A Fast Learning Agent Based on the Dyna Architecture

YUAN-PAO HSU[1] WEI-CHENG JIANG[2]
[1]*Department of Computer Science and Information Engineering*
National Formosa University
Yunlin, 632 Taiwan
[2]*Department of Electric Engineering*
National Chung Cheng University
Minhsiung,621 Taiwan

In this paper, we present a rapid learning algorithm called Dyna-QPC. The proposed algorithm requires considerably less training time than Q-learning and Table-based Dyna-Q algorithm, making it applicable to real-world control tasks. The Dyna-QPC algorithm is a combination of existing learning techniques: CMAC, Q-learning, and prioritized sweeping. In a practical experiment, the Dyna-QPC algorithm is implemented with the goal of minimizing the learning time required for a robot to navigate a discrete state-space containing obstacles. The robot learning agent uses Q-learning for policy learning and a CMAC-Model as an approximator of the system environment. The prioritized sweeping technique is used to manage a queue of previously influential state-action pairs used in a planning function. The planning function is implemented as a background task updating the learning policy based on previous experience stored by the approximation model. As background tasks run during CPU idle time, there is no additional loading on the system processor. The Dyna-QPC agent switches seamlessly between real and virtual modes with the objective of achieving rapid policy learning. A simulated and an experimental scenario have been designed and implemented. The simulated scenario is used to test the speed and efficiency of the three learning algorithms, while the experimental scenario evaluates the new Dyna-QPC agent. Results from both simulated and experimental scenarios demonstrate the superior performance of the proposed learning agent.

Keywords: Reinforcement learning, Q-learning, CMAC, Prioritized Sweeping, Dyna Agent

1. INTRODUCTION

Reinforcement Learning, or RL for short, is a method where the relationship between "states" and "actions" is mapped such that the largest accumulated reward is attainable from successive interactions between an agent and its environment. The way that the mapping between "states" and "actions" is learned is also called the learning of a "policy," which is a major characteristic of RL [1]. As opposed to the supervised learning which learns from externally provided examples, the RL method creates optimal or near optimal policy solely from interplays with its environment. Thus, RL has become a powerful tool for solving complex sequential decision-making problems in various areas [6-14].

RL policy can be "trained" by using a series of random "trial-and-error" searches through the problem space, however, a considerable length of time may be required before the policy converges upon an optimal solution. This weakness has prevented RL

from being widely applied in real-time, real-world applications. Several methods have been proposed to speed up the learning process such as the Dyna algorithm, prioritized sweeping, and so forth [2] [3].

Solving the problem of a complex system which has a large number of variables is difficult because of the heavy computational burden involved in calculating the values of all the various transition probabilities. This phenomenon is known as the "curse of modeling." For dealing with this, there have already many methods been presented such as neural networks [20], system identification methods [21], probabilistic methods [22], decision tree [24], CMACs [5], and so on.

In addition, there is an exponential increase in complexity associated with more state variables added for a rather big task with bigger problem space, so that manipulating or even storing the related elements of the value function becomes unmanageable. This is called the "curse of dimensionality [19]."

Dyna architecture is one of the most representative algorithms among model-based reinforcement learning algorithms. But the orginal concept of the Dyna didn't pay much attention on how the structure is implemented [2]. In order to realize the Dyna concept, the combined architectures of the Dyna and Q-learning were proposed where the Q-learning is in charge of the policy learning and a lookup table of the historical record of state-action transitions is built to present the system with a brief model [25-27]. We call this kind of architecures as the Table-based Dyna architecture.

In this research, we attempt to address the "curse of modeling" problem by improving the Table-based Dyna architecture. We will propose a Dyna like agent called Dyna-QPC which embraces the following techniques: (1) CMAC [4], (2) Q-learning [5], and (3) prioritized sweeping [3]. The architecture of the proposed agent employs a CMAC model to mimic the learning process and thus solve the "curse of modeling" problem to some degree. The Q-learning algorithm, a well known model-free algorithm, is used for policy learning, while the prioritized sweeping method is used to assess whether a particular state-action pair should be stored in a prioritized queue containing influential state-action pairs. Each state-action pair is associated with a Q-value which gives the desirability of choosing an action in that state. During execution, the agent retrieves these influential state-action pairs and updates their corresponding Q values to speed up the policy learning.

The Q-learning algorithm searches a real environment and gains real experiences which can be assembled into a Markov chain. The CMAC concurrently collects these real experiences and builds a model which approximates the Markov chain. Both forward state transitions (current state to future state) and backward state transitions (current state to previous state) are approximated by the CMAC. The Markov chain is vital for performing prioritized sweeping and updating Q-values.

The major advantage of the proposed Dyna-QPC agent is that it is able to explore the environment and learn through trial-and-error, while spending considerably small amount of training time. This is achieved by through an effective combination of CMAC and prioritized sweeping for the learning process in the agent. In this way, the Dyna-QPC agent retains all the benefits of a model-free method and yet also has advantages associated with a model-based method.

In this paper, we discuss the components of the architecture of the learning agent in Section II. The proposed architecture is described in Section III. In Section IV the

performance of different learning algorithms under simulated conditions is demonstrated before presenting the results of an experiment using our Dyna-QPC learning agent. Finally, conclusions terminate the paper.

2. BACKGROUND

2.1 MDPs

A reinforcement learning task satisfying the Markov property is called a Markov decision process or, MDP in short. A particular finite MDP is defined by its state and action sets and by the one-step dynamics of the environment. Given any state and action, s and a, the transition probability of each possible next state, s' is $p(s'|\, s, a)$. Also, the expected reward of executing action, a, in current state, s, transiting to next state, s', is $r(s, a, s')$. The objective is to find a policy which at time step t selects an action a_t given the state s_t and maximizes the expected discounted future cumulative reward, or return: $r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...$, where $\gamma \in [0, 1]$ is a factor called discount rate. MDPs have been used extensively in stochastic control theory of discrete-event systems [15] [16].

2.2 Q-Learning

A breakthrough in the development of RL was the formalization of the Q-learning algorithm, which has been characterized as a model-free algorithm. Thus, Q value is learned in order to estimate its best value Q* based on the values of optimized state-action pairs. The corresponding Q value is updated by:

$$Q(s,a) = Q(s,a) + \beta(r + \gamma \max_{a'} Q(s',a') - Q(s,a)) \tag{1}$$

where β is the learning rate. The pseudo code of the Q-learning algorithm is shown in Fig. 1. Eventually this algorithm has been shown to converge to Q* with probability 1. However, the drawback is that it might take a long period of time to interact with the environment to solve any given problem.

Initialize $Q(s, a)$, arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose action a; from s using policy derived from Q (e.g. ε-greedy)
    Observe resultant state, s', and reward, r
    $Q(s,\ a) = Q(s,\ a) + \beta(r + \gamma \max_{a'} Q(s',\ a') - Q(s,\ a))$
    $s \leftarrow s'$
Until s is terminal

Fig. 1 Q-learning algorithm

2.3 CMACs

Cerebella Model Articulation Controllers (CMACs) can learn to estimate the output of a function of interest by referring to a look-up table which stores a set of synaptic weights. The corresponding output is derived by summing up some of these synaptic

weights. By a fine-tuning process of the weights, the relationship between inputs and outputs of the function can be approximated by the CMAC. A typical CMAC is characterized by a series of mappings as shown in Fig. 2. The data flow is a sequence of mapping S → C→ P → O, where S, C, P, and O stand for the input state, conceptual memory, actual memory, and output, respectively. The overall mapping of S → O can be represented by O = h(S) [17].

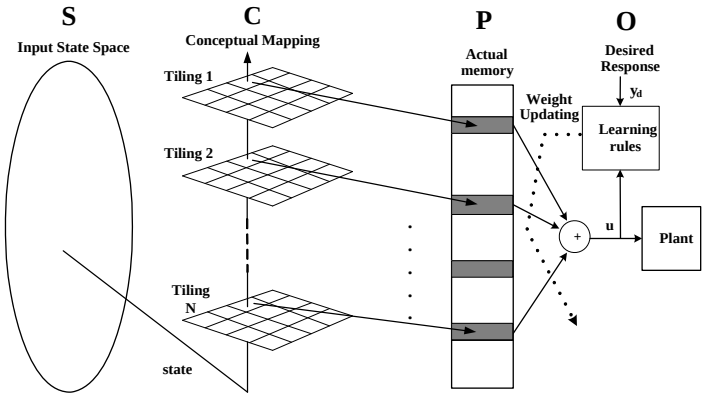


Fig. 2 CMAC architecture

As shown in (2), the output u_j corresponding to the j-th input of x is computed by summing the weights stored in the activated physical memory cells

$$u_j = W_j x_j = \sum_{i=1}^{R} W_{ji} x_j . \tag{2}$$

The weight of the indexed cell W_{ji} is updated according to the minimization of the error between the desired output value y_{di} and the summation of activated memory cell values $W_j x_j$, or u_j, based on the scale of:

$$\Delta W_{ji} = \eta \frac{y_{dj} - W_j x_j}{R} \tag{3}$$

where $W_j x_j$ is the actual output relative to the j-th input of x; y_{dj} is the desired output; η is the learning rate which is in the range of (0, 1]; R is the number of activated memory cells; i = 1, 2, …, R; W_j = [W_{j1}, W_{j2}, …, W_{jR}].

As can be seen, the CMAC has properties of local generalization, rapid computation, function approximation, and output superposition. This structure facilitates model approximation by collecting real experience through interaction with the world to build a virtual world model.

2.4 Prioritized Sweeping

As mentioned in Section 1, the Dyna algorithm uses state-action pairs to update its Q values during the time interval between interactions with the real world. However, all state-action pairs are not equal in their importance. The Q value indicates the degree of the effect of a state-action pair. Also, the preceding state-action pairs to those that have had a large effect in the past are also more likely to have a large effect in the future. So, it

is natural to prioritize the updates according to a measure of their urgency, and this is the idea behind the prioritized sweeping algorithm, as shown in Fig. 3 is the pseudo code.

Initialize $Q(s, a)$, $Model(s, a)$, for all s, a, and $PQueue$ to empty
Do forever:
(a) $s \leftarrow$ current (nonterminal) state
(b) $a \leftarrow policy(s, Q)$
(c) Execute action a; observe resultant state, s', and reward, r
(d) $Model(s, a) \leftarrow s', r$
(e) $p \leftarrow |r + \gamma \max_{a'} Q(s', a') - Q(s, a)|$.
(f) if $p > \theta$, then insert s, a into $PQueue$ with priority p
(g) Repeat N time, while $PQueue$ is not empty:
$s, a \leftarrow$ first($PQueue$)
$s', r \leftarrow Model(s, a)$
$Q(s, a) \leftarrow Q(s, a) + \beta (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
Repeat, for all $\underline{s}, \underline{a}$ predicted to lead to s:
$\underline{r} \leftarrow$ predicted reward
$p \leftarrow |\underline{r} + \gamma \max_{a} Q(s, a) - Q(\underline{s}, \underline{a})|$
If $p > \theta$ then insert $\underline{s}, \underline{a}$ into $PQueue$ with priority p

Fig. 3 Prioritized sweeping algorithm

The key problem in prioritized sweeping is that the algorithm relies on the assumption of discrete states. When a change occurs in one state, the method performs a computation on all the preceding states that may have been affected. However, it is not explicitly pointed out how these states could be identified or processed efficiently [2] [23]. In this paper, we use the CMAC to approximate the model and contend with this difficulty. The effect of this approximation process is that when an action executed on a state causes a great many variation of its Q value, all the affected states (backward and forward) are retrieved from the CMAC.

2.5 Dyna Architecture

Fig. 4 shows the general architecture of a Dyna agent in [1] [2]. The central bold arrows represent the interaction between the agent and the environment. This creates a set of real experiences. The left dash arrow in the figure represents the agent's learning policy. The value function is updated from direct Reinforcement Learning (RL) derived from the agent's interaction with the environment. The process of retrieving simulated experience from the model is shown by the search control arrow. The model is built based on real experiences that are recorded on a memory table that can also be retrieved to represent simulated experiences. The planning update function updates the policy from simulated experiences. Fig. 5 shows the pseudo code used in the implementation of the Dyna-Q algorithm which we call it as "Table-based Dyna-Q architecture" in this paper.

The Table-based Dyna-Q learning and planning are accomplished by the same algorithm; real experience is used for learning and simulated experience for planning. But, in Fig. 5, we have identified two weaknesses in the method of planning update. Firstly, in step (f), that the planning update chooses a state-action pair randomly from a table of observed state-action set is inefficient. Since, the experienced state-action pairs increase

following the learning progress making it difficult to target the most influential state-action pairs to be retrieved for update. Secondly, a virtual model is demanded in order to provide the predecessor and successor of any state-action pair used in the planning update.

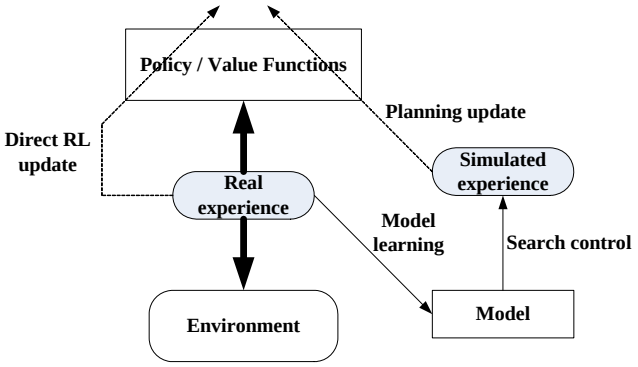


Fig. 4 General Dyna architecture

Initialize $Q(s, a)$, $Model(s, a)$, for all s, a
Do forever:
(a) $s \leftarrow$ current (nonterminal) state (direct RL update)
(b) $a \leftarrow \varepsilon$-greedy(s, Q)
(c) Execute action a; observe resultant state, s', and reward, r
(d) $Q(s, a) \leftarrow Q(s, a) + \beta (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
(e) $Model(s, a) \leftarrow s', r$ (assuming deterministic eniv.)
(f) Repeat N time, (planning update)
$s \leftarrow$ random previously observed state
$a \leftarrow$ random action previously taken in s
$s', r \leftarrow Model(s, a)$
$Q(s, a) \leftarrow Q(s, a) + \beta (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$

Fig. 5 Table-based Dyna-Q algorithm

These weaknesses are addressed in this paper. Firstly, in step (f) instead of choosing state-action pairs randomly, we should directly access the most influential state-action pairs and use them for the planning update. To this regard, the idea of prioritized sweeping is adopted to locate the most influential state-action pairs and organize them into a priority queue. The highest priority state-action pair has the greatest chance of being chosen for the update. By this, the planning update can be efficiently speeded up.

Secondly, a virtual model providing the system with forward and backward traceable state-action pairs should be built up. The CMAC structure is one of the best candidates to be utilized to model the environment. In turn, a CMAC-Model for forward tracing the successors and a CMAC-R for backward tracing the predecessors, of previously influential state-action pairs are developed.

3. PROPOSED ARCHITECTURE

3.1 The CMAC-Model

As mentioned in section 2.5, a forward and backward traceable model is required to amend the weaknesses of the Table-based Dyna-Q. The CMAC-Model shown in Fig. 6 is used to model the system environment for the forward tracing. The model maps the current state-action pair (s, a) obtained from input values (s', r), where s' is the successor state and r the immediate reward in memory. These memory values are modified based on the error between actual output and desired output.

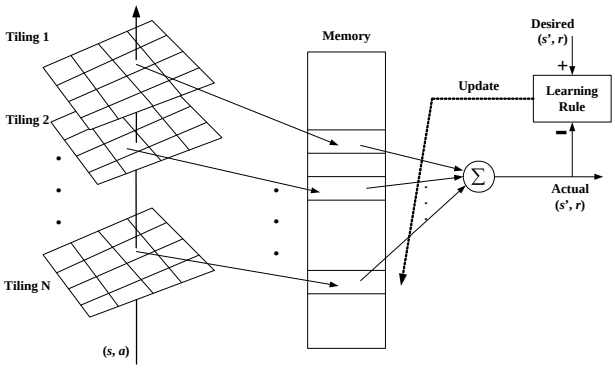


Fig. 6 CMAC-Model architecture.

The CMAC-Model is a virtual representation of the real world and provides the system with a simulated environment. The input of the CMAC-Model is dynamically generated by successively retrieving a series of state-action pairs (s, a) from a queue, which in turn is constructed by the prioritized sweeping algorithm. The CMAC-Model outputs a series of corresponding (s', r) pairs for the planning update function. Fig. 7 lists the algorithm in which γ represents learning rate of the CMAC-Model and c represents the number of memory cells per (s, a) pair.

Clear memory, $w_{s,a}(F) = 0$, where $\forall s \in$ states, $\forall a \in$ actions, $\forall F \in$ CMAC-tiles
Do forever:
(a) If model learning: (s, a) $\leftarrow$ environment
else: (s, a) $\leftarrow$ first(*PQueue*)
(b) F = *features*(s, a) (hit memory in CMAC)
(c) (s', r) = sum(F) (output)
(d) If model learning: (update contents of hit memory)
$w_{s,a}(F) = w_{s,a}(F) + \gamma$/c[desired($s$', a) – actual(s', a)]
(e) go to (a)

Fig. 7 CMAC-Model algorithm

3.2 CMAC-R Model

The CMAC-R is to model the environment for the purpose of providing a backward

tracing model. It should be aware that when the agent interacting with the environment it may produce an endless cyclic sequence path as Fig. 8 shows. For simplicity the subscript in the figure denotes the time step only. For instance, a robot at state s_n executes action a_n, say turn right, transiting to state s_{n+1}, then executes action a_{n+1}, say turn left, consequently the robot goes back to the original state s_n. If the inverse model approximator uses only state s_i as its input, the backward tracing sequence would be like $s_{n+2} \rightarrow (s_n, a_{n+2})$; $s_n \rightarrow (s_{n+1}, a_{n+1})$; $s_{n+1} \rightarrow (s_n, a_n)$; $s_n \rightarrow (s_{n+1}, a_{n+1})$, meaning that the backward tracing traps the robot in a cyclic path $s_n \rightarrow (s_{n+1}, a_{n+1})$, $s_{n+1} \rightarrow (s_n, a_n)$ as illustrated in Fig. 8. The state-action pair, (s_{n-1}, a_{n-1}), is never backward traceable from state s_n. Hence, in our design the CMAC-R Model concatenates state s, and action a, and uses this as its input in order to cope with an environment which may contain a cyclic path. This confirms the return of the correct preceding value.

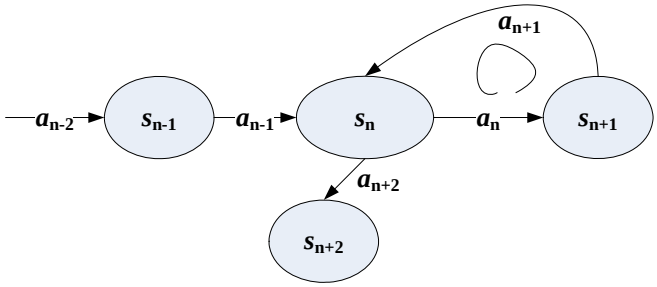


Fig. 8 Cyclic path

The architecture and algorithm of the CMAC-R Model is the same as the CMAC-Model (Fig. 6) except that the output (s', r) is replaced by ($\underline{s}$, $\underline{a}$) which stands for the predecessor of the input state-action pair, (s, a).

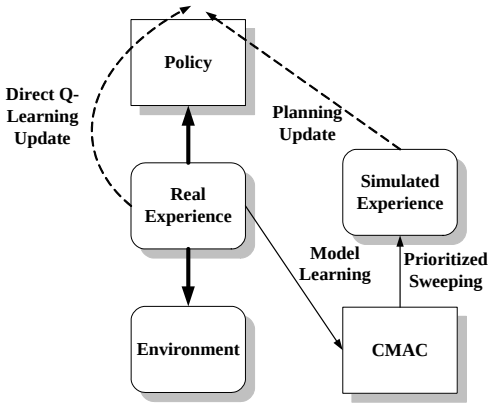


Fig. 9 Proposed Dyna agent architecture

3.3 Dyna-QPC Architecture

The proposed Dyna-QPC agent architecture is illustrated in Fig. 9. In the design, a CMAC model is employed to implement the learning function and the prioritized sweeping algorithm for search control. To train the Q-learning algorithm, data acquired from the agent's interaction with the real environment is used. Simultaneously, a CMAC model is approximating the environment using the same data. The CMAC environment model provides the Q-learning algorithm with virtual interaction experience in order to update the Q-learning policy and value function. This is achieved during the time interval when there is no interplay between the agent and the real environment. The agent switches seamlessly between real and virtual environment models, a strategy which greatly reduces the overall learning time.

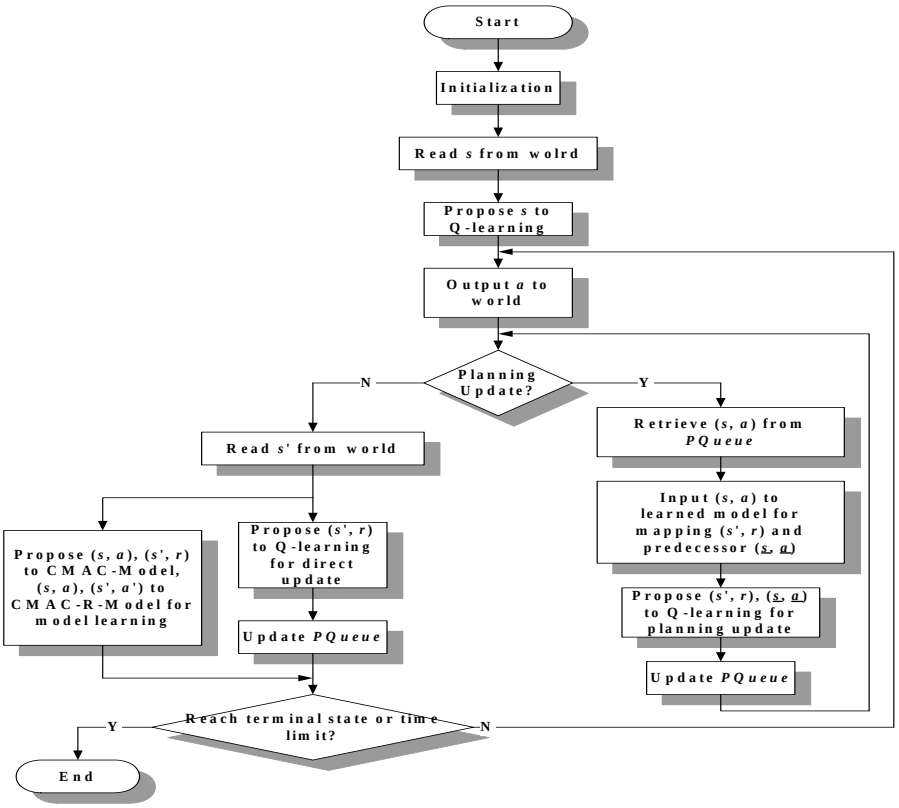


Fig. 10 Dyna-QPC algorithm

The Dyna-QPC algorithm is depicted as the flowchart in Fig. 10. The main difference between our design and the Table-based Dyna-Q is that the planning update function is not achieved by randomly retrieving previously observed state-action pairs. Instead a queue is used to record influential state-action pairs. If the variation of a state-action pair's Q-value (p in Fig. 3), exceeds a threshold value θ, the state-action pair is inserted into the queue with a priority determined by its p value. Therefore, the agent can always access a state–action pair with the potential of providing the greatest im-

provement to its policy during the planning update process.

Furthermore, the prioritized sweeping method does not specifically denote how to trace predecessor states from any given state. Actually, this is a model learning problem and has been investigated extensively by using a variety of methods. We use the CMAC structure to solve this model learning problem as mentioned in previous sections.

4. SIMULATIONS AND EXPERIMENTS

4.1 Simulated Environment

In the simulations, a virtual structural space is constructed in which a differential-wheeled mobile robot can move around. The robot learns an optimal or near optimal policy that can lead the robot to the goal in as few steps as possible without bumping into any obstacles in the space. The Q-learning, Table-based Dyna-Q, and Dyna-QPC algorithms are applied to this task.

◆ **Structural Space**

The structural space was an 8 meter by 8 meter flat space surrounded by walls. The goal was located at the center of the space and surrounded by obstacles as shown in Fig. 11. The robot had to learn to avoid contact with the walls and obstacles in accordance with its sensory data and successfully reach the goal.

◆ **The Differential-drive Robot**

A simulated robot with a length of 60 cm and a width of 50 cm was equipped with 16 sonar sensors and a portable GPS navigation system, as illustrated in Fig. 12. The sonar sensors had a detection range from 1 cm to 200 cm. Robot walking distance for each action command was about 25 cm. The GPS system was used to locate the planar coordinates, (x, y), and the angle, θ, of the robot in the space.

The concatenated data, $s = \{x, y, \theta, Sonar\}$, from the sonar sensors and the GPS system represents the aforementioned state, s. The robot had four actions: forward, backward, turn right, and turn left. The four positioning data elements x, y, θ, and *Sonar* were encoded by 5, 5, 4, and 4 binary bits respectively. Hence, the input state, s, was represented by an 18 bit binary number, meaning that the possible states would be $2^{18} = 262{,}144$. This is quite a large number of possible states from the viewpoint of learning from scratch.

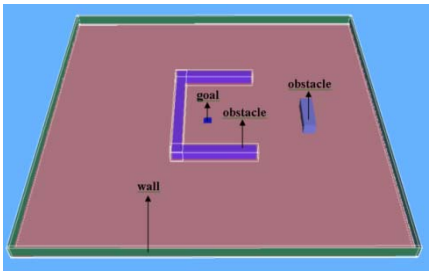


Fig. 11 8m x 8m simulation space.

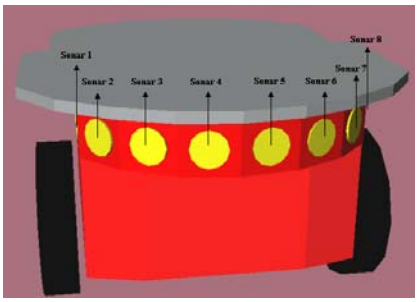


Fig. 12 Differential-drive mobile robot

4.2 Simulations

The learning agent was implemented using each of the three learning algorithms. Each implementation was simulated in 40 training sets. A training set included 200 episodes. An episode was terminated under one of two conditions; either the robot reached the goal or the robot exceeded 5000 steps. Once the episode was terminated, the robot commenced another episode from a randomly assigned starting position. Each average number of steps taken by the robot in each episode of a training set was recorded for comparison.

◆ **Q-Learning Algorithm**

For the Q-Learning algorithm, in order to reduce the possibility of the policy converging on any local optimum during simulation, the exploration rate ε was set to 1% (ε-greedy). The reward signal was set as follows: r = -100 when bumping into the walls or obstacles; r = 100 when reaching the goal; r = -1 otherwise. The reason for setting r to -1, when the robot is neither in contact with an obstacle nor the goal, is to increase the possibility of choosing a previously untried action when the robot finds itself at the same state at a later training stage. β and γ were both set to 0.9 in the simulation.

◆ **Table-based Dyna-Q Agent**

The Table-based Dyna-Q agent performs learning by storing every state-action pair that the system has experienced in a memory table and then randomly retrieving state-action pairs from the table during execution of the planning update function.

◆ **The Dyna-QPC Agent**

The Dyna-QPC agent used the same Q-learning algorithm as the Table-based Dyna-Q agent, but the method employed for training the model and the planning update function was modified. The CMAC-Model and the CMAC-R-Model were included as approximators for model learning. The CMAC structure using 16 tiles was used for mapping system inputs and outputs. The CMAC-Model converted the input (s, a) into output (s', r), whereas the CMAC-R- Model converted the same input to output ($\underline{s}$, $\underline{a}$). The 16 tilings correspond to parameter c in step (d) of Fig. 7, implying that for each input (state-action pair) the CMAC used 16 memory cells to store the related one-sixteenth output. A β value of 0.999 was used in the simulation.

4.3 Discussion of Simulation Results

Fig. 13 depicts the average number of steps for the robot taking to arrive at the goal using each of the three learning algorithms. Each learning episode was composed of 40 individual training sets. The three learning curves represent simulated results obtained from using the Q-learning, Table-based Dyna-Q and Dyna-QPC algorithms. The planning number is the number of retrieved state-action pairs taken from *PQueue* for the planning update function, in the simulation, this planning number was set to 5.

The proposed Dyna-QPC agent is considerably faster than both the Q-learning and the Table-based Dyna-Q agents. For example, the Dyna-QPC reached the goal within 500 steps after about 50 training episodes. For the same number of training episodes, the Table-based Dyna-Q agent took about 1000 steps to reach the goal. That is about twice the number of steps taken by the Dyna-QPC agent. The Q-learning algorithm was unable to reach the goal in this simulation, even after 200 training episodes.

The simulation was conducted by using a sampling interval of 0.192 seconds. The simulation results reveal that the robot took 1.6 seconds to traverse 500 steps to accomplish the task. Table 1 summarizes the learning time required for each of the robot navigation algorithms. The data in Table 1 indicate that the proposed Dyna-QPC learning architecture is capable of achieving optimal results more rapidly than the other two methods.

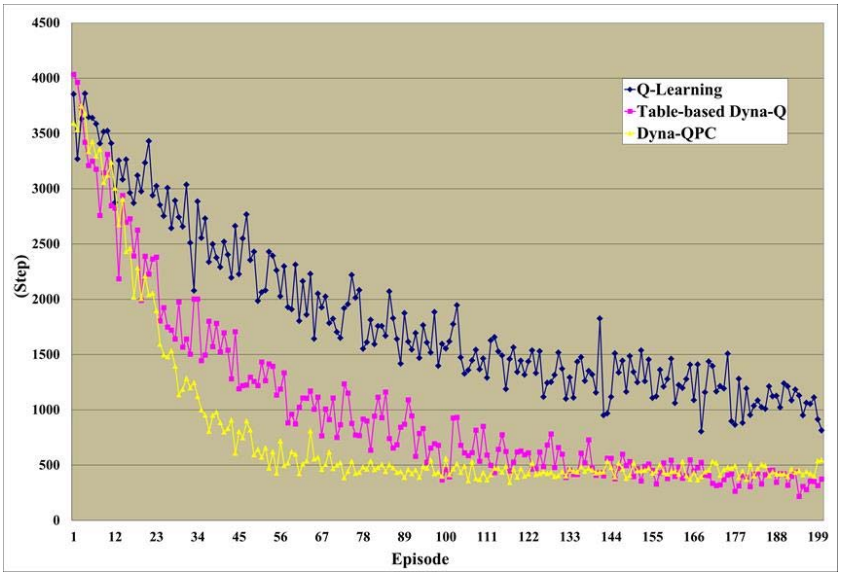


Fig. 13 Simulation results

Table 1 Training time for Q, Table-based Dyna-Q and Dyna-QPC

Method	**Training Time**
Q-learning	Unable to meet conditions
Table-based Dyna-Q	32193 seconds ≈ 8.9 hours
Dyna-QPC	18618 seconds ≈ 5.2 hours (41% faster than Table-based Dyna-Q)

4.4 The Experiment

The experimental system comprises a differential-wheeled mobile robot, called UBot, which is equipped with a gyro, a notebook computer, and a Cricket indoor positioning system, as illustrated in Fig. 14. The Dyna-QPC agent was implemented and used to control the Ubot. The gyro provides the system with the robot orientation. The Cricket system gives the robot planar coordinates in the working space. Cricket is an indoor location system and provides accurate location information between 1 cm and 3 cm [18].

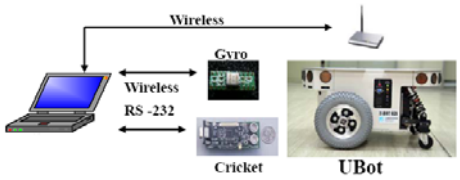


Fig. 14 Experiment system architecture

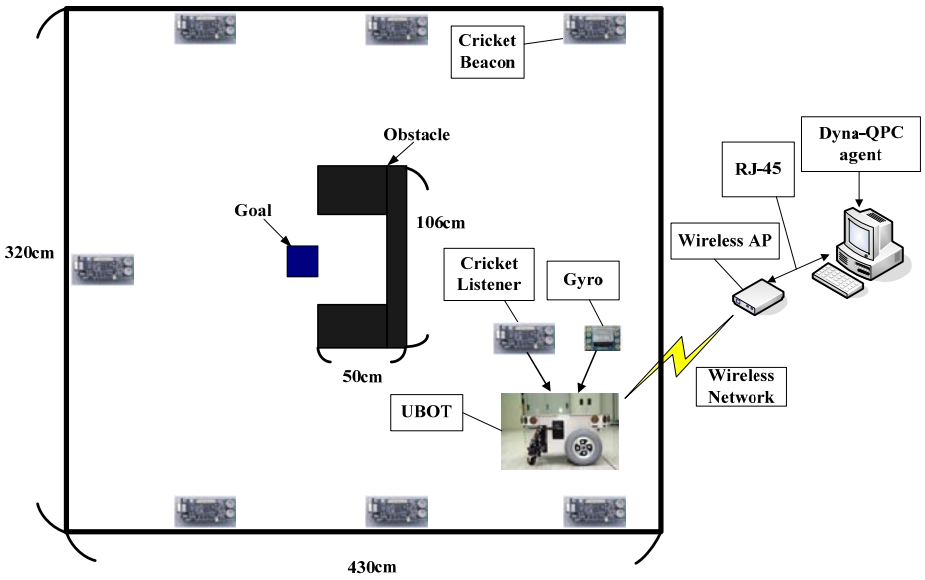


Fig. 15 Experiment workspace alignment and installation

The most common way to use Cricket is to deploy actively transmitting beacons on the walls and/or the ceiling around the workspace, and then attach listeners to host devices whose location needs to be monitored. Hence, the Cricket listener is mounted on the mobile robot and the Cricket beacons are deployed on the walls of the workspace. Fig. 15 displays the deployment of the Cricket beacons and the experimental workspace installation and alignment. There were some differences between the simulated and the experimental workspace as tabulated in Table 2.

In our experiment, $\varepsilon = 0.1$ was set. State, $s = \{x, y, \theta, \text{IRs}\}$, was encoded to be {3-bit, 3-bit, 4-bit, 3-bit} which meant that there were 8192 possible states in the state space. β and γ were the same as in the simulation. The CMAC used 16 memory tilings to store the related one-sixteenth output.

4.5 Discussion of Experimental Results

The experiment consisted of 5 training sets, each of which has 20 training episodes. A training episode was terminated under one of two conditions: either the robot reached the goal or the robot exceeded 200 steps. The average number of steps taken by the robot during each training episode was used to calculate the average for each training set. Fig.16 presents the experimental results in that the sampling interval was set to 0.192 seconds. The total time required for the agent to learn an optimal policy was 1.12 hours or 2019 steps. Following this initial training period, the agent was able to navigate the robot from any starting position in the workspace to the goal in about 50 steps or 1.6 minutes. Some snapshots of one of the training episodes are presented in Fig. 17.

Table 2 Differences between simulation and experimental conditions

	Simulation	Experiment
Workspace	8m x 8m	4.3m x 3.2m
Robot	Pioneer (simulation)	Ubot
IR sensor	100 cm	30 cm
Coordinate	GPS	Cricket
Angle	GPS	Gyro
State	262,144	8192

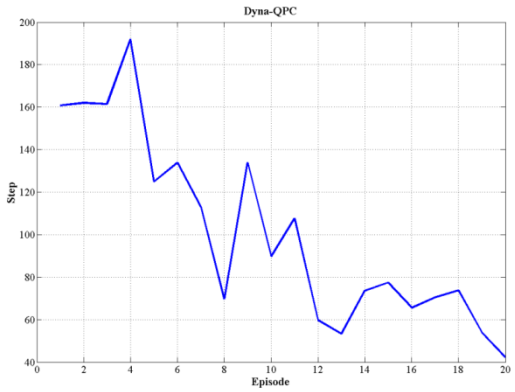
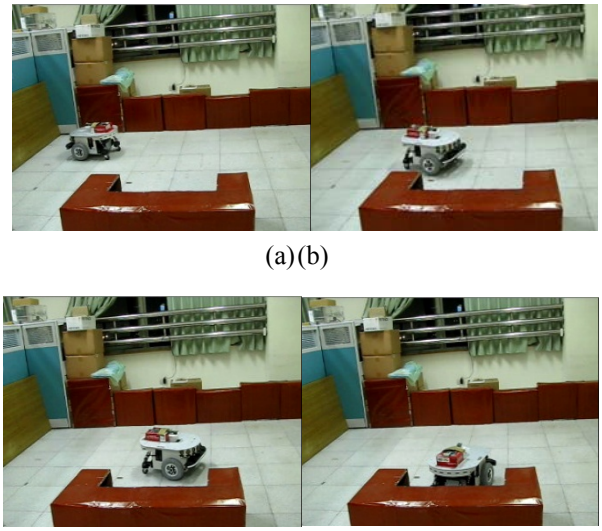


Fig. 16 Experimental results using Dyna-QPC

5. CONCLUSION

The paper has developed a rapid learning algorithm, Dyna-QPC, to overcome the difficulty caused by the original Q-learning algorithm, which could not train the robot to reach its goal within an acceptable time limit. In Dyna-QPC, the CMAC technique has been adopted as a model approximator. CMAC takes as its input a state-action pair, (s, a), and outputs a successive state-reward pair, (s', r), and a predecessor state-action pair, ($\underline{s}$, $\underline{a}$). The search control function uses the prioritized sweeping technique to retrieve relevant state-action pairs. The Dyna-QPC agent retrieves these influential simulated experiences and performs the planning update during the time period between agent direct RL

update. The simulation and experiment results demonstrate the performance of our design is superior in terms of learning time when compared to both the original Q-learning method and the Table-based Dyna-Q method.



(a)(b)

(c) (d)

Fig. 17 Snapshots of one episode (a) start, (b) one-third of the episode, (c) two-thirds through the episode, (d) end.

It is worth noting that by incorporating the CMAC into the Dyna-QPC agent, two problems can be solved. Firstly there is a computational cost down advantage and secondly we can recognize and resolve cyclic paths, as described in sub-section 3.2. One limitation must be addressed before the proposed method can be applied to real-world tasks is that the sensory data describing the robot's state should be reliable. However, the Cricket system can provide accurate robot position information only when the robot is stationary. The more the robot is moving around, the less accurate the position data of the robot is obtained from Cricket. This retards the motion of the robot and seriously jeopardizes the goal of reducing learning time. The agent has to wait for stable position data to be received from Cricket before it can transmit the next action command to the robot.

Nevertheless, the experimental results of the Dyna-QPC agent, as shown in Fig. 16, pay homage to the efficacy of our design. The agent took about 1.12 hours of training, and after that it was able to guide the robot to the goal within 1.6 minutes from any starting location in the experiment workspace. The reduced learning time is further supported by the simulation results and demonstrates that the Dyna-QPC agent can significantly shorten the distance between the lab experiment and real-world application.

REFERENCES

1. R. S. Sutton and A. G. Barto, *Reinforcement Learning An Introduction*, Cambridge,

Mass., MIT Press, 1998.
2. R. S. Sutton, “Dyna, an integrated architecture for learning, planning and reacting,” *Working Notes of the 1991 AAAI Spring Symposium on Integrated Intelligent Architectures* and *SIGART Bulletin 2,* pp. 160-163, 1991.
3. A. W. Moore and C. G. Atkeson, “Prioritized sweeping: reinforcement learning with less data and less real time,” Machine Learning, 13, pp 103-130, 1993.
4. C. J. C. H. Watkins, and P. Dayan, Technical note: Q-Learning, Machine Learning, 8(3-4): pp. 279-292, 1992.
5. J. S. Albus, “A new approach to manipulator control: the cerebellar model articulation controller (CMAC),” Trans. ASME, J. Dynamic Syst. Meas., Contr., Vol. 97, pp. 220-227, Sept. 1975.
6. K.-S. Hwnag, J.-Y. Chiou and T.-Y. Chen **“**Reinforcement learning in zero-sum Markov games for robot soccer systems**,”** IEEE International Conference on Networking, Sensing and Control, Vol. 2, pp: 1110 – 1114, 2004.
7. C. Ye, N.H.C. Yung, and D. Wang, ”A fuzzy controller with supervised learning assisted reinforcement learning algorithm for obstacle avoidance,**”** IEEE Transactions on Systems, Man and Cybernetics, Part B, , Vol. 33 , Issue: 1 , pp. 17–27, Feb. 2003.
8. M. C. Choy, D. Srinivasan, R.L. Cheu ,”Cooperative, hybrid agent architecture for real-time traffic signal control,**”** IEEE Transactions on Systems, Man and Cybernetics, Part A, Vol. 33, Issue: 5, pp. 597–607, Sept. 2003.
9. K. Iwata, K. Ikeda and H. Sakai, “A new criterion using information gain for action selection strategy in reinforcement learning,**”** IEEE Transactions on Neural Networks, Vol. 15, Issue: 4, pp. 792–799, July 2004.
10. E. Zalama, J. Gomez, M. Paul and J.R. Peran, “Adaptive behavior navigation of a mobile robot,**”** IEEE Transactions on Systems, Man and Cybernetics, Part A, Vol. 32, Issue: 1, pp. 160-169, Jan. 2002.
11. J. Si and Y.-T. Wang, “Online learning control by association and reinforcement,**”** IEEE Transactions on Neural Networks, Vol. 12, Issue: 2, pp. 264–276, March 2001.
12. G. H. Kim and C.S.G. Lee, “Genetic reinforcement learning approach to the heterogeneous machine scheduling problem,**”** IEEE Transactions on Robotics and Automation, Vol. 14, Issue: 6, pp. 879-893, Dec. 1998.
13. J. W. Lee, J. Park, Jangmin O, J. Lee, and E. Hong, “A Multiagent Approach to Q-Learning for Daily Stock Trading,” IEEE Transactions on Systems, Man, and Cybernetics, Part A, Systems and Humans, Vol. 37, No. 6, pp. 864-877, Nov. 2007.
14. C.-F. Juang and C.-M. Lu, “Ant Colony Optimization Incorporated with Fuzzy Q-Learning for Reinforcement Fuzzy Control,” IEEE Transactions on Systems, Man, and Cybernetics, Part A, Systems and Humans, Vol. 39, No. 3, pp. 597-608, May 2009.
15. D. Bertsekas. *Dynamic Programming and Optional control*, Athena, MA, 1995.
16. D. Bertsekas. *Neuro-Dynamic Programming*, Athena, MA, 1996.
17. K.-S. Hwang and Y.-P. Hsu, “An Innovative Architecture of CMAC,” IEICE Trans. Electron., Vol. E87-C, No. 1, pp. 81-93, Jan. 2004.
18. http://cricket.csail.mit.edu/
19. R. E. Bellman, *Dynamic Programming*, Princeton University Press, Princeton,

1957.
20. M. A. Arbib (Editor), *The Handbook of Brain Theory and Neural Networks*, Second Edition, MIT Press, 2003.
21. L. Ljung, T. Soderstrom, *Theory and Practice of Recursive Identification*, MIT Press, Cambridge, MA, 1983.
22. F. J. Torres, M. Huber, "Learning a Causal Model from Household Survey Data by Using a Bayesian Belief Network," Journal of the Transportation Research Board, Vol. 1836, pp. 29-36, 2003.
23. T. Hester and P. Stone, "Learning and Using Models," in Marco Wiering and Martijn van Otterlo, editors, Reinforcement Learning: State of the Art, Springer Verlag, Berlin, Germany, 2011.
24. T. Hester, M. Quinlan, and P. Stone, "Generalized Model Learning for Reinforcement Learning on a Humanoid Robot," in IEEE International Conference on Robotics and Automation (ICRA 2010), Anchorage, Alaska, May 2010.
25. H. H. Viet, P. H. Kyaw, T. Chung, "Simulation-Based Evaluations of Reinforcement Learning Algorithms for Autonomous Mobile Robot Path Planning," In: Proceedings of International Conference on Intelligent Robotics, Automations, Telecommunication Facilities, and Applications, pp.467- 476, 2011.
26. M. Santos, J. A. Martín H., V. López, G. Botella, "Dyna-H: A Heuristic Planning Reinforcement Learning Algorithm Applied to Role-playing Game Strategy Decision Systems," Knowledge-Based Systems, vol. 32, pp. 28-36, 2012.
27. H. H. Viet, S. H. An and T. C, Chung, "Extended Dyna-Q Algorithm for Path Planning of Mobile Robots," Journal of Measurement Science and Instrumentation, vol. 2, no. 3, pp. 283-287, Sep. 2011.

Yuan-Pao Hsu (徐元寶) received the Ph.D. degree in Department of Electric Engineering, National Chung Cheng University, Minhsiung, Taiwan, in 2004. He is working in Department of Computer Science and of Information Engineering, National Formosa University, Yunlin, Taiwan. His current research interests include hardware / software co-design, image processing, machine learning, and control system.

Wei-Cheng Jiang (蔣惟丞) received the M.E. in Institute of Electro-Optical and Materials Science from National Formosa University in 2009. He is now a Ph.D. student in Department of Electric Engineering, National Chung Cheng University, Minhsiung, Taiwan.