A Genetic Algorithm for Automatically Designing Modular Reinforcement Learning Agents

Isao Ono
Faculty of Engineering,
University of Tokushima,
2-1, Minami-Josanjima,
Tokushima, 770-8506, JAPAN.
email: isao@is.tokushima-u.ac.jp
phone: +81-88-656-9139

Tetsuo Nijo
Faculty of Engineering,
University of Tokushima,
2-1, Minami-Josanjima,
Tokushima, 770-8506, JAPAN.
email: niji@is.tokushima-u.ac.jp
phone: +81-88-656-9139

Norihiko Ono
Faculty of Engineering,
University of Tokushima,
2-1, Minami-Josanjima,
Tokushima, 770-8506, JAPAN.
email: ono@is.tokushima-u.ac.jp
phone: +81-88-656-7509

Abstract

Reinforcement learning (RL) is one of the machine learning techniques and has been received much attention as a new self-adaptive controller for various systems. The RL agent autonomously learns suitable policies via the clue of reinforcement signals by trial and error. Most RL methods have a serious problem that computational resources and time to learn appropriate policies grow rapidly as the size of the problem space increases. To overcome the problem, the Modular Reinforcement Learning Architecture (MRLA) has been proposed and shown good results. However, the performance of an agent with MRLA deteriorates unless the agent consists of appropriate modules for a given task, which cannot be predicted in advance. This means that a human designer has to identify a good combination of modules by trial and error. In this paper, we propose a genetic algorithm for finding appropriate combination of modules and show its effectiveness by applying it to a benchmark problem.

1 INTRODUCTION

Reinforcement learning (RL) is one of the machine learning techniques and, recently, has been received much attention as a new self-adaptive controller for various systems such as robots and plants. The RL agent autonomously learns suitable policies, mappings from states to actions, via the clue of reinforcement signals, rewards, by trial and error. Most reinforcement learning methods have a serious problem that computational resources and time to learn appropriate policies grows rapidly as the size of the problem space increases.

To remedy the above problem, Whitehead *et al.* proposed the modular reinforcement learning architecture (MRLA) [Whitehead 93]. The basic concept in MRLA is that the modular architecture has multiple modules which receive the subsets of the sensory inputs to an agent to reduce the state space. This means that the whole problem space is decomposed into some smaller subproblem spaces, which are assigned to corresponding RL modules, respectively, and each RL module takes charge of a part of the whole problem. Ono *et al.* applied MRLA to some multi-agent problems, to which most RL methods are difficult to apply because of their large problem space, and demonstrated its effectiveness [Ono 96].

However, there are no guidelines to design a set of modules, i.e. the number of modules and the combination of modules in the set. A human designer has to design a set of modules for a given task by trial and error since an appropriate set of modules depends on the task. Some attempts have been made to remedy this problem. Kohri *et al.* proposed a heuristic method, named the Automatic Modular Q-learning (AMQL), for enabling agents to obtain a suitable set of modules by themselves [Kohri 97]. A serious problem of AMQL is that the number of modules is fixed and has to be given in advance though we cannot predict an appropriate number of modules, which depends on a given problem. Yoshida *et al.* proposed a hill climbing method for automatically finding an appropriate set of modules [Yoshida 99]. In this method, the number of modules is variable. However, this method possibly falls into local optima because it is local search.

In this paper, we propose a genetic algorithm (GA) for finding an appropriate set of modules for a given problem, which is named the Genetic Modular Q-Learning (GMQL). In GMQL, the number of modules is variable. Further, GMQL is expected to find globally good sets of modules because it is based on GAs. To show the effectiveness of GMQL, we compare the perfor-

mance of GMQL with that of AMQL and of the hill climbing method on a well-known benchmark problem called the Pursuit problem [Benda 85]. Further, we compare the performance of the agent designed by G-MQL with the one by a human designer [Ono 96].

In section two, we briefly explain one of the most popular RL method, Q-learning [Watkins 92], and the modular learning architecture with Q-learning, Modular Q-learning. We also show an example of applying Modular Q-learning to some multi-agent problems by Ono *et al.*. Section three reviews some conventional methods for automatically finding an appropriate set of modules for a given problem. We propose the Genetic Modular Q-Learning (GMQL) in section four. In section five, we compare GMQL with AMQL [Kohri 97], the hill climbing method [Yoshida 99] and the human designer [Ono 96] on the Pursuit problem, and discuss the results. Section six is conclusions.

2 MODULAR REINFORCEMENT LEARNING

2.1 Reinforcement Learning

Reinforcement learning (RL) is a learning framework to adapt an agent to the environment by trial and error. Being different from supervised learning, the RL agent does not have a supervisor that tells correct actions in response to state inputs explicitly. Instead, the RL agent learns suitable action policies, mappings from state inputs to action outputs, via the clues of reinforcement signals, rewards, from the environment. The reward is a scalar with noise and is given to an agent after several state transitions, which means that the agent receives rewards from the environment for its action sequences. Since what a designer tells the agent is only the goal by designing rewards instead of how to achieve the goal, the RL has received much attention as a new self-adaptive controller for various systems such as robots and plants.

2.2 Q-learning

Q-learning [Watkins 92] is one of the most popular RL methods. In Q-learning, an agent learns the Q-function, $Q(s,a)$, that gives the estimated value of the action a at the state s. The algorithm of Q-learning is as follows:

1. At time t, the agent observes the environment and gets a state s.
2. The agent performs an action a according to an action-selecting strategy.
3. The agent receives a reward r from the environment.

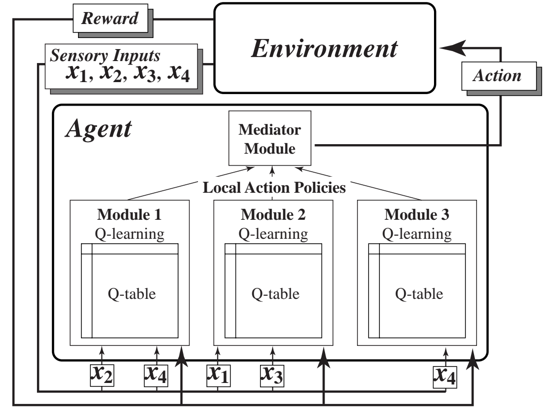


Figure 1: Modular Q-learning Architecture (MQLA)

4. The Q-value $Q(s,a)$ is updated as follows: $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a')]$, where $\alpha(0 < \alpha \leq 1)$ is the learning rate and $\gamma(0 \leq \gamma < 1)$ is the discount rate.
5. $t \leftarrow t+1$. Go to Step 1.

After the above algorithm is repeated sufficient times, a suitable policy can be obtained by choosing the action a at the state s where $Q(s,a)$ is the maximum Q-value at the state s.

The simplest way to express the Q-function is to make a Q-table that contains each Q-value for every pair of state and action. However, as the size of the problem space increases, this takes more computational resources and requires much more time for learning.

2.3 Modular Q-learning

The modular Q-learning architecture (MQLA) [Whitehead 93] was proposed to remedy the problem of Q-learning described in previous section. As shown in Fig. 1, MQLA has multiple modules which receive the subsets of the sensory inputs to an agent. This means that the whole problem space is decomposed into some smaller subproblem spaces, which are assigned to corresponding modules, respectively, and each module takes charge of a part of the whole problem. Each module has its own subgoal and decides its action policy according to its subgoal. The mediator module is used to mediate the global action with a certain strategy (decision by majority, etc.). Each module other than the mediator module includes a Q-table for the pairs of partial states and actions. Modules learn to achieve their corresponding goals through updating their Q-values. Thus, this architecture makes the Q-table smaller and improves learning time for problems.

Ono *et al.* successfully applied MQLA to one of the famous multi-agent problems called the Pursuit problem

[Ono 96] (see section 5.1). In [Ono 96], each hunter agent has three modules and each module takes the relative positions of the prey agent and of one of the other hunter agent as its sensory input. For example, suppose that each hunter agent has a 5×5 limited visual field. In this case, an agent with a monolithic Q-table needs enormous memory resource since the possible number of states that the agent may encounter is $(5^2+1)^4 = 456,976$. On the other hand, the above modular agent [Ono 96], in which each module takes the positions of only two agents as sensory inputs, needs much smaller size of memory because the possible number of states decreases to $(5^2+1)^2 = 676$. Through some experiments in the Pursuit problem, Ono *et al.* showed that agents with MQLA not only solved the state space problem but also succeeded in learning appropriate cooperative behavior.

To successfully apply MQLA to a given problem, a human designer has to appropriately decompose the state space and design a suitable set of modules which receive the subsets of sensory inputs to the agent. However, there are no guidelines to design a set of modules, i.e. the number of modules and the combination of modules in the set. A human designer has to design a set of modules for every given task by trial and error since the set of modules depends on the task.

3 PREVIOUS WORK

In this section, we review two conventional methods to remedy the problem of MQLA described in previous section. These methods attempt to automatically design a suitable set of modules for a Modular Q-learning agent.

3.1 Automatic Modular Q-Learning (AMQL)

Kohri *et al.* proposed a heuristic method named the Automatic Modular Q-Learning (AMQL) [Kohri 97]. This is the first attempt to automatically find an appropriate set of modules. The algorithm of AMQL is as follows:

1. Initialize the agent which has l modules, where l is a constant given by a designer in advance. For each module, the number and the combination of sensory inputs is randomly chosen and, then, a Q-table and initial Q-values are set.
2. Set the fitness of all modules to 0.
3. Set the execution counter t of agent to 0.
4. Set the agent to initial states.
5. The agent receives the current state s.
6. If the state s is the goal state, then $t \leftarrow t+1$ and go to 10.
7. The agent chooses its own action by the GM strategy [Whitehead 93] and execute it.
8. Update the Q-tables of each module.
9. If the agent receives a reward with action a, then increment the fitness of the module whose Q-value of action a is the largest in every action: $fitness \leftarrow fitness + 1$. Go to 5.
10. If $t < T$, then go to 4, where T is the period of select given by a designer.
11. Calculate a fitness threshold k which is calculated by $k = (the\ number\ of\ reward\ acquision) \times p$, where $p (0 < p < 1)$ is a constant given by a designer.
12. For the modules whose fitness values are smaller than the threshold, choose the number and the combination of sensory inputs randomly and , then, set Q-tables and initial Q-values. Go to 2.

In AMQL, a human designer has to determine the number of modules in advance. However, we cannot predict an appropriate number of modules, which depends on a given task. It is also a problem that only the modules which output action a directly attributing to rewards increase their fitness values because modules should be evaluated based on not a single action but a sequence of multiple actions in reinforcement learning problems.

3.2 Hill Climbing for Modular Q-Learning (HCMQL)

Yoshida *et al.* proposed a hill climbing method for finding a suitable set of modules. In this paper, we call this method the Hill Climbing for Modular Q-Learning (HCMQL). The algorithm of HCMQL is as follows:

1. Initialize the agent with a set of modules M_0. Set the number of modules in M_0 to zero. Set the evaluation value $E(M_0) = -\infty$.
2. Apply the *insertion* operator to M_0, let the result be M_1, and evaluate $E(M_1)$. Here, evaluating $E(M_1)$ means measuring the performance of the agent with the set of modules M_1 by actually applying it to a given problem.
3. Apply the *removal* operator to M_1, let the result be M_2, and evaluate $E(M_2)$.
4. $M_0 = M_{arg\max_{i=1,2,3} E(M_i)}$.
5. If a stop condition is satisfied, then terminate the search, otherwise go to step 2.

The *insertion* operator is an operator that adds some modules to the set of module as follows:

1. Let the copy of M_0 be M_1.
2. Determine the number of modules to be added $n_{\text{insertion}}$ randomly.
3. Randomly generate n_{insetion} modules which M_1 does not contain and add them to M_1.

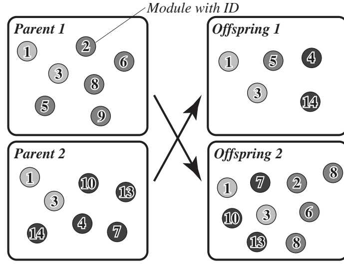


Figure 2: Module Exchange Crossover (MXX) ($n_1 = 4$, $n_2 = 2$)

The *removal* operator is an operator that removes some modules from the set of module as follows:

1. Set the set of modules $M_2 = \phi$.
2. Choose a module from M_1 randomly.
3. Make a set that consists of the module chosen in step 2 and its upper and lower modules in M_1. Let the set be T and $M_1 - T$ be M_1. Here, a module L_l is the upper (lower) module of a module L_m if $i(L_l) \supset i(L_m)$ $(i(L_l) \subset i(L_m))$, where $i(L_k)$ represents the set of sensory inputs to the module L_k.
4. With a probability $|T|/(|T|+1)$, choose a module randomly from T and add it to M_2. Otherwise, do nothing.
5. If $M_1 = \phi$, then terminate the algorithm. Otherwise, goto step 2.

In HCMQL, the number of modules in an agent is variable and the sequences of actions are evaluated. However, HCMQL may find bad local optima because it is local search.

4 Genetic Modular Q-Learning (GMQL)

In this section, we propose a GA for finding appropriate combination of modules, named the Genetic Modular Q-Learning (GMQL), to overcome the problems of the conventional methods.

4.1 Representation

We employ a set of modules whose size is variable as a representation. Modules are labeled with IDs so that the modules which take the same sensory inputs have the same ID. So, a chromosome is represented as a set of IDs, e.g. $\{6, 8, 9, 11, 12, 13\}$. Here, a chromosome is not permitted to include more than two same modules.

4.2 Crossover Operator

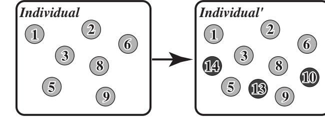


Figure 3: Mutation: Insertion Operator

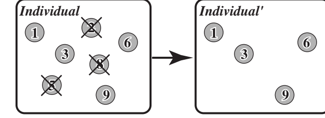


Figure 4: Mutation: Removal Operator

We propose the Module eXchange Crossover (MXX), which exchanges some modules between two parental set of modules, as shown in Fig. 2. The algorithm of MXX is as follows:

1. Let two parental sets of modules be P_1 and P_2, respectively.
2. Let the set of the modules which belong to both of P_1 and P_2 be C. If there are no modules which belong to both of P_1 and P_2, then set $C = \phi$.
3. Delete the elements of C from P_1 and P_2: $P_1 \leftarrow P_1 - C$ and $P_2 \leftarrow P_2 - C$.
4. Let two offspring sets of modules be O_1 and O_2, respectively, and initialize them: $O_1 = C$ and $O_2 = C$.
5. Randomly determine the numbers of the elements which are copied from P_1 to O_2 and from P_2 to O_1, n_1 and n_2: $n_1 \sim u(1, |P_1|)$ and $n_2 \sim u(1, |P_2|)$, where $u(x, y)$ is a uniform distribution with the range between x and y.
6. Randomly choose n_1 modules from P_1 and n_2 modules from P_2, respectively. Let the set of the modules chosen from P_1 be S_1 and that from P_2 be S_2, respectively.
7. Delete the elements of S_1 from P_1 and those of S_2 from P_2: $P_1 \leftarrow P_1 - S_1$ and $P_2 \leftarrow P_2 - S_2$.
8. Add the elements of S_1 to O_2 and those of S_2 to O_1: $O_1 \leftarrow O_1 + S_2$ and $O_2 \leftarrow O_2 + S_1$.
9. Add the rest elements of P_1 to O_1 and those of P_2 to O_2: $O_1 \leftarrow O_1 + P_1$ and $O_2 \leftarrow O_2 + P_2$.

4.3 Mutation Operator

In this paper, we introduce two mutation operators, the *insertion* operator and the *removal* operator.

4.3.1 Insertion Operator

The insertion operator adds some modules generated randomly to a set of modules as shown in Fig. 3. The algorithm of the operator is as follows:

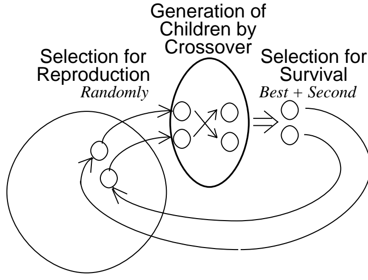


Figure 5: Generation-alternation Model

1. Let the target set of modules be X.
2. Randomly determine the number of the modules to be added to X: $n_{\text{insetion}} \sim u(1, N-|X|)$, where $u(x, y)$ is a uniform distribution with the range between x and y and N is the max of module ID.
3. Randomly generate $n_{\text{insertion}}$ modules which are not the members of X and add them to X.

4.3.2 Removal Operator

The removal operator deletes some modules from a set of modules as shown in Fig. 4. The algorithm of the operator is as follows:

1. Let the target set of modules be X.
2. Randomly determine the number of the modules to be removed from X: $n_{\text{removal}} \sim u(1, |X|)$, where $u(x, y)$ is a uniform distribution with the range between x and y.
3. Randomly choose n_{removal} modules from X and remove them from X.

4.4 Algorithm of GMQL

In this section, we design the algorithm of GMQL. In this paper, we employ a variant of the Minimal Generation Gap (MGG) [Satoh 96] shown in Fig. 5 as a generation-alternation model since MGG has shown to have an excellent capability of maintaining a diversity of the population.

The algorithm of GMQL is as follows:

1. *Initialization*
   Generate n_{pop} individuals randomly.
2. *Selection for Reproduction*
   Choose two individuals randomly to be parents from the population.
3. *Generation of Offspring*
   Apply MXX to the parents with a probability of p_c to generate two offspring. If MXX is not applied, the offspring are the copies of the parents. Then, apply the mutation operator, which is randomly chosen from the insertion operator and the removal operator, to the offspring with a probability of p_m.
4. *Evaluation of Offspring*
   Apply the agent with the set of modules specified by the offspring to a given learning problem and let the agent learn for a certain learning time, LT trials. A trial is terminated when the agent achieves the goal, e.g. capturing the prey in the Pursuit problem, or when the time step, that is the number of agent outputting actions, reaches MAX_{step} steps. After the learning period, evaluate the agent for a certain evaluation time, ET trials. Let the evaluation value of the agent be that of the corresponding offspring.
5. *Selection for Survival*
   Select two best individuals from the family consisting of the parents and their offspring and replace the parents in the population with the selected individuals.
6. Repeat the above procedures from Step 2 to Step 5 until a certain stop condition is satisfied.

The features of the proposed method are as follows:

- The size of the set of modules is variable.
- The sequence of actions are evaluated.
- Global search can be realized.

Therefore, we expect that GMQL can find better solutions than AMQL and HCMQL.

5 EXPERIMENTS

To demonstrate the effectiveness of GMQL, we compared GMQL with AMQL [Kohri 97] and HCMQL [Yoshida 99]. The benchmark problem used here is the modified version of the Pursuit Problem [Benda 85] which was used in [Ono 96]. We also compared the performance of the set of modules found by GMQL with that designed by a human designer [Ono 96].

5.1 Pursuit Problem [Benda 85] [Ono 96]

In an $n \times n$ toroidal grid world, initially, a single *prey* and four *hunter* agents are placed at random positions respectively, as shown in Fig. 6 (a). The final goal of the hunters is to capture the prey. The prey is judged to be captured when all of its four neighbor positions are occupied by the hunters, as shown in Fig. 6 (b).

At every time step, each hunter independently chooses their own actions, not communicating with each other but using only sensory inputs. The prey chooses actions randomly. A hunter has a $n_{\text{vf}} \times n_{\text{vf}}$ limited visual field, e.g. $n_{\text{vf}} = 3$ in Fig. 6 (a), and has a unique identifier. A hunter can recognize the identifier of any other agents within its visual field.

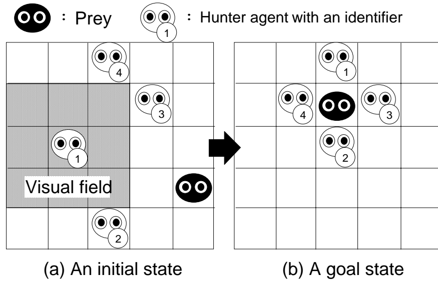


Figure 6: Pursuit Problem. A hunter is able to accurately recognize the relative position and the identifier of any other agents in its visual field. In Fig. (b), the prey is captured by the hunters.

Table 1: Summary of modules : The sensory input consists of four bits, (b_1, b_2, b_3, b_4). b_1, b_2, b_3 and b_4 correspond to the sensors observing the prey, the nearest hunter agent, the second nearest hunter agent and the farthest hunter agent, respectively, where one (zero) means that the corresponding sensory input is (not) used.

ID	0	1	2	3	4
Sensory Inputs	0001	0010	0100	1000	0011
ID	5	6	7	8	9
Sensory Inputs	0101	1001	0110	1010	1100
ID	10	11	12	13	
Sensory Inputs	0111	1011	1101	1110	

The sensory inputs to an agent are the relative positions to the prey and other hunter agents in the visual field. If the prey and/or some other hunter agent(s) do not exist in the visual field, the corresponding sensory inputs indicate that they do not exist in the visual field.

Each agent has a repertoire of five actions, that is, they can move in one principle direction (north, east, south or west), or remain at the current position. The prey can not occupy the same position that a hunter does, while hunters can occupy the same position.

5.2 Designing Modules and How to Evaluate Individuals

In the following experiments, we assume that all four hunter agents consist of the same set of modules specified by an individual. In this problem, the number of possible combinations of sensory inputs to a hunter agent is 16 because a hunter agent has four sensory inputs observing the relative positions of other three hunter agents and a prey agent respectively. Among these combinations, the cases of no sensory inputs and of all sensory inputs are not useful from a view point of modular approaches. Thus, we employ the 14 modules summarized in Table 1.

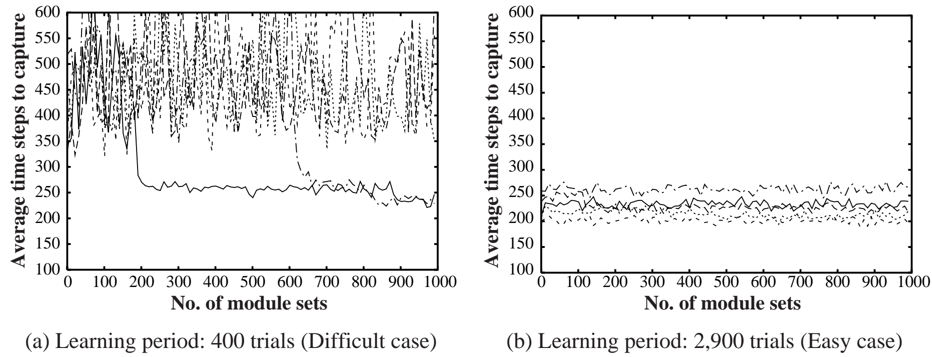
The evaluation value of an individual is given by the average time steps in which the hunter agents with the set of modules specified by the individual capture the prey over the evaluation period (see section 4.4.). We set the evaluation time ET to 100, the learning time LT to 400 (difficult case) or 2,900 (easy case) and the maximum time step in each trial MAX_{step} to 1,500, respectively.

5.3 Comparison of GMQL with AMQL and HCMQL

We compare the search performance of GMQL with that of AMQL and HCMQL. In this experiment, we set the size of the environment to 20×20 and n_{vf} for the visual field of agent to five. For parameters of AMQL, we set the number of modules l to 10 and p for calculating the fitness threshold to 0.9. For parameters of GMQL, we set the population size to 50, $p_c = 0.8$ and $p_m = 0.05$. We perform five independent runs for each methods. Figure 7, 8 and 9 show the convergence curves of AMQL, HCMQL and GMQL, respectively, when the learning time, LT, is 400 and 2,900. Each curve is the transition of the best evaluation value in the population during a single run. In each graph, the horizontal axis shows the number of newly generated sets of modules and the vertical axis the average time steps to capture. These results show that GMQL shows better performance than the conventional methods, AMQL and HCMQL.

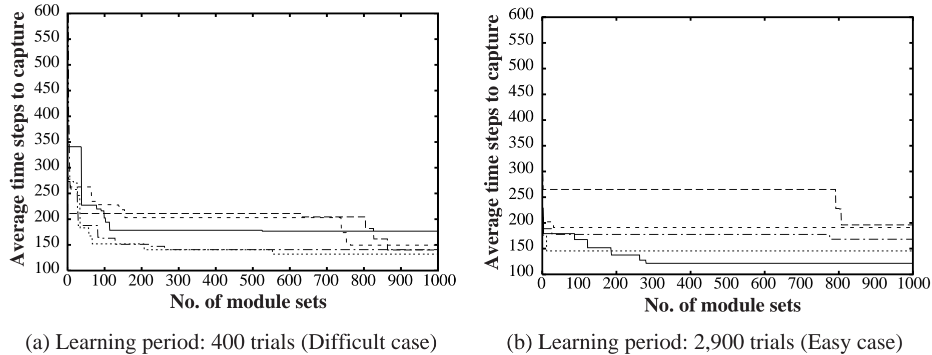
As shown in Fig. 7, AMQL fails to find good sets of modules and the performance is poor. It is thought that the reason is that the number of modules is fixed and modules are not evaluated based on the sequences of actions.

From Fig. 7 and 8, HCMQL shows better performance than AMQL. We believe that the reason of the better performance of HCMQL is that the number of modules is not fixed and modules are evaluated based on the sequences of actions. However, the variance of the final evaluation values, which are obtained at the time when the number of module sets generated is 1,000, is large, which means that the HCMQL found bad local optima. HCMQL obtained worse evaluation values in the case where learning time is 2,900 trials (Fig. 8 (b)) than it did in the case where learning time is 400 trials (Fig. 8 (a)) though the performance of an agent which learns for long period should be better than that which does for short period. This shows that the number of bad local optima grows as the learning period increases. We think that this is because redundant module sets can obtain relatively good policies if the learning period is long. In the contrast to HCMQL, GMQL succeeds in obtaining good evaluation values in all runs not only when learning time is 400 but also when learning time is 2,900 as shown in Fig. 9. This



(a) Learning period: 400 trials (Difficult case) (b) Learning period: 2,900 trials (Easy case)

Figure 7: Convergence curves : AMQL [Kohri 97]



(a) Learning period: 400 trials (Difficult case) (b) Learning period: 2,900 trials (Easy case)

Figure 8: Convergence curves : HCMQL [Yoshida 99]

shows the excellent search ability of GMQL.

5.4 Comparison of GMQL with A Human Designer [Ono 96]

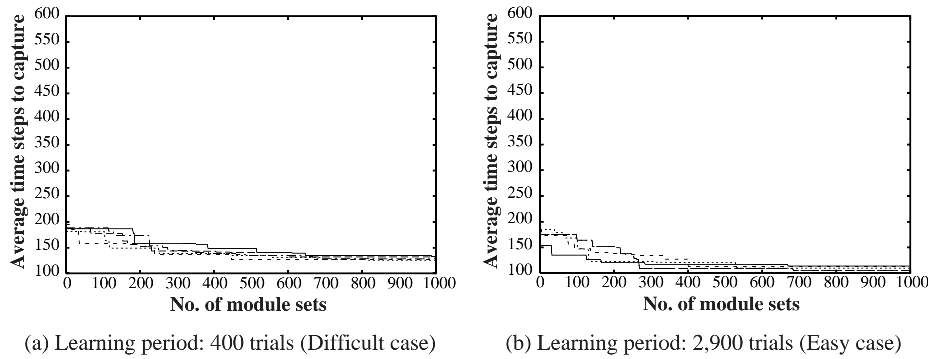
We compare the learning performance of the agent with the set of modules found by GMQL and that designed by a human designer [Ono 96]. In this experiment, n_{vf} for the visual field of agent is seven and the parameters for GMQL is the same in previous section. Figure 10 shows the learning curves of the modular structures designed by a human designer [Ono 96] and found by GMQL with the learning period of 2,900 and 400 trials. The horizontal axis shows the number of trials and the vertical one does the average time steps to capture.

As shown in Fig. 10, the average time steps of [Ono 96] fall slowly while that of the modular structures found by GMQL does very quickly. We believe that the reason for the good performance of the modular structures found by GMQL is that they have the module that observes only the prey as shown in Fig. 11. The module is not included in the modular structure designed by a human designer. We confirmed that the modular structure made by adding the module observing only the prey to the modular structure designed by a human designer showed almost the same performance as the modular structures found by GMQL.

6 CONCLUSIONS

In this paper, we proposed GMQL, which is a genetic algorithm to automatically find appropriate modular structures for a given task in MRLA. Using one of the famous benchmark problem named the Pursuit problem, we showed not only that the search performance of GMQL outperformed that of the conventional methods, AMQL [Kohri 97] and HCMQL [Yoshida 99], but also that the agent with the module set found by G-MQL performed better than the one designed by a human designer [Ono 96].

At the moment, GMQL optimizes only a single criterion, the average time steps to capture. Now, we are extending GMQL to be able to optimize two criteria, the average time steps to capture and the memory resources, at the same time. We have a plan to apply GMQL to other problems and confirm the effectiveness of GMQL. There are some other techniques which combine machine learning techniques and evolutionary computation such as [Miagkikh 99]. We would like to



(a) Learning period: 400 trials (Difficult case)   (b) Learning period: 2,900 trials (Easy case)

Figure 9: Convergence curves : GMQL (Proposed method)

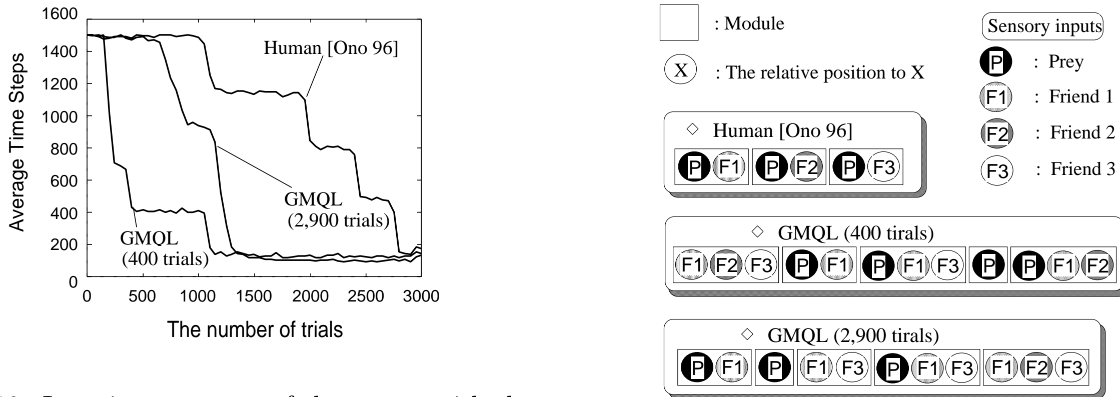


Figure 10: Learning processes of the agents with the module sets found by GMQL (learning period: 400 trial (difficult case), 2,900 trials (easy case)) and designed by a human designer [Ono 96]

Figure 11: Examples of the modular structures designed by a human designer [Ono 96] and found by GMQL

investigate possibilities of ML+EC approaches for future work.

References

[Benda 85] Benda, M., Jagannathan, V. and Dodhiawalla, R.: On Optimal Cooperation of Knowledge Sources, Technical Report BCS-G2010-28, Boeing AI Center, 1985.

[Ono 96] Ono, N. and Fukumoto, K.: Multi-agent Reinforcement Learning: A Modular Approach, Proc. 2nd Int'l Conf. on Multi-agent Systems, pp.252-258 (1996).

[Kohri 97] Kohri, T., Matsubayashi, K. and Tokoro, M.: An Adaptive Architecture for Modular Q-learning, Proc. 15th Int'l Joint Conf. on Artificial Intelligence, pp.820-825 (1997).

[Miagkikh 99] Miagkikh, V. V. and Punch, W. F.: An Approach to Solving Combinatorial Optimization Problems Using a Population of Reinforcement Learning Agents, Proc. 1999 Genetic and Evolutionary Computation Conf. (GECCO'99), pp.1358-1365 (1999).

[Satoh 96] Satoh, H., Yamamura, M. and Kobayashi, S.: Minimal Generation Gap Model for GAs Considering Both Exploration and Exploitation, Proc. 4th Int. Conf. on Soft Computing, pp.494-497 (1996).

[Watkins 92] Watkins, C. J. C. H., and Dayan, P.: Q-learning, Machine Learning 8, pp.279-292 (1992).

[Whitehead 93] Whitehead, S., Karlsson, J. and Tenenberg, J.: Learning Multiple Goal Behavior via Task Decomposition and Dynamic Policy Merging, in J. H. Connell *et al.* Robot Learning, Kluwer Academic Press (1993).

[Yoshida 99] Yoshida, S., Ono, I. and Ono, N.: Multi-agent Reinforcement Learning via Decomposed State Representation, Proc. 26th Symp. on Intelligent Systems, pp.163-168 (1999) (*in Japanese*).