA-learning converges to a solution faster than single agent Q-learning and some famous cooperative Q-learning algorithms.

Keywords Reinforcement learning · Q-Learning · Multi-agent system · Distributed system · Markov decision process · Factored Markov decision process

✉ Bilal H. Abed-alguni
bilal.abedalguni@uon.edu.au

Stephan K. Chalup
Stephan.Chalup@newcastle.edu.au

Frans A. Henskens
Frans.Henskens@newcastle.edu.au

David J. Paul
David.Paul@une.edu.au

[1] School of Electrical Engineering and Computer Science, University of Newcastle, Callaghan, NSW 2308, Australia

[2] School of Science and Technology, University of New England, Armidale, NSW 2351, Australia

1 Introduction

Classical reinforcement learning (RL) algorithms attempt to learn a problem by trying actions to determine how to maximise some reward. One such algorithm is Q-learning, which represents the cumulative reward for each state-action pair in a structure called a Q-table [34,35]. A major problem with these algorithms is that their performance typically degrades as the size of the state space increases [4,31]. Fortunately, many large state space problems can be decomposed into loosely coupled subsystems that can be processed independently [11].

One of the most efficient known approaches for RL decomposition of large size problems is the hierarchical methodology [12,13,27,32]. In this approach, the target learning problem is decomposed into a hierarchy of smaller problems. However, current hierarchical RL techniques do not allow migration of learners from one problem space to another in distributed systems. Instead, they focus on decomposing the state or action space into more manageable parts, and then statically assign each learner to one of these parts.

This paper proposes Q-learning with aggregation (QA-learning), an algorithm for cooperative policy construction for independent learners that is based on a distributed hierarchical learning model. QA-Learning reduces the complexity of large state space problems by decomposing the problem into more manageable sub-problems, and distributing agents between these sub-problems, to improve efficiency and enhance parallelisation [2].

The QA-learning model includes three specialisations of agents: workers, tutors and consultants. The consultant agent is the highest specialisation in the learning hierarchy. Each consultant is responsible for assigning a sub-problem and a number of worker agents to each tutor. The worker agents first learn the problem space of their tutor, then the tutor

aggregates its workers' Q-tables into its own Q-table. The tutors' Q-tables are then merged to create the consultant's Q-table. Finally, the consultant performs a small amount of further learning over the entire problem space to optimise its Q-table.

When a tutor finishes learning its sub-problem, the worker agents assigned to that tutor are released to the consultant, who can then reassign the workers to any tutors that are still learning. Thus, rather than remaining idle, worker agents are migrated to subsystems where they can help accelerate learning. This decreases the overall time required to learn the entire system.

The remainder of the paper is structured as follows: Sect. 2 presents basic definitions, Sect. 3 discusses related work, Sect. 4 introduces a motivating example, Sect. 5 discusses the QA-learning algorithm, Sect. 6 discusses simulation results using a distributed version of the hunter prey problem, and Sect. 7 presents the conclusion of this paper and future work.

2 Background in machine learning

This section briefly summarises some of the underlying concepts of reinforcement learning, and Q-learning in particular.

2.1 Markov decision process

A Markov decision process (MDP) is a framework for representing sequential decision making problems that facilitates a decision maker, at each decision stage, to choose from several possible next states [26]. MDPs are widely used to represent dynamic control problems, where the parameters of the MDP need to be learned through interaction with the environment [1].

An MDP model is a 4-tuple $[S, A, R, T]$ where:

1. $S = \{s_0, s_1 \ldots s_{n-1}\}$ is a set of possible states.
2. $A = \{a_0, a_1 \ldots a_{m-1}\}$ is a set of possible actions.
3. $R : S \times A \rightarrow \mathbb{R}$ is a reward function.
4. $T : S \times A \times S \rightarrow [0, 1]$ is a transition function.

In deterministic learning problems, all transition probabilities can only equal 1 or 0. A transition probability $T(s_i, a_j, s_k) = 1$ means that it is possible for transition from state s_i to state s_k by performing action a_j, while a transition probability $T(s_i, a_j, s_k) = 0$ means that the transition is not possible . The reward received for completing this action is $R(s_i, a_j)$.

The main aim of MDPs is to find a policy π that can be followed to reach a specific goal (a terminal state). A policy is a mapping between the state set and the action set $\pi : S \rightarrow A$. An optimal policy π^* always chooses the action that maximises a specific utility function of the current state. RL optimisation problems are often modelled using MDP inspired algorithms [31].

2.2 Factored Markov decision process

A factored MDP (FMDP) is a concept that was first proposed by Boutilier et al. [6]. A FMDP is an MDP with a state space S that can be specified as a cross-product of sets of state variables $S = S_0 \times S_1 \times \cdots \times S_{n-1}$. The idea of factored state space is related to the concepts of state abstraction and aggregation [10]. This idea is based on the fact that many large MDPs have many parts that are weakly connected (loosely coupled) and can be processed independently [23]. In FMDPs, T_a denotes the state transition model for an action a. This transition model is represented as a dynamic Bayesian network (DBN). It uses a two-layer directed acyclic graph G_a where the nodes $S = (S_0, S_1, \ldots, S_{n-1}, S'_0, S'_1, \ldots, S'_{n-1})$. In G_a, the parents of S'_i are noted as the $parents_a(S'_i)$, where these parents are assumed to be a subset of the state space $parents_a(S'_i) \subset S$. This means that there are no synchronous arcs from S_i to S'_j. The reward function R can be decomposed additively $\gamma_1 R_0 + \gamma_2 R_1, \ldots, +\gamma_{n-1} R_{n-1}$ and the differences of the decomposition do not depend on the state variables [36].

2.3 Q-Learning

Q-Learning is one of the best known RL algorithms that provides solutions for MDPs. This algorithm uses temporal differences (a combination of Monte Carlo methods and dynamic programming methods) to find mappings from state-action pairs to values. These values are known as Q-values, and are calculated using a reward function, called the Q-function, that returns the expected utility of taking a given action in a given state and following a fixed policy after that [34,35]. The fact that Q-learning does not require a model of the environment is one of its strengths [31].

An agent that applies Q-learning needs a number of learning episodes to find an optimal solution. An episode is a learning period that starts from a selected state and ends when a goal state is reached. During an episode, the agent chooses an action a from the set of possible actions from its current state s based on its selection policy. The agent then receives a reward $R(s, a)$ and perceives s', its new state in the environment. The agent then updates its Q-table based on equation (1). This procedure repeats until the agent reaches the goal state or a predetermined number of actions have been taken without the agent reaching its goal, which marks the end of the episode.

$$Q(s, a) \longleftarrow (1 - \alpha)\, Q(s, a) + \alpha\, [R(s, a) + \gamma \max_{a' \in A} Q(s', a')], \quad \text{where } s \in S, a \in A, \alpha \in [0, 1]$$

is the learning rate, and $\gamma \in [0, 1]$ is the discount factor.

(1)

The learning rate α determines how the acquired information will affect the current Q-values. A higher α value makes the agent prefers rewards received in later episodes over earlier reward values. The discount factor γ determines how much the current Q-values will be affected by potential future rewards. The weight of future reward increases as the value of γ approaches 1.

The main output of the Q-learning algorithm is a policy $\pi : S \rightarrow A$ which suggests an action for each possible state in an attempt to maximise the expected reward for an agent. Some variations of Q-learning are combined with function approximation methods such as artificial neural networks instead of Q-tables to represent continuous large state problems [29].

2.4 Cooperative Q-learning

Cooperative Q-learning allows independent agents to work together to solve a single Q-learning problem. Cooperative Q-learning is typically broken into two stages: individual learning, and learning by interaction. In the individual learning stage, each learner independently uses its own Q-learning algorithm to improve its individual solution. Then, in the learning by interaction stage, a Q-value sharing strategy is used to combine the Q-values of each agent to produce new Q-tables.

An example of a Q-value sharing strategy is BEST-Q [17–19]. In BEST-Q, the highest Q-value for each state-action pair is selected from the Q-tables of all of the agents. Then, each agent updates its Q-table by replacing each one of its Q-values with the corresponding best Q-value:

$Q_i(s, a) \longleftarrow Q^{\text{best}}(s, a)\ (\forall i, s, a)$, where i is the agent's identification number.

(2)

2.5 Classical hunter prey problem

The classical hunter prey problem is considered one of the standard test problems in the field of multi-agent learning [25]. Normally, there are two types of agents: hunters and prey. Each agent is randomly positioned in the cells of a grid at the beginning of the game. The agents can move in four directions (up, down, right, left) unless there is an obstacle to the movement direction, such as a wall or boundary. For example, Fig. 1 shows a classical version of the hunter prey problem of grid size 14 × 14 that involves 28 agents: 20 hunter agents (H) and eight prey agents (P).

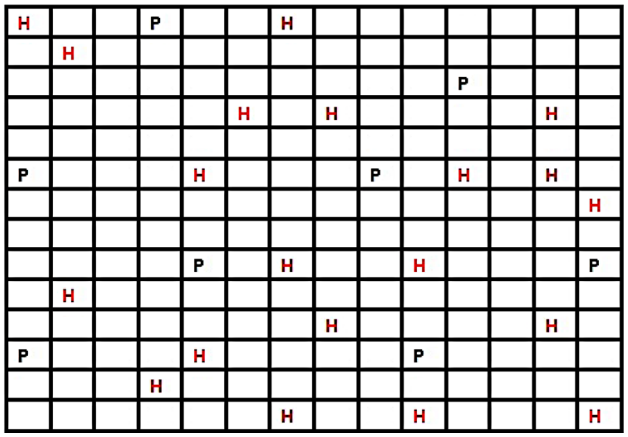

Fig. 1 An example of classical hunter prey problem on a 14 × 14 grid

In a typical hunter prey game, hunters chase the prey agents, and the prey agents try to escape from the hunters. At any instant, the distance between any two agents in a grid is measured using the Manhattan distance [12].

3 Related work

This section is divided into two subsections. Section 3.1 discusses famous approaches of hierarchical decomposition in the RL domain, while Sect. 3.2 discusses cooperative hunting strategies for the hunter prey problem.

3.1 Hierarchical decomposition in the RL domain

Decomposition of MDPs can be broadly classified into two categories. First, static decomposition which partially or totally requires the implementation designers to define the hierarchy [28,30]. Second, dynamic decomposition, in which hierarchy components, their positions, and abstractions are determined during the simulation process [11,16,32]. Both techniques focus on decomposing the state or action space into more manageable parts, and statically assign each learner to one of these parts. None of these techniques allow the migration of agents between different parts of the decomposition.

Parr and Russell [28] proposed a RL approach called HAMQ-learning that combines Q-learning with hierarchical abstract machines (HAMs). This approach effectively reduces the size of the state space by limiting the learning policies to a set of HAMs. However, state decomposition in this form is hard to apply, since there is no guarantee that it will not affect the modularity of the design or produce HAMs that have large state space.

Dietterich [11] has shown that an MDP can be decomposed into a hierarchy of smaller MDPs based on the nature of

the problem and its flexibility to be decomposed into smaller sub-goals. This research also proposed a MAXQ procedure that decomposes the value function of an MDP into an additive combination of smaller value functions of the smaller MDPs. An important advantage of MAXQ decomposition is that it is a dynamic decomposition, unlike the technique used in HAMQ-learning [5].

The MAXQ procedure attempts to reduce large problems into smaller problems, but does not take into account the probabilistic prior knowledge of the agent about the problem space. This issue can be addressed by incorporating Bayesian reinforcement learning priors on models, value functions or policies [9]. Cao and Ray [9] presented an approach that extends MAXQ by incorporating priors on the primitive environment model and on goal pseudo-rewards. Priors are statistic information of previous policies and problem models that can help a reinforcement agent to accelerate its learning process. In multi-goal reinforcement learning, priors can be extracted from models or policies of previous learned goals. This approach is a static decomposition approach. In addition, the probabilistic priors should be given in advance in order to incorporate them in the learning process.

Cai et al. [8] proposed a combined hierarchical reinforcement learning method for multi-robot cooperation in completely unknown environments. This method is a result of the integration of options with the MAXQ hierarchical reinforcement learning method. The MAXQ method is used to identify the problem hierarchy. The proposed method obtains all the required learning parameters through learning without any need for an explicit environment model. The cooperation strategy is then built based on the learned parameters. In this method, multiple simulations are required to build the problem hierarchy which is a time consuming process.

Sutton et al. [30] proposed the concept of options which is a form of knowledge abstraction for MDPs. An option can be viewed as a primitive task that is composed of three elements: a learning policy $\pi : S \rightarrow A$, where S is the state set and A is the action set, a termination condition $\beta : S^{+} \rightarrow [0, 1]$ and an initial set of states $I \subseteq S$. An agent can perform an option if $s_t \in I$, where s_t is the current state of the agent. An agent chooses an option then follows its policy until the policy termination condition becomes valid. In this case, the agent can select another option. A main disadvantage of this approach is that the options need to be determined in advance. In addition, it is difficult to decompose MDPs using this approach because many decomposition elements need to be determined for each option.

Jardim et al. [20] proposed a dynamic decomposition hierarchical RL method. This method is based on the idea that to reach the goal, the learner must pass through closely connected states (sub-goals). The sub-goals can be detected by intersecting several paths that lead to the goal while the agent is interacting with the environment. Temporal abstractions (options) can then be identified using the sub-goals. A drawback of this method is that it requires multiple simulations to define the sub-goals. In addition, this method is time consuming and cannot easily be applied to large learning problems.

Generally, multi-agent cooperation problems can be modelled based on the assumption that the state space of n agents represents a joint state of all agents, where each agent i has access to a partial view s_i from the set of joint states $s = \{s_1, \ldots, s_{n-1}, s_n\}$. In the same manner, the joint action is modelled as $\{a_1, \ldots, a_{n-1}, a_n\}$, where each agent i may only have access to partial view a_i. One simple approach to modelling multi-agent coordination is discussed in the survey study of Barto and Mahadevan [5]. It shows that the concurrency model of joint state and action spaces can be extended to learn task-level coordination by replacing actions with options. However, this approach does not guarantee convergence to an optimal policy since learning low level policies varies at the same time as learning high level policies.

The study of Barto and Mahadevan [5] discussed another hierarchical reinforcement learning cooperation approach. This approach is a hyper approach that combines options [5] and MAXQ decomposition [11] together. An option $o = \langle I, \pi, \beta \rangle$ is extended to a multi-option $\overrightarrow{o} = \langle o_1, \ldots, o_n \rangle$, where o_i is the option that is executed by agent i. A joint action value of a main task p, a state s and a multi-option $\overrightarrow{o}$ is denoted as $Q(p, s, \overrightarrow{o})$. Then the MAXQ decomposition of the Q-function can be extended for the joint action-values.

Hengst [16] proposed HEXQ, a hierarchical RL algorithm, that automatically decomposes and solves MDPs. It uses state variables to construct a hierarchy of sub-MDPs, where the maximum number of hierarchy levels is limited to the number of state variables. The results are interlinked small MDPs. As discussed in [16] the main limitation of HEXQ is that it must discover nested sub-MDPs and find policies for their exits (exits are non-predictable state transitions and not counted as edges of the graph) with probability of 1. This requires that the problem space must have state variables that change over a long time scale.

Tosic and Vilalta [32] proposed a RL conceptual framework for agents' collaboration in large-scale problems. The main idea here is to reduce the complexity of RL in large-scale problems through modelling RL as a process of three levels: single learner level, co-learning among small groups of agents and learning at the system level. An important advantage is that it supports dynamic adaption of coalition among agents based on continuous exploration and adaption of the three layered architecture of the proposed model.

The proposed model of Tosic and Vilalta [32] does not specify any communication scheme among its three RL learning levels. Moreover, the model suffers from the absence of detailed algorithmic specifications on how RL can be implemented in this three layered learning architecture.

Guestrin and Gordon [14] proposed a distributed planning algorithm in hierarchical factored MDPs that solves large decision problems by decomposing them into sub-decision problems (subsystems). The subsystems are organised in a tree structure. Any subsystem has two types of variables: internal variables and external variables. Internal variables are the variables that can be used by the value function of the subsystem, while the external variables cannot be used because their dynamics are unknown. Although the algorithm does not guarantee convergence to an optimal solution, its output plan is equivalent to the output of a centralised decision system. This proposal has some limitations. First, although coordination and communication between agents are not centralised, they are restricted by the subsystem tree structure. Second, the definition of a subsystem as an MDP composed of internal and external variables only fits decision problems.

Gunady et al. [15] proposed a RL solution for the problem of territory division on hide-and-seek games. The territory division problem is the problem of dividing the search environment between cooperative seekers to reach optimal seeking performance. The researchers combined a hierarchical RL approach with state aggregation in order to reduce the state space. In state aggregation, similar states are grouped together in two directions: topological aggregation and hiding aggregation. In topological aggregation, the states are divided into regions based on the distribution of obstacles. In hiding aggregation, hiding places are grouped together and treated as the target of aggregation action. A disadvantage of this algorithm is that it requires the model information of the environment to be known in advance.

The distributed hierarchical learning model described in this paper is based on the structure of modern software systems, where a system is decomposed into multiple subsystems. There are no restrictions on the structure of the system hierarchy. Additionally, there are two levels of learning and coordination between subsystems: at the subsystem level; and at the system level. A major goal of this model is to handle dynamic migration of learners between subsystems in a distributed system to increase the overall learning speed.

3.2 Cooperative hunting strategy

Yong and Miikkulainen [37] described two cooperative hunting strategies that can be implemented in the hunter prey problem. The first is a cooperative hunting strategy for non-communicating teams that involves two different roles of hunters: chasers and blockers. The role of a chaser is to follow the prey movement, while the role of the blocker is to move in a horizontal direction to the prey, staying in the vertical axis of the prey. This allows the blocker to limit the movement of the prey. The second strategy is also a cooperative hunting strategy for non-communicating teams, but only involves chasers. In order for two chasers to sandwich the prey, at least two chasers are required to follow the prey in opposite directions to eventually surround the prey agent. Both hunting strategies were experimentally proven to be successful. One main advantage of these strategies is that no communication is required between hunters. However, both strategies require the prey position to be part of the state definition to provide the chasers and/or blockers with knowledge of the prey's position.

Lee [24] proposed a hunting strategy that also involved hunter roles of chaser and blocker. In this paper, the roles of hunters are semantically similar to the roles of the hunters of the first strategy of Yong and Miikkulainen [37]. However, the description of roles is relatively different. The chasers drive the prey to a corner of the grid, while the role of blockers is to surround the prey so it does not have enough space to escape. The hunting is considered successful if the prey is captured. A main difference between blockers in this paper and [37] is that blockers are required to communicate to surround the prey agent. This is considered as a disadvantage of the hunting strategy of [24]. Communication is a disadvantage of the hunting strategy, because communication between agents requires extra computation.

4 Distributed hunter prey problem

This section introduces a distributed version of the classical hunter prey problem to demonstrate how independent agents can cooperatively learn a policy for a distributed large state space problem. The main argument for this design is that the reduction of a state space S into n state spaces, $S \rightarrow \{S_0, S_1, \ldots, S_{n-1}, S_n\}$, accelerates convergence to the optimal solutions [4,5,10].

Figure 2 shows a new version of the hunter prey problem that is composed of 4 grids of size 7×7 [2]. Firstly, each hunter learns to hunt the prey in its own sub-grid. If a hunter finishes learning its own sub-grid then it can be migrated to

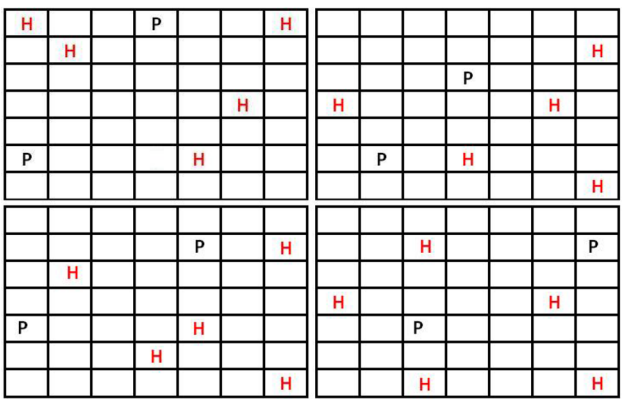

Fig. 2 Distributed hunter prey problem

another sub-grid to enhance that sub-grid's learning speed. Then, once each sub-grid has been learnt, these solutions are aggregated and enhanced to provide a policy for the entire grid.

The cooperative hunting strategies in Sect. 3.2 are not directly applicable for distributed hunter prey problems. Those strategies require hunters to have knowledge of the entire system, while hunters in the distributed hunter prey problem only have knowledge of their local grid. However, the semantic ideas behind chaser and blocker hunters can still be implemented. Section 5.4 will describe one possible implementation.

5 QA-Learning

5.1 Problem model

The problem model of Q-learning with aggregation (QA-learning) is based on a loosely coupled FMDP organised into two levels: system and subsystem. The loose coupling characteristic of an FMDP means that each one of its subsystems has or uses little knowledge of other subsystems [22].

A system is a tuple $[S, A, W, T]$, where S, A, and T are defined as in an MDP (see Sect. 2.1) and W is a set of reward functions $R : S \times A \rightarrow \mathbb{R}$ for different roles that may be used in the system. A role can then be defined as an MDP $[S, A, R, T]$, where $R \in W$.

A subsystem is a MDP with a connection set that defines the subsystem's boundaries with its neighbouring subsystems. More formally, given a role Role $= [S, A, R, T]$, a subsystem is a tuple Sub $= [M, C]$, where

1. $M = [S_{\text{sub}}, A_{\text{sub}}, R_{\text{sub}}, T_{\text{sub}}]$ is a MDP where:
   (a) $S_{\text{sub}} \subseteq S$ is the set of states in the subsystem.
   (b) $A_{\text{sub}} \subseteq A$ is the set of actions in the subsystem.
   (c) $R_{\text{sub}} : S_{\text{sub}} \times A_{\text{sub}} \rightarrow \mathbb{R}$ is a reward function such that, given $s \in S_{\text{sub}}, a \in A_{\text{sub}}, r \in \mathbb{R}$, $R_{\text{sub}}(s, a) = r \iff R(s, a) = r$.
   (d) $T_{\text{sub}} : S_{\text{sub}} \times A_{\text{sub}} \times S_{\text{sub}} \rightarrow [0, 1]$ is a transition function such that, given $s_i, s_j \in S_{\text{sub}}, a_k \in A_{\text{sub}}, t \in [0, 1]$, $T_{\text{sub}}(s_i, a_k, s_j) = t \iff T(s_i, a_k, s_j) = t$.
2. $C : S_{\text{sub}} \times A \times (S \backslash S_{\text{sub}}) \rightarrow [0, 1]$ is a connection set which specifies how *Sub* connects to other parts of the system such that, given $s_i \in S_{\text{sub}}, a \in A, s_j \in S \backslash S_{\text{sub}}, t \in [0, 1]$, $C(s_i, a, s_j) = t \iff T(s_i, a, s_j) = t$.

5.2 Agent specialisations

The design of the hierarchical learning model of the QA-leaning algorithm is based on the specialisation principle.

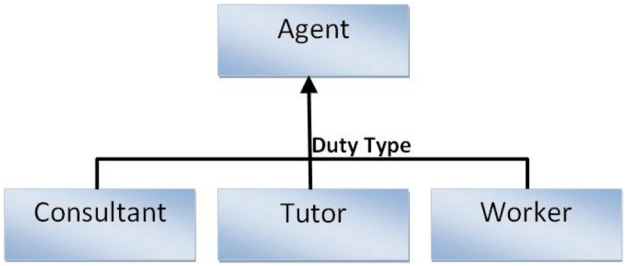

Fig. 3 Generalisation of QA-learning agents

This principle supports the separation of duties among agents in distributed learning problems. The distributed hierarchical learning model includes three agent specialisations: workers, tutors and consultants (Fig. 3).

The following are the basic definitions of the three agent types of the proposed learning hierarchy:

- Worker agents are learners at the subsystem level where each worker can be assigned different roles.
- Tutor agents are coordinators at the subsystem level where each subsystem has one tutor for each role. Each tutor agent aggregates its workers' Q-tables into its own Q-table.
- Consultant agents are coordinators at the system level. A distributed system has a consultant agent for each role (or a single consultant may handle multiple roles). A consultant agent learns the solution at system level by incorporating its tutors' Q-tables into its own Q-table calculations. Consultant agents are also responsible for redistributing worker agents among tutors to help accelerate the overall learning process

5.3 Migration of agents

A major design goal of the QA-learning algorithm is to increase the efficiency of worker agents as much as possible. Worker agents in some subsystems might be working to achieve their goals, while worker agents in other subsystems may have completed theirs. Such a situation requires redistribution of worker agents to the subsystems that are still active to help learn each subsystem's tasks more quickly. For example, Fig. 4 shows that the hunter agents on sub-grid A have finished hunting all their prey agents, while the hunting is still active in sub-grids B, C and D. The workers assigned to the tutor for sub-grid A should be redistributed to the remaining sub-grids

The redistribution of worker agents is one of the responsibilities of consultant agents. A consultant agent has a monitor programme that uses two data structures to monitor tutor's activities. First, a service queue that is used to register the tutors that are still active. An active tutor is one whose worker

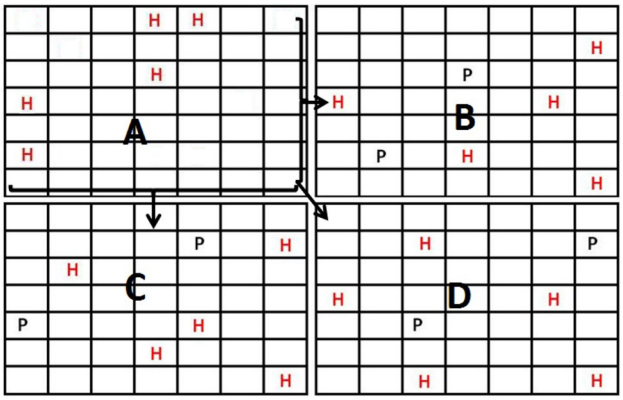

Fig. 4 Partially finished hunting. The hunter agents on sub-grid *A* have finished hunting all their prey agents, while the hunting is still active in sub-grids *B*, *C* and *D*

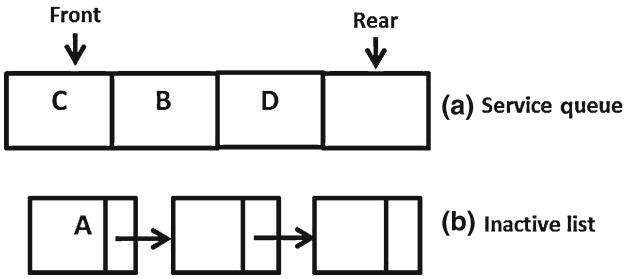

Fig. 5 Service queue and inactive list. The service queue is a first in first out queue (FIFO) and the inactive list is a FIFO list. However, the service queue can be implemented as a priority queue. This figure shows the contents of the lists for the case shown in Fig. 4

agents are still performing or learning their assigned tasks. Second, an activity list that is used to register inactive tutors (inactive list). Figure 5 shows the contents of the service queue and inactive list for the case shown in Fig. 4.

The monitor programme keeps track of the progress of each tutor by recording the number of goals of each tutor, and registering active tutors in the service queue. The monitor programme also initiates the migration of worker agents, when required. The monitor programme works as follows:

1. Register each tutor that is active and working to achieve its goals in the service queue.
2. If any tutor finishes processing its goals, remove it from the service queue, register it in the inactive list, and flag the state of its workers as available.
3. Split the worker agents of the tutor agents registered in the inactive list between the tutor agents in the service queue.
4. Apply the migration procedure for all worker agents that follow the tutor that is registered first in the inactive list.
5. Delete the first entry in the inactive list.
6. Go to step 1.

The algorithm that performs the migration is inspired by the research of Boyd and Dasgupta [7] and Vasudevan and Venkatesh [33] in the field of process migration in operating systems. However, the migration algorithm is an application level algorithm that organises the migration process of reinforcement learners between subsystems. The migration algorithm consists of the following steps:

1. The consultant agent declares the migrant worker to be in a migrating state at subsystem level and at system level.
2. If migrant worker is running in client server mode:
   (a) Duplicate migrant worker.
   (b) Maintain a communication channel between the migrant worker and its copy for the rest of the steps.
   (c) Go to step 5.
3. Write the problem space of the migrated agent to the tutor of the source-subsystem.
4. Terminate the migrant worker process or thread.
5. Relocate the migrant agent to a new subsystem.
6. Inform the destination subsystem' tutor of the migrant's new location.
7. Resume the agent thread.
8. Allocate problem space to the migrant agent.
9. Resume the execution of migrant agent.
10. If a worker agent finishes execution and is running in mobile agent mode:
    (a) Terminate the migrant agent process or thread.
    (b) Deallocate memory and data of the migrant agent.
    (c) Relocate the migrant agent to its original subsystem.

Steps 3 and 8 of the migration algorithm are related to the problem space of RL agents. The problem space is related to the MDP model described in Sect. 2.1.

5.4 Roles of worker agents

Worker agents can play different roles to perform tasks at the subsystem level. As coordinators, consultant agents and tutor agents are responsible for assigning roles to worker agents.

The semantic idea behind both chaser and blocker hunters (Sect. 3.2) can still be implemented to the distributed hunter prey problem (Sect. 4). The following strategy presents a solution for the hunter chasing prey over a grid system:

– Chaser hunter: each chaser hunter learns to chase and catch the prey agents inside its sub-grid. Each chaser hunter inherits the problem space and the Q-table of its tutor agent.
– Blocker hunter: each blocker hunter learns to occupy some blocking cells in the corners of a its sub-grid to stop prey from moving in that direction. Each blocker

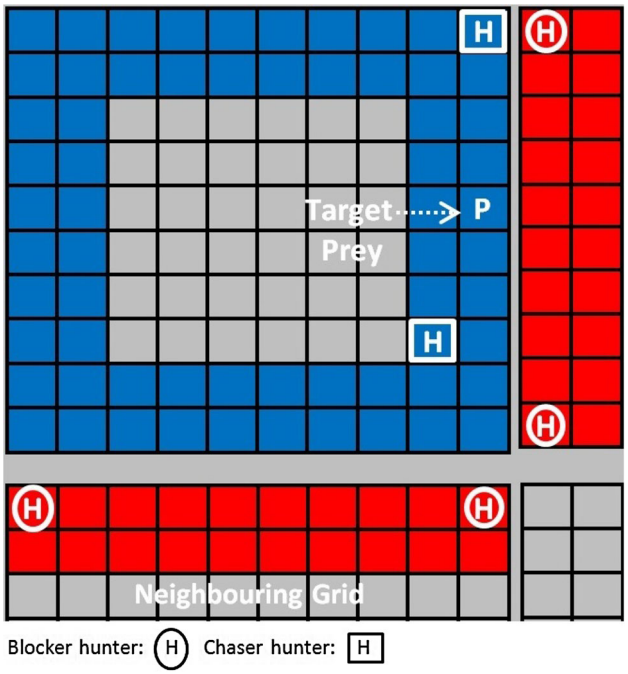

Fig. 6 Blocker and chaser agents in a distributed hunter prey problem. The number of chaser agents in the target prey's sub-grid is 2. Each adjoining sub-grid to the target prey's sub-grid has two blocker agents

hunter inherits the problem space and the Q-table of its tutor agent.

Figure 6 shows an example of a hunting strategy applied to an instance of the distributed hunter prey problem. In this figure, the chaser hunters are located in the same grid as the target prey, while the blocker hunters are located in the neighbouring grids of the target prey. In general, the idea is for the blocker hunters to funnel the prey through the centre of the neighbouring sub-grid, where the chaser hunters can catch it. There are two blocker cells in each corner of the neighbouring sub-grid of the target prey, which the blocker hunters target.

Consultant agents use the hunting algorithm in Fig.7 to choose the blocker and chaser hunters. In the beginning, the algorithm identifies the prey agents of each grid that are most likely to escape (target prey agents) to other grids (Line 2). The algorithm then chooses a specific number of worker hunters to chase each target prey agent (Lines 4 and 5). The chaser hunters of each target prey should be the nearest non-specialised worker hunters to each target prey. These chaser agents inherit the problem space of the tutor agent of the chosen sub-grid (Line 5). The algorithm then continues by choosing a specific number of worker hunters on each neighbouring grid of each target prey agent to be blocker hunters. The blocker hunters of each target prey should be the nearest non-specialised worker hunters to each target prey (Lines 6 and 7). These blocker hunters inherit the problem space of

```
1: for each sub-grid ∈ grid do
2:    Choose the prey agents that have the minimum
      Manhattan distance to the boundaries of each neigh-
      bouring grid to be the target prey agents.
3:    for each target prey do
4:       Choose n number of worker hunters from the
      same grid as the prey to be chaser hunters that sat-
      isfy the following conditions:
        – They are not blocker or chaser hunters.
        – They have the minimum Manhattan distance to
          the target prey agent.
5:       Have each new chaser hunter inherit the prob-
      lem space of the tutor agent of the sub-grid.
6:       for each neighbouring grid do
7:          Choose m worker hunters from the current
      neighbouring grid to be blocker hunters that satisfy
      the following conditions:
        – They are not blocker or chaser hunters.
        – They have the minimum Manhattan distance to
          the target prey agent.
8:          Have each new blocker hunter inherit the
      problem space of the tutor agent of the sub-grid.
9:       end for
10:   end for
11: end for
```

Fig. 7 Hunting algorithm

the tutor hunters of the neighbouring sub-grids of the target prey's sub-grid (Line 8).

5.5 QA-Learning algorithm

The QA-learning algorithm comprises two learning stages:

- *First learning stage* In this stage, each worker agent copies its tutor's Q-table into its own Q-table and applies Q-learning to improve the tutor's solution. After each period of individual learning, the tutor aggregates its workers' Q-tables into its own Q-table using the Q-value sharing strategy of BEST-Q [17–19].
- *Second learning stage* This stage takes place at the end of the first stage. In this stage, the consultant agent incorporates the Q-tables of its tutors into its Q-table for the entire system.

Figure 8 shows the QA-learning algorithm. In this algorithm, lines 4–34 represent the first stage of QA-learning while lines 35–53 represent the second stage of QA-learning. In the first stage of the algorithm, each tutor assigns a copy of its Q-table to each one of its worker agents then the worker agents learn independently for a number of episodes to improve their individual solution (lines 4–21). After the end of the individual learning stage, each tutor aggregates the Q-tables of its worker agents into its own Q-table using BEST-Q (lines 22–30). If any tutor finishes learning, its identity is stored in the inactive list (lines 30–33) and its worker agents are reassigned to tutor agents that are still active. The

1: **Input**:
   – Number of tutors.
   – Number of workers assigned to each tutor i.
   – State set S_i, action set A_i, reward function R_i, connection set C_i, and initialised Q-table$_i$ of the subsystem's of each tutor i.
   – State set S_c, action set A_c, reward function R_c, and Q-table$_c$ of the whole system.
2: **Output**:
   – Optimised Q-table$_i$ for the subsystem of each tutor i.
   – Optimised Q-table$_c$ for the whole system.
3: **Begin**
4: **for** $i = 1$ to number of tutors **do**
5: **for** $j = 1$ to number of workers of tutor i **do**
6: Copy the Q-table of tutor i (Q_i) to the Q-table of worker j (Q_j)
7: **while** learning is not complete for tutor i **do**
8: Select s from S_i of tutor i using a policy derived from Q_j
9: **for** episode= 1 to p number of episodes **do**
10: **while** s is not a terminal state **do**
11: Choose an action a from the action set A_i using a policy derived from Q_j
12: Take action a
13: Receive r, s'
14: Update Q_j as follows:
15: $Q_j(s,a) \longleftarrow (1-\alpha_j)Q_j(s,a) + \alpha_j[R_j(s,a) + \gamma_j V_j(s')]$,
16: $V_j(s) \leftarrow max_{a \in A} Q_j(s,a)$, where $s \in S, a \in A$, the learning rate $\alpha_j \in [0,1]$ and the discount factor $\gamma_j \in [0,1]$.
17: $s \longleftarrow s'$
18: **end while**
19: **end for**
20: **end while**
21: **end for**
22: **for** $j = 1$ to number of workers of tutor i **do**
23: **for** $row = 1$ to number of rows of Q_i **do**
24: **for** $column = 1$ to number of columns of Q_i **do**
25: **if** $Q_j[row][column] > Q_i[row][column]$ **then**
26: $Q_i[row][column] = Q_j[row][column]$
27: **end if**
28: **end for**
29: **end for**
30: **end for**
31: **if** learning complete for tutor i **then**
32: add tutor i to the inactive list and migrate workers to another tutor
33: **end if**
34: **end for**
35: **for** $i = 1$ to number of tutors **do**
36: Append Q_i of tutor i to the consultant's Q-table Q_c using the connection set C_i of tutor i.
37: **end for**
38: **while** learning is not complete for consultant **do**
39: **for** $i = 1$ to number of tutors **do**
40: **for** each $C_i(s,a,s') \in C_i$ **do**
41: Set the initial state to be s
42: **while** s is not a terminal state **do**
43: Choose an action a from the action set A_c or C_i using a policy derived from Q_c
44: Take action a
45: Receive r, s'
46: Update the Q-table$_c$ as follows:
47: $Q_c(s,a) \longleftarrow (1-\alpha_c)Q_c(s,a) + \alpha_c[R_c(s,a) + \gamma_c V_c(s')]$,
48: $V_c(s) \leftarrow max_{a \in A} Q_c(s,a)$, where $s \in S_c, a \in A_c$ or $a \in C_i$, the learning rate $\alpha_c \in [0,1]$ and the discount factor $\gamma_c \in [0,1]$.
49: $s \longleftarrow s'$
50: **end while**
51: **end for**
52: **end for**
53: **end while**
54: **End**

Fig. 8 The QA-learning algorithm

first learning stage repeats until each tutor completes learning. In the second stage, the consultant aggregates the Q-table of each tutor agent into its Q-table using the connection sets of the tutors (lines 35–37). The consultant then attempts to improve its Q-table's solution by applying Q-learning to its Q-table (lines 38–53) starting the beginning of each episode from an initial state that belongs to the connection set of one of its tutor agents (lines 39–41). The consultant relearns its tutors' solutions in order to connect the tutors' Q-tables through a small number of exploratory learning steps.

6 Experiments

Two versions of the QA-learning algorithm were implemented: QA-learning with support for migration and QA-learning without migration. These two versions were compared with Q-learning, MAXQ, HEXQ and HAMQ-learning for different instances of the distributed hunter prey problem.

6.1 Setup

Three different grid sizes were selected to test the algorithms on small, medium, and large problems. In the first experiment, a grid size of 100×100 was used. The second experiment used a grid size of 200×200, and the third 500×500. Each experiment included 16 prey, with four prey distributed randomly in each quarter of the grid.

Two chaser hunters and three blocker hunters were assigned to each tutor. The Q-tables of each worker in the same role (chaser or blocker) were aggregated into the tutor's Q-table for that role after each 25 learning episodes.

The reward that each agent receives was defined as:

$$R(s,a) = \begin{cases} +100.0 & \text{if it reaches its goal} \\ 0 & \text{otherwise} \end{cases}$$

The learning parameters were set as follows:

– As suggested in [3,21], the learning rate $\alpha = 0.4$ and the discount factor $\gamma = 0.9$ for the Q-learning algorithm and the two learning stages of the QA-learning algorithm.
– As suggested in [16], the learning rate $\alpha = 0.25$ and the discount factor $\gamma = 1$ for HEXQ and MAXQ.
– As suggested in [28], the learning rate $\alpha = 0.25$ and the discount factor $\gamma = 0.999$ for HAMQ-learning.
– The selection policy of actions for all algorithms was the Softmax selection policy [31]:
  Given state s, an agent tries out action a with probability

$$p_s(a) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_{b=1}^{n} e^{\frac{Q(s,b)}{T}}}$$

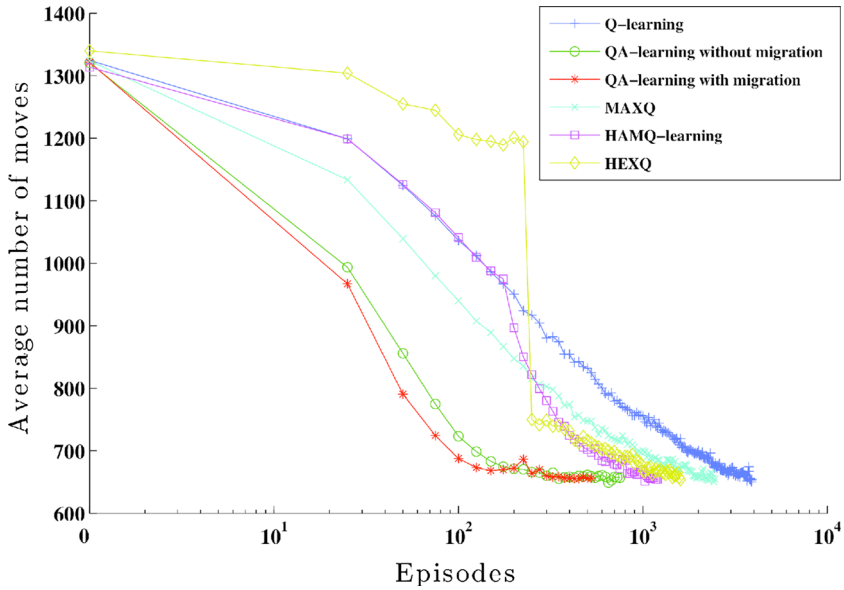

Fig. 9 Experiment 1: average number of moves per 25 episodes in a distributed hunter prey problem of a grid size 100×100

In the above equation, the temperature T controls the degree of exploration. Assuming that all Q-values are different, if T is high, the agent will choose a random action, but if T is low, the agent will tend to select the action with the highest weight. The value of T was chosen to be 0.7 to allow expected rewards to influence the probability while still allowing reasonable exploration.

For QA-learning, a parallel scheduling algorithm was used and each grid was split into four quarters (i.e. the 100×100 grid was split into four 50×50 sub-grids, the 200×200 grid into four 100×100 sub-grids, and the 500×500 grid into four 250×250 sub-grids. Decomposition of a problem into sub-problems is currently a manual process by the implementation designer. Thus, to demonstrate QA-learning, the decomposition of the problem space into four sub-grids was used for each problem size. While this is sufficient for an initial evaluation of QA-learning, the algorithm is in no way limited to any number of sub-problems and further research will be needed to determine optimal decompositions.

HAMQ-learning, HEXQ and MAXQ used two value functions: blocker and chaser value functions. The state variables for the chasing subtask are the position of the prey and the position the hunter while the state variables for the blocking subtask are the position of the blocker and the position of the blocking cell.

The learner in HEXQ explored the environment every 25 episodes and kept statistics on the frequency of change of each of the state variables. Each hierarchical exit is a state-action pair of the form (position of the goal, capture).

In all experiments, the position of each hunter agent was chosen randomly at the beginning of each episode. A learning episode ended when the hunter agents captured all the prey agents, or after 5000 moves without capturing all the prey agents. An algorithm is said to have converged when the average number of moves in its policy improves by less than one move over d consecutive episodes where $d = n/2$ for a grid size of $n \times n$.

6.2 Results and discussion

This section compares the performance of QA-learning with migration, QA-learning without migration, MAXQ, HEXQ, HAMQ-learning, and single agent Q-learning in the distributed hunter prey problem.

Figure 9 shows the average number of moves per 25 episodes required to capture all the prey agents in a distributed hunter prey problem of size 100×100. For QA-learning with migration, each tutor converges to a solution for its sub-grid after 175 episodes of learning, which marks the end of the first learning stage. The consultant converges to a solution after 375 episodes meaning that QA-learning converges after 550 episodes to a solution for the whole grid. On the other hand, QA-learning without migration converges after 750 episodes, MAXQ converges after 2450, HAMQ-learning converges after 1200, HEXQ converges after 1600 episodes, and basic Q-learning converges after 3900 episodes. These results suggest that the performance of QA-learning with migration is better than the other algorithms. This is because QA-learning allows multiple tutors to learn in parallel then the consultant combines their solutions through a small number of learning episodes. Even if tutors do not learn in parallel, the total number of learning episodes[1] required to converge to a solution ($175 \times 4 + 375 = 1075$), is only 27.6 % of

[1] Duration of the first stage of QA-learning $\times$ the number of tutors + duration of the second learning stage until the consultant convergence to a solution.

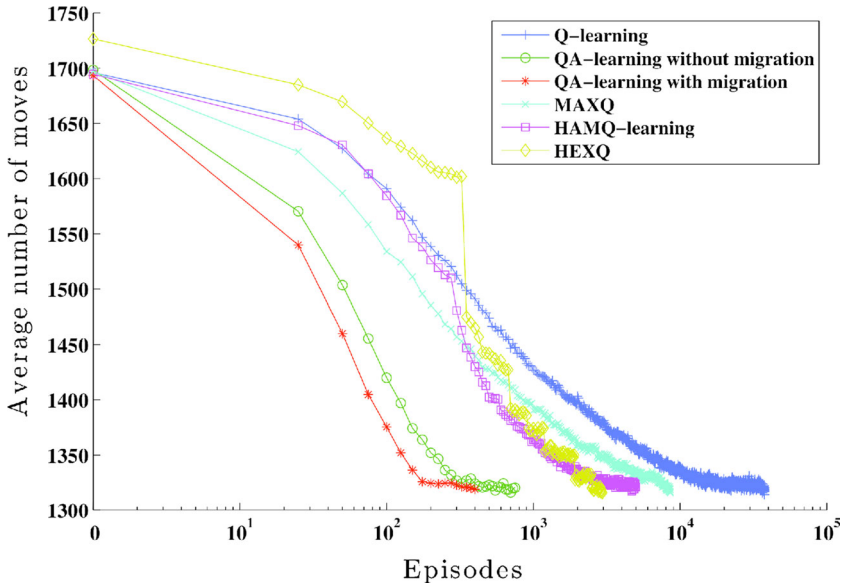

Fig. 10 Experiment 2: average number of moves per 25 episodes in a distributed hunter prey problem of a grid size 200×200

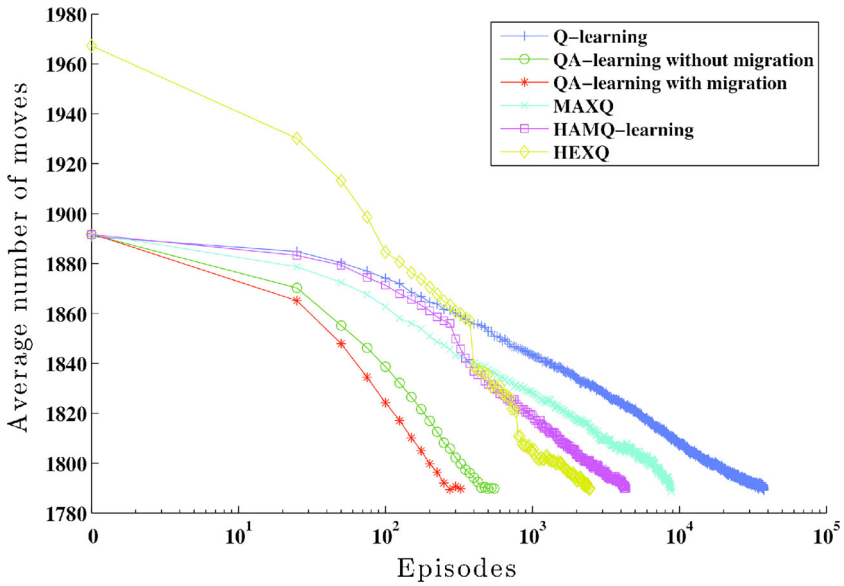

Fig. 11 Experiment 3: average number of moves per 25 episodes in a distributed hunter prey problem of a grid size 500×500

the number of episodes required for single agent Q-learning, (43.9 % for MAXQ, 67.2 % for HEXQ, and 89.6 % for HAMQ-learning. The support of migration of learners provided by QA-learning accelerates the learning process even faster as shown in the figure. Further, since the tutors in the QA-learning scenario are learning smaller subsets of the overall problem, their individual learning episodes are typically of shorter duration.

Figure 10 shows the average number of moves per 25 episodes required to capture all the prey agents in a distributed hunter prey problem of size 200×200. For QA-learning with migration, the tutors finish the first learning stage after 300 episodes, and the consultant converges to a solution after 400 episodes. Single agent Q-learning requires 37,525 episodes to converge to a solution. This is much more than the training time required for QA-learning. If QA-learning did not support parallel execution of learners, the total number of learning episodes required to converge to a solution would be $300 \times 4 + 100 = 1300$, only 3.5 % of the number of episodes required for single agent Q-learning (37,525), 15.2 % of MAXQ (8575), 26.1 % of HAMQ-learning (4975) and 44.1 % of HEXQ (2950).

Figure 11 shows the average number of moves per 25 episodes required to capture all the prey agents in a distributed hunter prey problem of size 500×500. In this experiment, QA-learning with migration converges to a solution after around 300 episodes (stage one: 225, stage two: 75). If the tutors at the first stage execute sequentially, the

Table 1 The ratio of the number of episodes in the cooperative learning algorithms to the number of episodes in Q-learning

	Experiment 1 (%)	Experiment 2 (%)	Experiment 3 (%)
Parallel QA-learning	14.1	1.1	0.8
Sequential QA-learning	27.6	3.5	2.59
MAXQ	58.3	22.85	23.4
HAMQ-Learning	30.8	13.3	11.5
HEXQ	41	7.9	6.6

Table 2 The running time of Q-learning vs the running time of the other algorithms in seconds

	Experiment 1	Experiment 2	Experiment 3
Q-Learning	583	68,662	509,417
QA-Learning without migration	167	1121	8409
QA-Learning with migration	160	683	4198
MAXQ	383	13,977	75,600
HAMQ-learning	232	8109	68,400
HEXQ	287	4808	21,047

total number of episodes required to converge to a solution is $225 \times 4 + 75 = 975$. The results in Fig. 11 show that Q-learning and the other cooperative Q-learning algorithms perform worse than QA-learning. The Q-learning algorithm converges to a solution after 37,550 episodes. This means that Q-learning requires almost 38 times the number of learning episodes required for QA-learning.

The overall results of the experiments suggest that QA-learning performs better than single agent Q-learning and the other cooperative Q-learning algorithms even in the sequential learning case and without the support of migration of learners. The performance difference in the parallel execution case of QA-learning became even larger as the task size increased (see row 1 of Table 1); in Experiment 1 QA-learning required 14.1 % of the episodes of single agent Q-learning, in Experiment 2 QA-learning required only 1.1 % of the episodes of single agent Q-learning, and in Experiment 3 QA-learning required only 0.8 % of the episodes of single agent Q-learning. Row 2 of Table 1 shows that sequential QA-learning reduces the required learning episodes of single agent Q-learning. This is because the learners in the first stage of QA-learning can quickly learn the smaller subsets of the original problem. This also means that the average length of an episode in QA-learning is shorter than the average length of an episode in single agent Q-learning.

The experiments were conducted using an Intel Xeon 3.4 GHz CPU with 16 GB RAM running 64-bit Windows. Table 2 shows the running time of the different algorithms for the three experiments in seconds. While the running time of both learning algorithms increases as the size of the problem increases, QA-learning is much faster for each of the three experiments. The smaller sub-problem size, combined with its parallel learning, makes QA-learning much more efficient.

7 Conclusion and future work

The hierarchical organisation of distributed systems provides an efficient decomposition of large problem spaces into more manageable components. This paper introduced the QA-learning algorithm for cooperative policy construction for independent learners that is based on three specialisations of agents: workers, tutors and consultants. Each consultant is responsible for assigning a sub-problem and a number of worker agents to each tutor. The worker agents first learn the problem space of their tutor, then the tutor aggregates its workers' Q-tables into its own Q-table. The consultant then merges the tutors' Q-tables to create its Q-table. Finally, the consultant performs a few rounds of Q-learning to optimise its Q-table.

The QA-learning algorithm has many advantages. First, the problem model of the QA-learning algorithm is a loosely coupled FMDP. This model reduces the complexity of large state space problems by taking advantage of the decomposable nature of the system itself.

Second, worker agents that have finished learning can be reassigned by the consultant to another tutor that is still learning to accelerate its learning process. This decreases the time required for the consultant agent to learn the entire system.

Finally, the results of the pilot experiments suggest that QA-learning performs faster than conventional Q-learning and other famous cooperative Q-learning algorithms, even if the tutors do not learn in parallel. Further, the average length of an episode in QA-learning is shorter than the average length of an episode in the other algorithms.

Currently, the decomposition process of QA-learning is the duty of the implementation designers. It goes hand in hand with the process of the design of distributed systems. This means that all decompositions need to be predefined

and have to be compatible with the distributed organisation of the system.

Future work includes the implementation of QA-learning in single goal hierarchical systems, the automatic identification of subsystems, the reusability of sub-solutions in QA-learning, and the applicability of QA-learning in partially observable environments

References

1. Abbeel, P., Ng, A.: Exploration and apprenticeship learning in reinforcement learning. In: Proceedings of the 22nd International Conference on Machine Learning, pp. 1–8 (2005)
2. Abed-Alguni, B.H.K.: Cooperative reinforcement learning for independent learners. Ph.D. thesis, The University of Newcastle, Australia. Faculty of Engineering and Built Environment, School of Electrical Engineering and Computer Science (2014)
3. Arai, S., Sycara, K.: Effective learning approach for planning and scheduling in multi-agent domain. In: Proceedings of the 6th International Conference on Simulation of Adaptive Behavior, pp. 507–516 (2000)
4. Asadi, M., Huber, M.: State space reduction for hierarchical Reinforcement Learning. In: Proceedings of the Seventeenth International FLAIRS Conference, pp. 509–514 (2004)
5. Barto, A.G., Mahadevan, S.: Recent advances in hierarchical reinforcement learning. Discrete Event Dyn. Syst. **13**(1–2), 41–77 (2003)
6. Boutilier, C., Dearden, R., Goldszmidt, M.: Exploiting structure in policy construction. In: International Joint Conference on Artificial Intelligence, vol. 14, pp. 1104–1113. Lawrence Erlbaum Associates Ltd (1995)
7. Boyd, T., Dasgupta, P.: Process migration: a generalized approach using a virtualizing operating system. In: Proceeding of the 22nd International Conference on Distributed Computing Systems ICDCS 2002, pp. 385–392 (2002)
8. Cai, Y., Yang, S., Xu, X.: A combined hierarchical reinforcement learning based approach for multi-robot cooperative target searching in complex unknown environments. In: 2013 IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning (ADPRL). Singapore, pp. 52–59 (2013)
9. Cao, F., Ray, S.: Bayesian hierarchical reinforcement learning. In: F. Pereira, C. Burges, L. Bottou, K. Weinberger (eds.) Advances in Neural Information Processing Systems, vol. 25, pp. 73–81. Curran Associates (2012)
10. Daoui, C., Abbad, M., Tkiouat, M.: Exact decomposition approaches for Markov decision processes: a survey. In: Advances in Operations Research 2010 (2010)
11. Dietterich, T.G.: Hierarchical reinforcement learning with the MAXQ value function decomposition. J. Artif. Intell. Res. **13**(1), 227–303 (2000)
12. Erus, G., Polat, F.: A layered approach to learning coordination knowledge in multiagent environments. Appl. Intell. **27**(3), 249–267 (2007)
13. Ghavamzadeh, M., Mahadevan, S., Makar, R.: Hierarchical multi-agent reinforcement learning. Auton. Agents Multi-Agent Syst. **13**(2), 197–229 (2006)
14. Guestrin, C., Gordon, G.: Distributed planning in hierarchical factored MDPs. In: Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence, pp. 197–206. Morgan Kaufmann (2002)
15. Gunady, M.K., Gomaa, W., Takeuchi, I.: Aggregate reinforcement learning for multi-agent territory division: the hide-and-seek game. Eng. Appl. Artif. Intell. **34**, 122–136 (2014)
16. Hengst, B.: Discovering hierarchy in reinforcement learning with HEXQ. In: Machine Learning: Proceedings of the Nineteenth International Conference on Machine Learning, pp. 243–250. Morgan Kaufmann (2002)
17. Iima, H., Kuroe, Y.: Reinforcement learning through interaction among multiple agents. In: The 2006 International Joint Conference of the Japanese Society of Instrument and Control Engineers and the Korean Institute of Control, Automation and System Engineers, pp. 2457–2462 (2006)
18. Iima, H., Kuroe, Y.: Swarm reinforcement learning algorithms—exchange of information among multiple agents. In: 2007 Annual Conference of the Japanese Society of Instrument and Control Engineers, pp. 2779–2784 (2007)
19. Iima, H., Kuroe, Y.: Swarm reinforcement learning algorithms based on sarsa method. In: 2008 Annual Conference of the Japanese Society of Instrument and Control Engineers, pp. 2045–2049 (2008)
20. Jardim, D., Nunes, L., Oliveira, S.: Hierarchical reinforcement learning: learning sub-goals and state-abstraction. In: 2011 6th Iberian Conference on Information Systems and Technologies. Chaves, Portugal, pp. 1–4 (2011)
21. Jiang, D.W., Wang, S.Y., Dong, Y.S.: Role-based context-specific multiagent Q-learning. Acta Autom. Sinica **33**(6), 583–587 (2007)
22. Kaye, D.: Loosely coupled: the missing pieces of Web services. In: Bing, A., Kaye, C. (eds.) 1st edn. Chap. 10, RDS Strategies LLC p. 132 (2003)
23. Kim, K.E., Dean, T.: Solving factored MDPs via non-homogeneous partitioning. Proceedings of the 17th International Joint Conference on Artificial Intelligence. IJCAI'01, vol. 1, pp. 683–689. Morgan Kaufmann, San Francisco (2001)
24. Lee, M.R.: A multi-agent cooperation model using reinforcement learning for planning multiple goals. J. Secur. Eng. **2**(3), 228–233 (2005)
25. Liu, F., Zeng, G.: Multi-agent cooperative learning research based on reinforcement learning. In: G. Weiß (ed.) The 10th International Conference on Computer Supported Cooperative Work in Design, pp. 1–6 (2006)
26. Mausam, Weld, D.S.: Solving concurrent Markov decision processes. In: Proceedings of the 19th National Conference on Artificial Intelligence, pp. 716–722. AAAI Press (2004)
27. Ono, N., Fukumoto, K.: A modular approach to multi-agent reinforcement learning. In: Weiß, G. (ed.) Distributed Artificial Intelligence Meets Machine Learning Learning in Multi-Agent Environments. Lecture Notes in Computer Science, vol. 1221, pp. 25–39. Springer, Berlin (1997)
28. Parr, R., Russell, S.: Reinforcement learning with hierarchies of machines. In: Advances in Neural Information Processing Systems, vol. 10, pp. 1043–1049. MIT Press (1997)
29. Strösslin, T., Gerstner, W.: Reinforcement learning in continuous state and action space. Artif. Neural Netw. ICANN **2003**, 4 (2003)
30. Sutton, R., Precup, D., Singh, S.: Between MDPs and Semi-MDPs: a framework for temporal abstraction in reinforcement learning. Artif. Intell. **112**, 181–211 (1999)
31. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)

32. Tosic, P.T., Vilalta, R.: A unified framework for reinforcement learning, co-learning and meta-learning how to coordinate in collaborative multi-agent systems. Proc. Comput. Sci. **1**(1), 2217–2226 (2010)
33. Vasudevan, N., Venkatesh, P.: Design and implementation of a process migration system for the Linux environment. In: 3rd International Conference on Neural, Parallel and Scientific Computations. Atlanta, USA (2006)
34. Watkins, C.: Learning from delayed rewards. Ph.D. thesis, Cambridge University, Cambridge, England (1989)
35. Watkins, C., Dayan, P.: Technical Note: Q-Learning. Mach. Learn. **8**(3), 279–292 (1992)
36. Wu, B., Feng, Y., Zheng, H.: Model-based bayesian reinforcement learning in factored Markov decision process. J. Comput. **9**(4), 845–850 (2014)
37. Yong, C., Miikkulainen, R.: Coevolution of role-based cooperation in multiagent systems. IEEE Trans. Auton. Ment. Dev. **1**(3), 170–186 (2009)