Submitted to Workshop on Artificial Intelligence

Computational Evidence on Genetic Algorithms and Reinforcement Learning Algorithms for Module Assignment in Distributed Systems

Peng-Yeng Yin[1*], Yung-Pin Cheng[2], Chung-Chao Yeh[3] and Benjamin B. M. Shao[4]

[1]Department of Computer Science, Ming Chuan University, Taoyuan, Taiwan
[2]Department of Information and Computer Education, National Taiwan Normal University, Taipei, Taiwan
[3]Department of Computer Science, National Taiwan Ocean University, Keelung, Taiwan
[4]School of Accountancy and Information Management, Arizona State University, Tempe, AZ, USA

pyyin@mcu.edu.tw, ypc@ice.ntnu.edu.tw, ccyeh@cs.ntou.edu.tw, Benjamin.Shao@asu.edu

ABSTRACT

In a distributed system, it is important to find an assignment of program modules to processors such that system cost is minimized or system throughput is maximized. Researchers have proposed several versions of formulation to this problem. However, most of the versions proposed are NP-complete, and thus finding the exact solutions is computationally intractable. In this paper, we propose a genetic algorithm and a reinforcement learning algorithm to find the near-optimal module assignment. We present the computational evidence of the two algorithms with a set of simulated data. The direction of future research is suggested according to the experimental results.

Keywords: Module assignment problem, distributed systems, genetic algorithms, reinforcement learning.

[*]Corresponding author: Peng-Yeng Yin
5 Teh-Ming Rd., Taoyuan 333, Taiwan

1. Introduction

In a distributed system environment, it is important to find an assignment of program modules to a number of distributed processors such that a certain measure of system costs incurred by this assignment is minimized. There are several versions of the module assignment problems (MAP) proposed by researchers for dealing with various system costs and environmental constraints. But in general, the MAP is NP-complete [1]. Therefore, finding exact solutions to large-scale MAP problems is impractical.

The endeavors of interested researchers have been mainly devoted to two directions. One direction is to develop efficient algorithms that can derive exact solutions to simplified MAP versions under some particular constraints. For example, Nicol and O'Hallaron [2] proposed several efficient algorithms for tackling the particular MAP in linear array, shared-memory, and host-satellite systems. Fernandez-Baca and Medepalli [3] proposed a divide-and-conquer algorithm that can solve, in polynomial time, the MAP with a partial k-tree communication graph. The other direction is to design heuristic algorithms to yield approximate solutions to the MAP of more general cases. Lo [4] proposed a family of highly efficient heuristic algorithms that minimize the system costs of the module assignment as well as more complex existing algorithms. Hamam and Hindi [5] used a simulated annealing approach to find approximate solutions of several MAP models.

Our analysis is concentrated on the second direction. In this paper, we present the computational experience of employing genetic algorithms and reinforcement learning algorithms to find the optimal assignment of the program modules to heterogeneous processors such that the total execution and communication costs are minimized.

The remainder of this paper is organized as follows. Section 2 presents the problem definition of the MAP that will be addressed. A genetic algorithm and a reinforcement learning algorithm are proposed for the MAP in Sections 3 and 4, respectively. The simulation results are presented and discussed in Section 5. Finally, Section 6 concludes this work.

2. Problem Definition

In a distributed system, we have r program modules to be assigned to n heterogeneous processors. Assume module i requires memory resource and computation resource of m_i and p_i units, respectively, to accomplish its execution. However, each processor has its own resource capacities and such limit cannot be exceeded. Let M_k and P_k be the memory capacity and the computation capacity of processor k. First, we define an assignment $\boldsymbol{X}$ as a specification of $r \times n$ random variables x_{ik}, $1 \le i \le r$ and $1 \le k \le n$, where

$$x_{ik} = \begin{cases} 1 & \text{if module } i \text{ is assigned to processor } k; \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

and

$$\sum_{k=1}^{n} x_{ik} = 1, \tag{2}$$

since each module can be assigned to one processor only.

To satisfy the capacity constraints, we have

$$\sum_{i=1}^{r} x_{ik} m_i \le M_k \ , \tag{3}$$

and

$$\sum_{i=1}^{r} x_{ik} p_i \le P_k \ . \tag{4}$$

Further, we assume the communication link between two different processors is also capacitated, and the communication load on this link can not exceed this capacity. Let the communication capacity between processor k and processor l be D_{kl} and the communication load between module i and module j be d_{ij}. We thus have

$$\sum_{i=1}^{r}\sum_{j=1}^{r} x_{ik} x_{jl} d_{ij} \le D_{kl}, \quad \forall k \ne l \ . \tag{5}$$

In this paper, we consider two types of system costs, namely, the execution cost and the communication cost. Since the processors are heterogeneous, the execution cost incurred by assigning a module to different processors is not necessarily the same, so is the communication cost. We denote by e_{ik} the execution cost if we assign module i to processor k, and by c_{ijkl} the communication cost if we assign modules i and j to processors k and l, respectively. The optimal module assignment seeks to minimize the sum of the two costs. As such, the objective function of our MAP is

$$\text{Minimize} \qquad MAP(\boldsymbol{X}) = \sum_{i=1}^{r}\sum_{k=1}^{n} x_{ik} e_{ik} + \sum_{i=1}^{r}\sum_{j=1}^{r}\sum_{k=1}^{n}\sum_{l=1}^{n} x_{ik} x_{jl} c_{ijkl} \ , \tag{6}$$

subject to constraints (1)-(5).

The above version of the MAP is NP-complete [5]. Therefore, pursuing exact solutions to the MAP of large problem size is impractical. In the following sections, we employ a genetic algorithm and a reinforcement learning algorithm to approximate the optimal solution of the MAP.

3. Genetic Algorithms for Optimal Module Assignment

This section starts with an introduction of genetic algorithms, and the application to the MAP is then described.

3.1 Genetic algorithms

Genetic algorithms (GAs) are meta-heuristic algorithms based on natural genetic systems [6]. It has been theoretically proved that GAs provide robust search even when the search space is not continuous. GAs have found a variety of applications in combinatorial optimization problems. To solve an optimization problem, GAs maximize a fitness function corresponding to the merit of the underlying problem. The fitness function is parameterized by a binary string of 0' s and 1's, called a chromosome, and is used to evaluate the goodness of a possible assignment of parameter values. The effect of binary coding is to increase the number of dimensions of the search space such that the fitness function is stretched and the probability of being confined to a local optimum is decreased. A collection of such strings forms a population. GAs start with an initial population generated at random. The number of strings in the population is fixed during all generations. The population evolves to the next population using three genetic operators: selection, crossover, and mutation. The evolution process is iterated until either a near-optimal string is obtained or a pre-specified number of generations is reached.

Selection operator mimics the natural survival of the fittest. The probability with which a string is selected is proportional to the string fitness that is derived by calculating the fitness function of the corresponding string. The selected strings form a mating pool for the crossover operation. Crossover operator is a process by which each individual string can interchange information with its mate chosen at random from the mating pool. Since the highly fit strings occupy a large proportion of the population, they experience more trials of crossover operations and the search is navigated to "good" regions of the solution space. Crossover operator is performed with a crossover probability, p_c, which is usually among [0.6, 1.0] in various applications. Mutation operator is an occasional alteration of a string with a very small mutation probability, p_m. Mutation preserves sufficient diversity between strings of the population to prevent the unwilling premature convergence, and it also guarantees a non-zero probability of the search to any feasible string.

The user is required to specify the following parameters for applying GAs: the population size P, crossover probability p_c, mutation probability p_m, and maximal number of generations G. Next, we describe how GAs are applied to solve the MAP.

3.2 String representation and fitness function

Each assignment of an MAP can be encoded into a string as

$$\boldsymbol{A} = a_1 a_2 \cdots a_r, \tag{7}$$

where $a_i \in [1, n]$ represents the index of the processor to which module i is assigned. Note that character a_i can be encoded in binary with $\lceil \log_2 n \rceil$ bits. String $\boldsymbol{A}$ can be easily transformed to a corresponding assignment $\boldsymbol{X}$ (see Eq. (1)) of r modules to n processors; however, this assignment may violate constraints (2)-(5). The fitness of string $\boldsymbol{A}$, in a sense, is inversely proportional to the sum of the incurred cost and the exceeded requirement of the resources. Thus, we define the fitness function of string $\boldsymbol{A}$ as

$$f(\boldsymbol{A}) = K - \left(MAP(\boldsymbol{A}) + E(\boldsymbol{A})\right), \tag{8}$$

where K is a constant, $MAP(\boldsymbol{A})$ is the total execution and communication costs (see Eq. (6)) incurred by assignment $\boldsymbol{A}$, and $E(\boldsymbol{A})$ is a possible excess of resource requirement determined by

$$E(\boldsymbol{A}) = \sum_{k=1}^{n}\left(\max\left(\sum_{i=1}^{r} x_{ik} m_i - M_k, 0\right)\right) + \sum_{k=1}^{n}\left(\max\left(\sum_{i=1}^{r} x_{ik} p_i - P_k, 0\right)\right) + \sum_{k=1}^{n}\sum_{l=1}^{n}\left(\max\left(\sum_{i=1}^{r}\sum_{j=1}^{r} x_{ik} x_{jl} d_{ij} - D_{kl}, 0\right)\right). \tag{9}$$

3.3 Genetic operators

A population of P strings according to (7) is generated at random. This population then repeatedly evolves to subsequent populations using the three genetic operators described next.

Roulette-wheel selection. This operator reproduces P strings from the current population. Each string of the current population is selected with a probability proportional to its fitness, i.e., $Select_i = f(\boldsymbol{A_i}) \Big/ \sum_{j=1}^{\boldsymbol{P}} f(\boldsymbol{A_j})$, where $Select_i$ is the selection probability of string $\boldsymbol{A_i}$.

Single-point crossover. This operator randomly makes $P/2$ pairs from the strings of the population such that each string belongs exactly to one pair. Each pair would probably undergo crossover with a crossover probability. Let strings $\boldsymbol{A_i}$ and $\boldsymbol{A_j}$ be one pair to which the crossover operator is applied, for example,

$\boldsymbol{A_i} = 1\ 0\ 1\ 0\ 0 \quad 0\ 1\ 0\ 1\ 1$,

$\boldsymbol{A_j} = 0\ 0\ 1\ 1\ 1 \quad 1\ 1\ 0\ 0\ 1$.

The operator then randomly generates a crossover position λ between [1, $r\lceil \log_2 n \rceil$]. Notice that $r\lceil \log_2 n \rceil$ is equivalent to the string length. Two offspring $\boldsymbol{A}_i'$ and $\boldsymbol{A}_j'$ are produced by interchanging the substrings starting at position λ of $\boldsymbol{A_i}$ and $\boldsymbol{A_j}$. For example, say, λ = 5, we obtain

$\boldsymbol{A}_i' = 1\ 0\ 1\ 0\ 1 \quad 1\ 1\ 0\ 0\ 1$,

$\boldsymbol{A}_j' = 0\ 0\ 1\ 1\ 0 \quad 0\ 1\ 0\ 1\ 1$.

As such, the solution space is explored by interchanging information between strings.

Mutation. Each string may undergo mutation with a mutation probability. Note that we perform the mutation operator on character scale instead of binary bit. Let string $\boldsymbol{A_i} = a_1 a_2 \cdots a_r$ be chosen to be mutated. The operator randomly substitutes a new value for one character.

4. Reinforcement Learning for Optimal Module Assignment

This section introduces the reinforcement learning and illustrates how we solve the MAP by utilizing the most popular approach, *Q*-learning algorithm, to implement the reinforcement learning.

4.1 Reinforcement learning

The reinforcement learning addresses the issue of how a simple agent can learn a task through a sequence of trial-and-error interactions with its environment [7, 8]. The general concept of the reinforcement learning is depicted in Fig. 1. The agent examines the current state of its environment and makes a decision of choosing an action to perform. The state of the environment is, therefore, triggered by the agent's action and changed to another state. The agent observes the new state and receives a reward regarding the desirability about the state transition. The process is repeated and the agent learns an optimal policy that maximizes the expected sum of the cumulative rewards received over time.

4.2 Applying the reinforcement learning algorithm to the MAP

The optimal assignment of the MAP can be learned by a reinforcement learning algorithm. Fig. 2 depicts the relationship between them. The directed graph consists of $r+2$ layers of nodes. The first layer and the last layer contain only one node, namely, the starting node and the sinking node. The remaining r layers represent the possible assignments of the r modules. Each of these layers contains n nodes and each of the n nodes corresponds to the assignment of this module to a specific processor. A path emanating from the starting node and terminating at the sinking node represents one possible assignment of the r modules. Next, we define the five key components of the reinforcement learning for the MAP as follows.

(1) The set of environment states, $S = \{s_0, s_f\} \cup \{s_{i,j}\}_{1 \le i \le r, 1 \le j \le n}$. Elements s_0 and s_f are the initial state and the final state corresponding to the starting node and the sinking

node, respectively, and $s_{i,j}$ indicates the state that module *i* is assigned to processor *j*.

(2) The set of agent actions, $A = \{a_i\}_{1 \le i \le n}$. Selecting action a_i to perform means assigning the next module to processor *i*. The action selection rule will be further discussed.

(3) The set of scalar rewards, R. The reward value is computed by the reward function discussed below.

(4) The state transition function, $\delta(s_{i,j}, a_k) = s_{i+1,k}$. This is apparent by the definitions of *S* and *A*.

(5) The reward function,

$$\rho(s_{i,j}, a_k) = \frac{\sum_{t=1}^{n}(M_t - \sum_{w=1}^{i+1} x_{wt} m_w) + \sum_{t=1}^{n}(P_t - \sum_{w=1}^{i+1} x_{wt} p_w) + \sum_{t=1}^{n}\sum_{v=1}^{n}(D_{tv} - \sum_{w=1}^{i+1}\sum_{u=1}^{i+1} x_{wt} x_{uv} d_{wu})}{e_{i+1,k} + \sum_{w=1}^{i}\sum_{t=1}^{n} x_{wt} c_{w,i+1,t,k}}. \quad (10)$$

Since we have already known the assignment of the first *i*+1 modules when we receive the reward $\rho(s_{i,j}, a_k)$, the remaining resource capacity and the extra system costs incurred by the assignment of the (*i*+1)*th* module can be computed. We design the reward function to favor the assignment of the next module that maximizes the remaining resource capacity (for later use) and results in least system cost (for optimization objective).

Q-learning algorithm is commonly used to learn the optimal policy in the reinforcement learning [8] and, hence, we employ the *Q*-learning algorithm to learn the optimal assignment of the MAP. First, we define the Q function, $Q(s_{i,j}, a_k)$, as the maximum cumulative reward which can be attained by performing action a_k in state $s_{i,j}$ and then proceeding optimally until the final state s_f is observed. The recursive definition of $Q(s_{i,j}, a_k)$ is given by

$$Q(s_{i,j}, a_k) = \rho(s_{i,j}, a_k) + \gamma \max_{a_l} Q(s_{i+1,k}, a_l), \tag{11}$$

where $\gamma \in (0,1)$ is the discounting factor that determines the relative value of the rewards received in the future. The agent then initializes a table of the estimate of the Q function for each possible state-action pair. These table entries are iteratively updated using Eq. (11). The Q-learning algorithm for the MAP is summarized in Table 1. The algorithm is repeated for a pre-specified maximum number of iterations. Then the r sequential actions chosen by the learned optimal policy constitute the optimal assignment. There still remains an issue of how we choose the action in a given state (see Step 4 in Table 1). Obviously, if every action can be visited infinitely often, the optimal assignment can be reached. However, in the real-world applications where the computational time is limited, how do we choose the minimum number of actions which can sufficiently explore the policy space?

Assume that the agent is in state $s_{i,j}$ and faces the choice among a set of available actions $\{a_k\}_{1 \le k \le n}$. We propose a thresholded maximum-exploitation action selection rule to determine the probability of choosing action a_k as follows.

$$p(a_k | s_{i,j}) = \begin{cases} 1 & \text{, if } q < q_0 \text{ and } k = \arg\max_l \hat{Q}(s_{i,j}, a_l); \\ 0 & \text{, if } q < q_0 \text{ and } k \neq \arg\max_l \hat{Q}(s_{i,j}, a_l); \\ \frac{1}{n} & \text{, otherwise,} \end{cases} \tag{12}$$

where $q \in [0, 1]$ is a randomly drawn number, $q_0 \in (0, 1)$ is the threshold controlling the relative emphasis on each sub-rule. This rule facilitates the controlled tradeoff between the selection of the action that delivers the maximum estimate of $\hat{Q}$ and the uniform random selection. It has been empirically proved that the thresholded maximum-exploitation action selection rule outperforms several other competing rules [9].

5. Simulation Results

In this section we present the computational evaluations on the proposed algorithms for a set of simulated MAPs. The proposed algorithms were encoded in C++ language and were executed on a 233MHz PC with 32MB RAM. The testing problem set was generated at random for various values of number of modules (*r*) and number of processors (*n*). According to the values of *r* and *n*, the testing problems can be classified into small-scale (Problems 1-6), medium-scale (Problems 7-11), and large-scale problems (Problems 12-20), as shown in the first three columns of Table 2. For each problem, we draw parameter values at random from the following ranges, $m_i \in [1, 20]$, $M_k \in [1, 100]$,

$p_i \in [1, 20]$, $P_k \in [1, 100]$, $d_{ij} \in [1, 4]$, $D_{kl} \in [1, 100]$, $e_{ik} \in [1, 100]$, and $c_{ijkl} \in [1, 100]$. To simulate the communication graph between modules, we randomly generate a spanning tree with $r-1$ edges covering r nodes. Each node represents a module and each edge specifies the communication requirement between its two connecting modules. The communication direction is decided randomly.

To evaluate the comparative performances between the genetic algorithm and the *Q*-learning algorithm, we specify the same CPU time limit for the executions of both algorithms. The CPU time limits are 1 second for small-scale problems, 10 seconds for medium-scale problems, and 500 seconds for large-scale problems (see the last column in Table 2). Since both the genetic algorithm and the *Q*-learning algorithm are probabilistic and each independent run of the same algorithm on a same testing problem may yield a different result, we calculate the average costs incurred over 10 independent runs for each problem (see the fourth and the fifth columns in Table 2). It is observed that the *Q*-learning algorithm consistently outperforms the genetic algorithm on all but the first two problems. Since the first two problems are relatively small, both algorithms can obtain the same optimal solution. To provide a clearer view on the comparative performances, the cost offset that is computed as the ratio between the cost difference and the cost derived by *Q*-learning algorithm is tabulated in the sixth column of Table 2. It is seen that the cost offset ranges from 0% to 14.20%, and the average cost offset is 5.26% over all problems. Therefore, the experimental results manifest that, using our current implementation schemes, the *Q*-learning algorithm is more effective than the genetic algorithm for the MAP. We argue the following are the possible reasons. Within each iteration, the *Q*-learning algorithm constructs a solution module-by-module by examining the immediate rewards that express the desirability about the assignment of the module currently under consideration. It is therefore easier to meet the capacity constraints when assigning the next module (as we model in the reward function (10)). On the other hand, the genetic algorithm produces offspring by fostering a pool of population. Each member of the population is a solution for the assignment of all modules, and the fitness function (see Eq. (8)) of the member measures the desirability about the assignment of all modules instead of the assignment of a single module. Thus, it is hard to predict the feasibility of the yielded offspring. This theoretical difference is often referred to as the reason why genetic algorithms appear to be less effective than other meta-heuristics in various applications [6]. Researchers have suggested to develop customized genetic operators for specific applications [10], and their experiments showed the performance can be significantly improved by such tailored operators. Consequently, our future research will be concentrated on the development of customized genetic operators for the MAP.

6. Conclusions

In this paper, we have proposed a genetic algorithm and a reinforcement learning algorithm for the module assignment problem. The underlying problem has been formally defined and the rationale of converting the problem to appropriate forms was derived. A set of simulated problems of different scales is experimented with. The computational experience manifests that, under our current implementation scheme, the *Q*-learning algorithm outperforms the genetic algorithm in solving the module assignment problem. The performance of the genetic algorithm may be improved by developing more sophisticated operators, which suggest the avenues for future research.

REFERENCES

[1] V. M. Lo, “Task assignment in distributed systems”, Ph.D. dissertation, Dep. Comput. Sci., Univ. Illinois, Oct. 1983.

[2] D. M. Nicol and D. R. O’Hallaron, “Improved algorithm for mapping pipelined and parallel computations”, IEEE Trans. on Computers, Vol. 40, 1991, pp. 295-306.

[3] D. Fernandez-Baca and A. Medepalli, “Parametric module allocation on partial k-trees”, IEEE Trans. on Computers, Vol. 42, 1993, pp. 738-742.

[4] V. M. Lo, “Heuristic algorithms for task assignment in distributed systems”, IEEE Trans. on Computers, Vol. 37, 1988, pp. 1384-1397.

[5] Y. Hamam and K. S. Hindi, “Assignment of program modules to processors: A simulated annealing approach”, European Journal of Operational Research, Vol. 122, 2000, pp. 509-513.

[6] D. E. Goldberg, Genetic Algorithms: Search, Optimization and Machine Learning, Addison-Wesley, Reading, MA, 1989.

[7] L. P. Kaelbling and A. W. Moore, “Reinforcement learning: a survey”, Journal of Artificial Intelligence Research, Vol. 4, 1996, pp. 237-285.

[8] T. M. Mitchell, Machine Learning, McGraw-Hill, 1997.

[9] P. Y. Yin, “Maximum entropy-based optimal threshold selection using deterministic reinforcement learning with controlled randomization”, to appear in Signal Processing, 2002.

[10] S. Y. Ho and Y. C. Chen, “An efficient evolutionary algorithm for accurate polygonal approximation”, Pattern Recognition, Vol. 34, 2001, pp. 2305-2317.

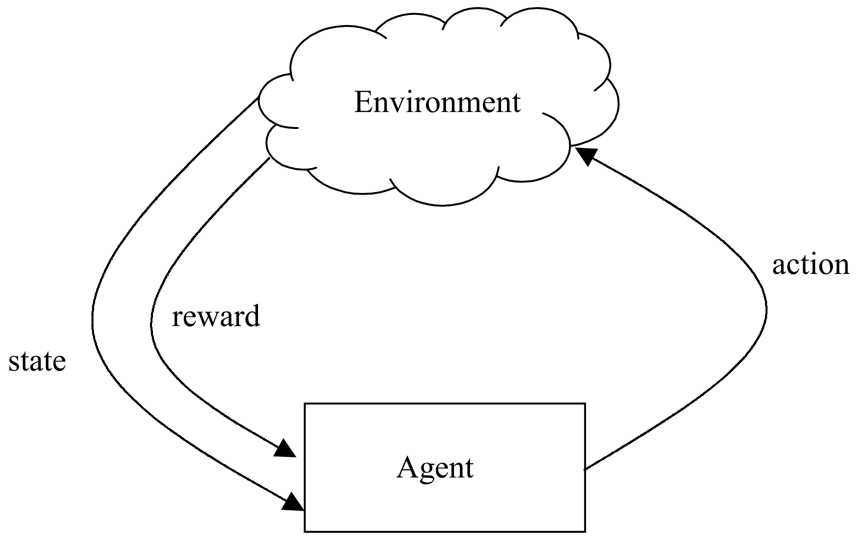

Fig. 1 General concept of the reinforcement learning.

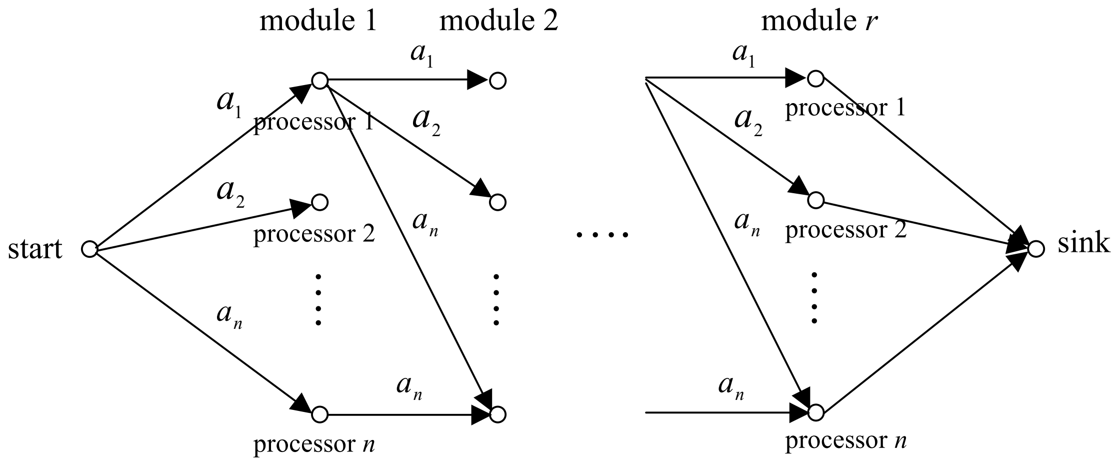

Fig. 2 The relationship between the MAP and the reinforcement learning.

Table 1 The *Q*-learning algorithm for the MAP.

Step 1	Initialize the table entry $\hat{Q}(s_{i,j}, a_k) = 1$ for each *i*, *j*, *k*.
Step 2	Set *Iteration* = 1.
Step 3	Start with the initial state s_0.
Step 4	Select an action according to the action selection rule.
Step 5	Update the table entry $\hat{Q}(s_{i,j}, a_k)$ using Eq. (11).
Step 6	If the final state s_f is not yet reached, goto Step 4.
Step 7	If *Iteration* < *MAX_ITERATION* // the stopping criterion is not yet satisfied// Set *Iteration* = *Iteration* + 1, and goto Step 3.
Step 8	Start from the initial state s_0, output the sequence of actions which result in the maximum $\hat{Q}$ values until the final state s_f is reached.

Table 2 The average total costs obtained by applying the genetic algorithm and *Q*-learning algorithm with their offsets given the same CPU time limits.

Problem	r	n	Total cost (1) Genetic algorithm	Total cost (2) *Q*-learning algorithm	$\frac{(1)-(2)}{(2)}\times 100\%$	CPU time
1	10	5	492	492	0.00%	1 second
2	10	7	478	478	0.00%	
3	30	15	869	845	2.84%	
4	30	20	853	827	3.14%	
5	50	25	1739	1594	9.10%	
6	50	35	1520	1331	14.20%	
7	80	40	2276	2215	2.75%	10 seconds
8	80	50	2007	1921	4.48%	
9	100	50	2634	2575	2.29%	
10	100	60	2556	2315	10.41%	
11	100	70	2318	2160	7.31%	
12	130	60	3640	3383	7.6%	500 seconds
13	130	80	2613	2544	2.71%	
14	130	100	2666	2418	10.26%	
15	150	75	3359	3199	5.00%	
16	150	95	3138	2933	6.99%	
17	150	115	2887	2700	6.93%	
18	180	90	3852	3664	5.13%	
19	180	110	3791	3687	2.82%	
20	180	130	3343	3304	1.18%	
Average					5.26%	