Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2016) 24: 1747 – 1767
© TÜBİTAK
doi:10.3906/elk-1311-129

Reinforcement learning-based mobile robot navigation

Nihal ALTUNTAŞ[1,*], Erkan İMAL[2], Nahit EMANET[1], Ceyda Nur ÖZTÜRK[1]
[1]Department of Computer Engineering, Faculty of Engineering, Fatih University, İstanbul, Turkey
[2]Department of Electrical and Electronics Engineering, Faculty of Engineering, Fatih University, İstanbul, Turkey

Received: 15.11.2013 • **Accepted/Published Online:** 30.06.2014 • **Final Version:** 23.03.2016

Abstract: In recent decades, reinforcement learning (RL) has been widely used in different research fields ranging from psychology to computer science. The unfeasibility of sampling all possibilities for continuous-state problems and the absence of an explicit teacher make RL algorithms preferable for supervised learning in the machine learning area, as the optimal control problem has become a popular subject of research. In this study, a system is proposed to solve mobile robot navigation by opting for the most popular two RL algorithms, Sarsa(λ) and Q(λ). The proposed system, developed in MATLAB, uses state and action sets, defined in a novel way, to increase performance. The system can guide the mobile robot to a desired goal by avoiding obstacles with a high success rate in both simulated and real environments. Additionally, it is possible to observe the effects of the initial parameters used by the RL methods, e.g., λ, on learning, and also to make comparisons between the performances of Sarsa(λ) and Q(λ) algorithms.

Key words: Reinforcement learning, temporal difference, eligibility traces, Sarsa, Q-learning, mobile robot navigation, obstacle avoidance

1. Introduction

With the advancement of technology, people started to prefer machines instead of human work in order to increase productivity. In the beginning, machines were only used to automate work that did not require intelligence. However, the invention of computers urged people to consider machine learning (ML). Today, artificial intelligence (AI) continues to be a subject of study to provide machines with learning abilities [1]. It is important to understand the nature of learning in order to achieve the goal of intelligent machines. Although there is a great number of algorithms that were developed as supervised and unsupervised learning methods in the ML field, the fundamental idea of reinforcement learning (RL) usage is that of learning from interaction. In order to obtain interaction ability, various sensors are mounted on machines, including infrared, sonar, inductive, and diffuse sensors.

The term 'reinforcement' was first used in psychology and is built on the idea of learning by trial and error that appeared in the 1920s. Afterwards, this idea became popular in computer science. In 1957, [2] introduced dynamic programming, which led to optimal control and then to Markov decision processes (MDPs). Although dynamic programming is a solution for discrete stochastic MDPs, it requires expensive computations that grow exponentially as the number of states increases. Therefore, temporal difference brought a novel aspect to RL, and in 1989 Q-learning explored by [3] was an important breakthrough in the AI field. Finally, the Sarsa

*Correspondence: nihaltuntas@gmail.com

algorithm was introduced by [4] in 1994 as 'modified Q-learning'. There were other studies to enhance RL techniques for faster and more accurate learning results [1,5–7].

RL techniques have started to be preferred as learning algorithms since they are more feasible and applicable than other techniques that require prior knowledge. The RL approach has been used for many purposes such as feature selection of classification algorithms [8], optimal path finding [5,9,10], routing for networks [11], and coordination of communication in multiagent systems [12,13]. A number of studies using RL to provide effective learning for different control problems are explained in the following paragraphs.

In 2009, a new RL algorithm, $E_x < \alpha > (\lambda)$, was developed by [14] to deal with the problems of using continuous actions. This algorithm is based on the k-nearest neighbors approach dealing with discrete actions and enhances the kNN-TD(λ) algorithm.

Due to the complexity of the navigation problem, RL is a widely preferred method for controlling mobile robots. For example, [15] used prior knowledge within Q-learning in order to reduce the memory requirement of the look-up table and increase the performance of the learning process. That study integrated fuzzy logic into RL for this purpose, and the Q-learning algorithm was applied to provide coordination among behaviors in the fuzzy set. Furthermore, an algorithm called 'piecewise continuous nearest sequence memory', which extends the instance-based algorithm for discrete, partially observable state spaces, and the nearest sequence memory [16] were presented in [17]. Another study using the neural network approach in RL for obstacle avoidance of mobile robots was performed in a simulated platform in [18]. The main aim of this study was simply to avoid obstacles while roaming; there was no specific goal to achieve. The reason for using neural networks instead of a look-up table was to minimize the required space and to maximize learning performance .

The paper, which is an extension of work done in [19], comprises five sections. Section 2 gives information about RL for navigation problems and Section 3 explains the implementation details of the proposed system. The experimental results are given in Section 4 and the paper is concluded with Section 5.

2. Reinforcement learning for navigation problem

RL aims to teach the agent how to behave when placed in an unknown environment by achieving the optimal Q-value function that gives the best results for all states. The agent uses rewards received from the environment after action selections for each state to update Q-values for convergence of optimality.

Not knowing the environment in an RL algorithm causes a trade-off between exploration and exploitation. Selecting the action with the greatest estimated value means that the agent exploits its current knowledge. Instead, selecting one of the other actions indicates that the agent explores in order to improve its estimate of the values of those actions. Exploitation maximizes the reward in the short term, yet does not guarantee the maximization of the total reward in the long run. On the other hand, although exploration reduces short-term benefit, it produces a greater reward in the long run, because after the agent has explored better actions, it can start to exploit them. The agent can neither explore nor exploit exclusively without failing at the task, and it cannot both explore and exploit in one selection. Therefore, it is vital to balance exploration and exploitation to converge the optimal value function. The most common method for balancing this trade-off is the ε-greedy method. In this method, the action with the greatest estimated value is called the greedy action, and the agent usually exploits its current knowledge by selecting the greedy action. However, there is also a chance of probability ε for the agent to explore by randomly selecting one of the nongreedy actions. This type of action selection is called the ε-greedy method [20].

RL can be passive or active, depending on whether it uses a fixed policy or not. Passive learning aims

only to learn how good the fixed policy is. The only challenge is that the agent knows neither the transition model nor the reward function since the environment is unknown. The agent executes a set of trials using the fixed policy to calculate the estimated value of each state. Distinct from passive learning, active learning also aims to learn how to act in the environment. Since the agent does not know either the environment or the policy, it has to learn the transition model in order to find an optimal policy. However, in this case, there is more than one choice of actions for each state, because the policy is not fixed [21]. Here, the balance between exploration and exploitation is important for deciding which action to choose.

Certain solution classes, developed to find optimal Q-values as quickly and accurately as possible, are explained in the following subsections.

2.1. Dynamic programming

Dynamic programming (DP) algorithms are based on the assumption of a complete environment model; in other words, they are a model-based solution class. Although DP algorithms are generally impractical due to their perfect model assumption and their great computational expense, there have been studies to improve and implement the DP approach in the RL field [22–24]. Additionally, DP is theoretically important, since it provides an important foundation for other solution methods. It can be said that other types of methods have the purpose of achieving the same effect as DP with less computation and without a complete model of the environment. Many solution methods are based on DP and its algorithms, such as policy iteration.

Policy iteration consists of two interacting processes, policy evaluation and policy improvement. Policy evaluation calculates a consistent value function for the current policy, whereas policy improvement finds the greedy policy for the value function. After policy evaluation, the policy is no longer greedy for the modified value function, and policy improvement makes the value function inconsistent with the greedy policy. However, after a certain point, both the evaluation and the improvement processes stabilize, meaning that the value function and the policy become optimal.

During the policy evaluation process, DP basically calculates value functions using the Bellman equation [25]. The Bellman equation is used to update the values of each state iteratively. Additionally, the main idea of policy improvement is to find if there is a better policy than the current one.

Furthermore, during policy iteration, letting policy evaluation and policy improvement processes interact with each other to find a joint solution is termed generalized policy iteration (GPI) [20]. The following two subsections explain different approaches to solving these problems.

2.2. Monte Carlo methods

Monte Carlo (MC) methods use experience from the interaction with the environment. Instead of assuming complete knowledge of the environment, MC methods average sample values obtained from experience that consists of episodes. The only assumption is that each episode eventually terminates without depending on selected actions. Estimation update is performed after the termination of each episode but not after each step.

The main idea introduced by DP, i.e. GPI, is used by MC. The only difference is that the update of value estimation depends on experience instead of estimation of another state. Therefore, the computational complexity of estimating a value for a state does not depend on the number of states.

During the policy evaluation process, MC uses sample values obtained after state visits in episodes to estimate the expected value.

Since the environment is unknown, it is important to ensure that all state–action pairs are visited and

estimated to compare all possible action values for each state for successful learning. This is the exploitation/exploration trade-off problem. There are two approaches to solving this problem: *on*-policy and *off*-policy. *On*-policy methods evaluate and improve the same policy used for decisions in experiences. To ensure that all actions are selected in order to learn better choices, ε-greedy policies are used, as mentioned above. On the other hand, *off*-policy methods use two different policies: behavior policy and estimation policy. Behavior policy is used to make decisions and needs to ensure that all actions have a probability to be selected to explore all possibilities. Estimation policy is the evaluated and improved policy, and since it does not affect decisions, this policy can be totally greedy.

2.3. Temporal-difference learning

Temporal-difference (TD) learning divides the learning problem into a prediction problem and a control problem, as described by GPI. As a solution to these problems, TD follows an approach that combines the DP and MC methods. TD methods can learn from experience without any need of a model; hence, TD is model-free, similarly to MC. However, it is not necessary for TD to wait until the end of the episodes for an update of the value functions. Whereas MC uses actual values to update estimates, TD uses immediate rewards and estimates of successor states for value estimation. Another advantage of TD over MC is that if there is a limited amount of experience that is not sufficient to generalize the whole problem, MC can only find an optimal policy in a limited way, whereas TD can converge to an optimal policy, representing the problem completely. Additionally, two different approaches are applied to deal with the exploration/exploitation trade-off: *on*-policy and *off*-policy.

Although there are several other TD methods using the *on*-policy approach, the most common one is Sarsa, which uses the ε-greedy method to ensure that all actions are possible. This algorithm uses quintuple parameters $(s,\ a,\ r,\ s\prime, a\prime)$, from which the Sarsa name comes. For state s, the agent decides and performs action a, then it observes an immediate reward r and a new state $s\prime$, and then it decides another action $a\prime$ for state $s\prime$.

The most popular TD algorithm is Q-learning, which is an *off*-policy method and was first introduced in [3]. The Q-learning algorithm updates Q-values with the greedy method after selecting action a for state s and receiving reward r and next state $s\prime$ by behavior policy, which is ε-greedy.

Sarsa, in a way, is an enhancement of Q-learning; thus, it generally learns faster. Besides Sarsa, there have been other studies to improve the learning performance of Q-learning [5,10,26]. Additionally, different methods using the TD approach have also been studied [27,28], but the most common ones are the Sarsa and Q-learning algorithms.

2.4. Eligibility traces

Eligibility traces are one of the unified algorithms developed to embody the key ideas of TD learning and MC methods. On one hand, MC methods use real values returned after a complete episode; consequently, they suffer from the requirement of waiting until the termination of episodes. On the other hand, TD methods use a one-step return value and update the value estimation of only one state–action pair, which causes the problem of slow convergence for large state–action spaces. However, it is possible to update the estimated values of recently visited state–action pairs.

Eligibility traces use two different ways to update the estimations of Q-values: on-line and off-line updating. In the former updating, value estimates are updated during the episodes, whereas in the latter, updates are accumulated and used after termination of the episodes. When performances of on-line and off-line

updating methods are compared, empirical results show that on-line methods converge to the optimal policy faster than off-line methods [20]. Thus, the proposed system uses on-line updating.

In order to implement the TD(λ) method, an additional variable $e_t(s,a)$ is defined as the eligibility trace for all state–action pairs at time t. The trace-decay parameter λ in the TD(λ) algorithm refers to the use of eligibility traces for updating the values of recently visited states. At each step, eligibility traces for all states are decayed by $\gamma\lambda$, defining the recently visited traces where γ is a discount factor. Two kinds of traces, accumulated traces and replacing traces, are used to record the traces of state–action pairs. The accumulated traces increase the eligibility trace $e_t(s,a)$ by 1 for each visit of state–action pairs, while the replacing traces reset the eligibility trace to 1. The replacing traces can also reset the other actions available in the visited state to 0. Generally, replacing traces have a better performance than accumulating traces [20]. Eq. (1) defines $e_t(s,a)$, the replacing traces used by the proposed system:

$$e_t(s,a) = \begin{cases} 1 & \text{if } s = s_t \text{ and } a = a_t \\ 0 & \text{if } s = s_t \text{ and } a \neq a_t \\ \gamma\lambda e_{t-1}(s,a) & \text{if } s \neq s_t \end{cases} \quad \text{for all } s,a \tag{1}$$

where $e_t(s,a)$ is the eligibility trace for all state–action pairs at time t, λ is the trace-decay parameter referring to the use of eligibility traces in order to update the values of the recently visited states, and γ is the discount factor describing the effect of future rewards on the value of the current state. Both λ and γ range between 0 and 1.

The one-step TD error calculated by Eq. (2) is used to update the values of all recently visited state–action pairs by their nonzero traces using Eq. (3):

$$\delta_t = R_a\left(s, s^{'}\right) + \gamma Q_t\left(s^{'}, b\right) - Q_t(s,a) \tag{2}$$

$$\Delta Q_t(s,a) = \alpha\delta_t e_t(s,a), \qquad \text{for all } s,a \tag{3}$$

where α is the step-size parameter, also known as the learning rate and ranging between 0 and 1; $R_a(s,s\prime)$ is the reward for transition from state s to state $s\prime$ applying action a; and $Q^{\pi}(s,a)$ is the expected value for selecting action a in state s under policy π at time t. The value of $Q_t(s\prime,b)$ depends on the preferred algorithm, which can be Sarsa(λ) or Q(λ), explained in the following subsections.

2.4.1. Sarsa(λ)

The eligibility trace version of Sarsa is known as Sarsa(λ), and Eq. (4) defines the update step of the Sarsa(λ) algorithm:

$$Q_{t+1}(s,a) = Q_t(s,a) + \Delta Q_t(s,a), for all s,a \tag{4}$$

where $\Delta Q_t(s,a)$ is defined by Eq. (3), and $Q_t(s',b) = Q_t(s',a')$ in Eq. (2), since Sarsa(λ) is an *on*-policy method. Figure 1 shows the pseudocode for the complete Sarsa(λ) algorithm using ε-greedy policy, on-line updating, and replacing traces.

2.4.2. Q(λ)

The proposed system uses Watkins's Q(λ), which is a Q-learning algorithm into which the eligibility traces are integrated [3]. Since Q(λ) is an *off*-policy algorithm, only traces of greedy actions are used. Thus, the

eligibility traces are set to 0 when a nongreedy action is selected for exploration. Q(λ) uses Eq. (4) to update Q-values with $Q_t(s',b) = max_a Q_t(s',a)$ in Eq. (2), because it uses the greedy method in the policy improvement process. Figure 2 shows the pseudocode for the complete Q(λ) algorithm using ε-greedy behavior policy, greedy estimation policy, on-line updating, and replacing traces.

```
Initialize Q(s, a) ← 0 and e(s, a) ← 0 for all s, a
Repeat (for each episode):
    Initialize s, a
    Repeat (for each step of episode):
        Take action a, observe r, s'
        Choose a' from s' using policy derived from Q (ε-greedy)
        δ ← r + γQ(s', a') – Q(s, a)
        For b in Action Set:
            e(s, b) ← 0
        e(s, a) ← 1
        For all s in State Set, a in Action set:
            Q(s, a) ← Q(s, a) + αδe(s, a)
            e(s, a) ← γλe(s, a)
        s ← s'
        a ← a'
    until s is terminal
```

Figure 1. Pseudocode for Sarsa(λ) algorithm used by the proposed system.

```
Initialize Q(s, a) ← 0 and e(s, a) ← 0 for all s, a
Repeat (for each episode):
    Initialize s, a
    Repeat (for each step of episode):
        Take action a, observe r, s'
        Choose a' from s' using policy derived from Q (ε-greedy)
        a* ← argmax_b Q(s', b)
        δ ← r + γQ(s', a*) – Q(s, a)
        For b in Action Set:
            e(s, b) ← 0
        e(s, a) ← 1
        For all s in State Set, a in Action set:
            Q(s, a) ← Q(s, a) + αδe(s, a)
            If a' = a*, then e(s, a) ← γλe(s, a)
            Else            e(s, a) ← 0
        s ← s'
        a ← a'
    until s is terminal
```

Figure 2. Pseudocode for Q(λ) algorithm used by the proposed system.

3. Implementation

First, it is necessary to design the system properly in order to achieve the main purpose of autonomous navigation. The mobile agent is supposed to reach the goal by following an optimal path and avoiding obstacles in the environment after a set of trials. The proposed system uses the on-line updating technique, meaning that the robot improves its guess about the environment during the trials and does not wait until the termination of the trials to update the guess. As the robot performs more trials and gains more information about its environment, it exhibits better performance for navigation purposes.

Although there are a few applications using RL algorithms for mobile robot navigation, most of them are implemented only on simulated platforms instead of real mobile robots. This system implements Sarsa(λ) and Q(λ) on a real platform by using a state set, an actions set, and a reward function described in a novel way.

In the implementation of the system, Q-values used by the RL algorithms are in tabular form and are represented as matrices causing many matrix calculations during the execution. Since these calculations can be made in a concise way with the help of MATLAB, the system was implemented as a MATLAB application (http://www.mathworks.com). Experimentation of the system was performed on a Robotino®, which is an educational robot by Festo Company (http://www.festo-didactic.com). Robotino MATLAB API was integrated into the system in order to communicate with the Robotino.

3.1. Hardware platform

Robotino has a built-in PC working at 500 MHz and using a Linux-based operating system. Although the PC embedded in Robotino is insufficient for executing the proposed system, it provides W-LAN and other services to communicate with another computer by transferring sensor values and receiving the necessary orders. Therefore, the system is executed on an external computer and uses Robotino MATLAB API for communication. The technical properties of the external computer on which the system is executed are given in Section 4.

In order to make the Robotino perform select actions properly, it is necessary to send related orders for the actions according to the degrees of freedom (DoF) of the Robotino (Figure 3). For instance, moving forward and left means moving through the x-axis and y-axis in a positive direction, respectively; therefore, when the Robotino is ordered to turn θ° , it turns to the left, and for $-\theta^\circ$, it turns right. The world reference frame that is stated according to the initial location is used to navigate the Robotino.



Figure 3. Degrees of freedom (DoF) of the Robotino.

Although Robotino has several other sensors, only three are designed to be used in the system: the bumper, the odometer, and the camera, with a resolution of 320 $\times$ 240 pixels. The bumper is used to detect collisions with the obstacles. The odometer is used to obtain the location of the robot with respect to its first location, and the obstacle locations are given to the system as prior knowledge.

3.2. Methodology

The solution classes explained in Section 2 provide several alternative approaches for the RL algorithm. The selection of an approach among its alternatives, i.e. model-based or model-free [29], affects the performance. In this study, two model-free methods, Sarsa(λ) and Q(λ), were implemented for the mobile robot to navigate in an unknown environment with no prior knowledge.

Indeed, the main goal was to find optimal Q-values during navigation. Thus, how Q-values were represented in the system was important. Although it is possible to use certain learning algorithms for

representation [30,31], such as neural networks, a matrix representation was used in this study. Values of related state–action pairs were updated in the matrix for each step.

Another important issue for a good learning performance when using RL is how to define the reward function, the state set, and the action set. In this study, although the reward function could return 4 different values, the state and the action set had 81 and 3 different members, respectively. Details are explained in the next subsection.

3.3. Software implementation

Figure 4 shows a general block diagram indicating the processing and argument modules of the proposed system. Arrows in the figure imply providing information for another module, and the modules in light blue are argument modules. It is important to define system arguments appropriately so as to have a good performance in learning. The following list explains the modules by briefly constructing the system.

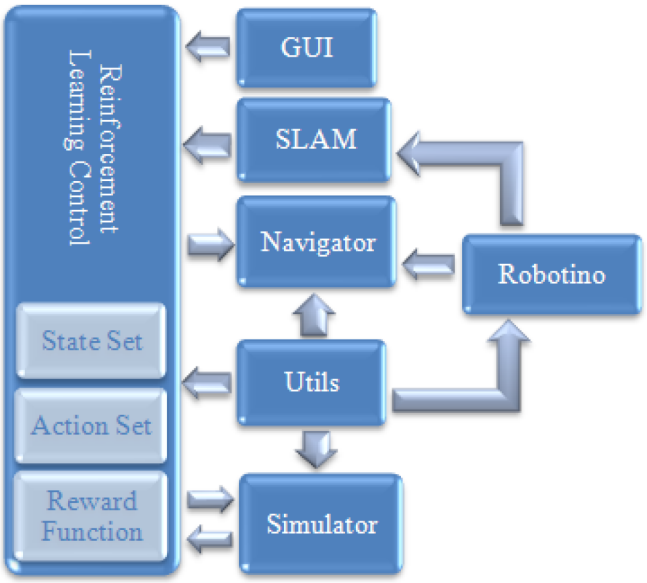


Figure 4. Block diagram of the proposed system.

3.3.1. Action set

Since the system is used in a continuous environment, it is important to decide the action set. The actions selected should be small enough to be executed in one step and sufficient enough to let the robot navigate to its goal. The action set used in the system consists of three actions: 'move forward with linear velocity v', 'turn right with angular velocity w', and 'turn left with angular velocity w'. The default values of variables v and w are 150 mm/s and 15 $^\circ$ /s, respectively, though the user can change those values through the GUI module.

In the simulation platform, it is better to perform actions considering the movement amount of the robot instead of its velocity. Therefore, the simulated action set is a little different from the real-platform action set. The three simulated actions are 'move forward for distance d', 'turn left for angle θ', and 'turn right for angle θ'. While the d and θ variables can also be changed by the user, the default values are 100 mm and 10° , respectively.

3.3.2. Graphical user interface

This module provides an interface to the user to execute the system and test its algorithms. The user can tune initial parameters, such as the learning rate, and select the embedded algorithm executed in the RL control module to observe how they affect the learning performances.

Figure 5 shows the interface of the system, which consists of 6 main parts: navigation path, reward function, initial parameters, info box, image frames, and button groups.

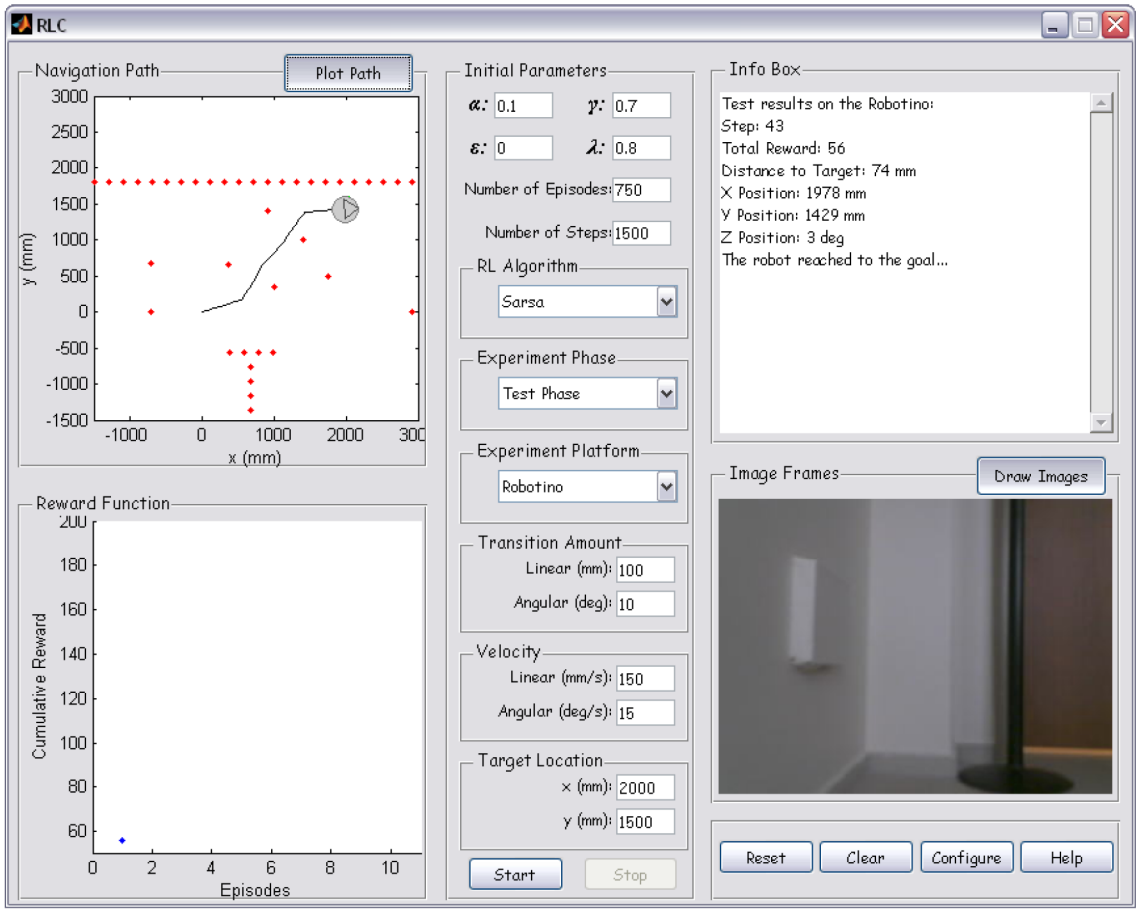


Figure 5. GUI of the proposed system.

The navigation path part plots the target as a green circle, the obstacles in the environment as red demands, the robot as a gray-filled circle, and its path during the episode as a black line. This part can be made inactive to increase the learning speed by avoiding unnecessary graphical processes. After termination of the episodes, the robot path is plotted to give an indication to the user about the learning process. The reward function part plots the cumulative reward gathered from the environment after each episode. When the agent learns the optimal behavior well enough, the reward function plotted in this part is expected to become stable. The initial parameters part gives different options to the user for controlling the system, e.g., selecting the performed algorithm, experiment phase, and experimental platform. The image frames part displays the images captured by the camera of the robot at a rate of 5 Hz. This part is disabled for simulated platforms and can be made inactive to increase the speed of the learning process in real experiments. The info box part informs the user after each episode. This part also gives information about what the parameters are when the

user clicks the 'help' button at the right bottom corner. Finally, the button groups include four buttons. The 'help' button gives information about the initial parameters and their features. The 'reset' button resets all parameters to their default values. The 'clear' button clears the four parts informing the user: the navigation path, reward function, info box, and image frames parts. Since the system is not capable of detecting obstacles, it is necessary to configure the environment before the learning process. Hence, a 'configure' is added to the interface of the system.

Figure 6 shows the interface for configuration of the environment. The boundary of the environment can be changed, and there are three different obstacle options. 'No obstacle' is the default option, and the system executes in an environment without obstacles. When the 'random obstacles' option is selected, the system generates N random obstacles for the environment, where N can be set by the user. The last option is 'set obstacles', which enables the user to set obstacles manually.

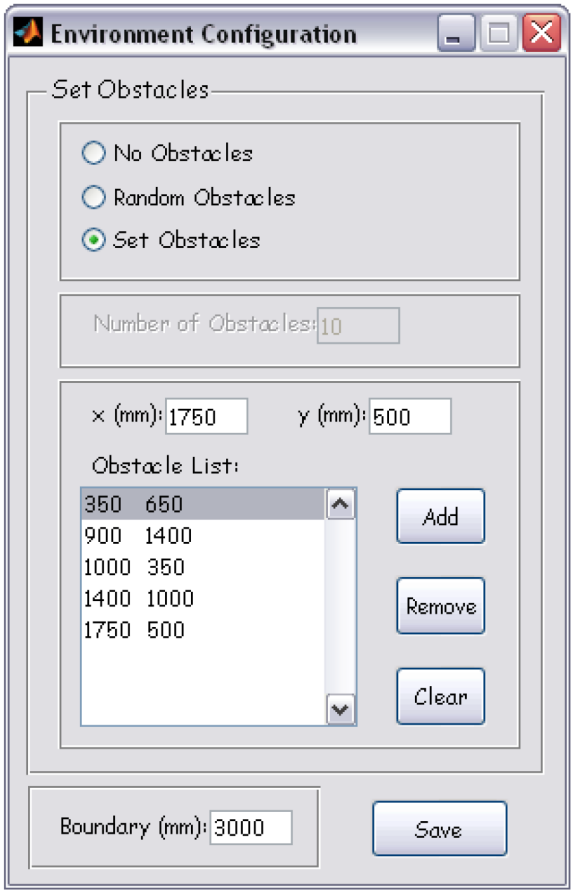


Figure 6. Interface for environment configuration of the system.

3.3.3. Navigator

The navigator module is responsible for performing the actions selected by the reinforcement learning control module. In order to connect to the robot and send the action orders, this module uses the functions of Robotino MATLAB API, which are provided by the Robotino module.

3.3.4. Reinforcement learning control

This is the central module running the selected algorithms in the system. The user can select either the Sarsa(λ) or Q(λ) algorithms through the GUI module. The λ parameter, which can also be changed by the user, is a trace-decay parameter. The initial value of the ε variable can be changed by the user through the GUI module, but it is decayed by 0.99 after each episode as the agent learns more.

This module includes three main argument modules: the action set, the reward function, and the state set modules. These are critical modules that affect the performance of the RL control. It is essential to define these modules well enough to enable the agent to learn as expected.

3.3.5. Reward function

This module indicates the rewards the agent receives from the environment as a result of its actions. In other words, the reward function determines the response of the environment to the actions of the agent. There are four different rewards, defined as +100 for achieving the goal, –100 for obstacle collision, –1 for the actions making the agent get closer to the target, and –2 otherwise.

3.3.6. Simulator

The simulator module combines the navigator, Robotino, and simultaneous localization and mapping (SLAM) modules, which are used for the real platform. When the user selects the simulated platform, the simulator module performs the actions selected by the RL control module and returns the resulting position of the robot during the learning process. Since the system is designed to be stochastic, the simulator module is developed to have 10% Gaussian noise, similar to the real platform.

3.3.7. State set

The great number of states defined in RL algorithms not only increases the complexity of the system, but also makes it harder to sample all the states and consequently to find an optimal policy. However, if the states are not defined well enough, the system can suffer from an insufficient representation of all situations in the environment.

The proposed system discretizes continuous states of the environment, depending on the position of the agent according to the target and the nearest obstacle on the way. For this purpose, it uses four variables: d_{tar}, which is the distance in mm of the unit between the robot and the target; θ_{tar}, which is the orientation in degrees of the robot with respect to the target; d_{obs}, which is the distance between the robot and the closest obstacle; and θ_{obs}, which is the orientation of the robot with respect to the closest obstacle. The distance variables range is [0 mm, 1000 mm, 2000 mm] and the orientation variables range is [–30° , 0° , 30°]. This makes the system be represented by 81 different states. Figure 7 shows the variables defined in the state set. When determining the nearest obstacle, not all obstacles are considered; only the obstacles in the interval of [–45° , 45°] (Figure 7b) with respect to the robot direction are taken into account.

The state of the robot is defined depending on dynamic points, such as obstacle locations and target, which can vary with different experiments. This definition provides the system with the ability to find an optimal Q-value matrix that gives successful results in different environments with various obstacle locations.

4. Experimental results

The proposed system was designed to be executed in both simulated and real platforms. The RL-based control part of all experiments was performed using MATLAB R2007b on a PC with 2.00 GHz Duo CPU, 3 GB RAM,

and Windows XP. In addition to the control part, the same computer was used to implement the virtual robot for the simulated platform and to communicate with the Robotino, as mentioned in Section 3.1. Experiments were performed in two phases: the learning and testing phases. Preferably, the learning phase is experienced only in a simulated platform, since the real robot can be damaged due to collisions with obstacles at the beginning of the phase. It is also hard to relocate the robot between consecutive episodes, but several episodes should be executed for successful learning.

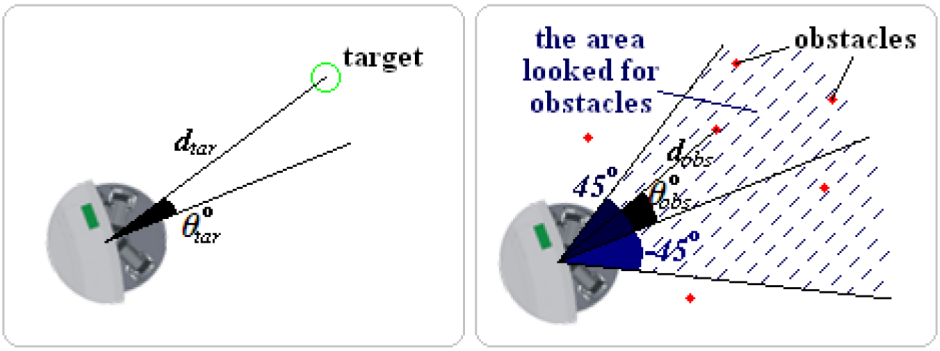


Figure 7. The variables of the state set module are defined by robot distance and orientation, according to (a) the target position represented as d_{tar} and θ_{tar}, and (b) the nearest obstacle on the way, represented as d_{obs} and θ_{obs}.

The system is tested in environments with and without obstacles. As expected, the learning performance is improved in environments without obstacles, since there is no risk of obstacle collisions, which makes the environment much simpler to be learned.

For the simulated platform, a virtual robot is implemented to perform the selected actions at each step of the episodes. In order to perform a realistic simulation, some errors are added to those actions during execution. The main purpose of the simulation is to obtain an optimal Q-value matrix to be used with the real robot.

When there are no obstacles in the environment, the performance of both the Sarsa(λ) and Q(λ) algorithms is similar, and the cumulative reward stabilizes at around 20 episodes with default initial parameter values. However, adding obstacles to the environment makes the learning process more complicated. Thus, the performance of both algorithms decreases. Tables 1 and 2 list the average episode numbers at which the cumulative rewards become stable for the Sarsa(λ) and Q(λ) methods, respectively. Figure 8 illustrates the environment containing 10 obstacles in Tables 1 and 2.

Figure 9 illustrates a chart diagram comparing the results listed in Tables 1 and 2. The x-axis shows the number of experiments whose initial parameters are given in Tables 1 and 2. The y-axis maps the average number of episodes necessary for the convergence of Q-values. Variation of initial parameters affects both Sarsa(λ) and Q(λ) in the same way; thus, both methods perform better or worse for the same parameter values. In addition, Sarsa(λ) mostly converges faster in the first experiments, whereas Q(λ) generally has better performance for the rest of the experiments, as shown in Figure 9.

Figures 10–13 plot line diagrams to illustrate the effects of value changes on the initial parameters α, γ, ε, and λ, respectively. The x-axis shows other parameter values during the experiments, and the y-axis maps the average number of episodes necessary for convergence of Q-values in Figures 10–13. The effects of these four parameters are illustrated in Tables 1 and 2 for both the Sarsa(λ) and Q(λ) methods, which are represented by 2 different shades of blue and purple, respectively. Figures 10 and 11 show that the values of the α and γ parameters influence each other's effect on the performance, e.g., when $\alpha = 0.1$, the performance of

both methods is better for the first 4 experiments and worse for the last 4 experiments in Figure 10. The only difference between the first and last 4 experiments is the value of γ. Figure 12 shows that the value of ε highly influences the learning performance. When ε gets smaller, the policy used by the system becomes greedier, which renders the system unable to explore better ways to reach the target. Finally, Figure 13 shows that λ has different effects for different parameter combinations. Therefore, it is important to tune parameters to get better performance results.

Table 1. Simulated results of Sarsa(λ) algorithm for different initial parameters.

Experiment no.	α	γ	ε	λ	Number of obstacles	Episode no. for stable cumulative reward
1	0.1	0.1	0.1	0.1	0	**28**
2	0.1	0.1	0.1	0.1	10	**186**
3	0.1	0.1	0.1	0.5	0	**15**
4	0.1	0.1	0.1	0.5	10	**255**
5	0.1	0.1	0.5	0.1	0	**113**
6	0.1	0.1	0.5	0.1	10	**412**
7	0.1	0.1	0.5	0.5	0	**123**
8	0.1	0.1	0.5	0.5	10	**372**
9	0.1	0.5	0.1	0.1	0	**15**
10	0.1	0.5	0.1	0.1	10	**80**
11	0.1	0.5	0.1	0.5	0	**22**
12	0.1	0.5	0.1	0.5	10	**184**
13	0.1	0.5	0.5	0.1	0	**145**
14	0.1	0.5	0.5	0.1	10	**264**
15	0.1	0.5	0.5	0.5	0	**109**
16	0.1	0.5	0.5	0.5	10	**285**
17	0.5	0.1	0.1	0.1	0	**65**
18	0.5	0.1	0.1	0.1	10	**155**
19	0.5	0.1	0.1	0.5	0	**58**
20	0.5	0.1	0.1	0.5	10	**80**
21	0.5	0.1	0.5	0.1	0	**185**
22	0.5	0.1	0.5	0.1	10	**249**
23	0.5	0.1	0.5	0.5	0	**213**
24	0.5	0.1	0.5	0.5	10	**354**
25	0.5	0.5	0.1	0.1	0	**86**
26	0.5	0.5	0.1	0.1	10	**249**
27	0.5	0.5	0.1	0.5	0	**115**
28	0.5	0.5	0.1	0.5	10	**225**
29	0.5	0.5	0.5	0.1	0	**181**
30	0.5	0.5	0.5	0.1	10	**329**
31	0.5	0.5	0.5	0.5	0	**189**
32	0.5	0.5	0.5	0.5	10	**273**

Table 2. Simulated results of Q(λ) algorithm for different initial parameters.

Experiment no.	α	γ	ε	λ	Number of obstacles	Episode no. for stable cumulative reward
1	0.1	0.1	0.1	0.1	0	**25**
2	0.1	0.1	0.1	0.1	10	**107**
3	0.1	0.1	0.1	0.5	0	**43**
4	0.1	0.1	0.1	0.5	10	**300**
5	0.1	0.1	0.5	0.1	0	**172**
6	0.1	0.1	0.5	0.1	10	**427**
7	0.1	0.1	0.5	0.5	0	**170**
8	0.1	0.1	0.5	0.5	10	**319**
9	0.1	0.5	0.1	0.1	0	**21**
10	0.1	0.5	0.1	0.1	10	**197**
11	0.1	0.5	0.1	0.5	0	**29**
12	0.1	0.5	0.1	0.5	10	**137**
13	0.1	0.5	0.5	0.1	0	**113**
14	0.1	0.5	0.5	0.1	10	**171**
15	0.1	0.5	0.5	0.5	0	**183**
16	0.1	0.5	0.5	0.5	10	**260**
17	0.5	0.1	0.1	0.1	0	**51**
18	0.5	0.1	0.1	0.1	10	**99**
19	0.5	0.1	0.1	0.5	0	**46**
20	0.5	0.1	0.1	0.5	10	**83**
21	0.5	0.1	0.5	0.1	0	**169**
22	0.5	0.1	0.5	0.1	10	**225**
23	0.5	0.1	0.5	0.5	0	**180**
24	0.5	0.1	0.5	0.5	10	**288**
25	0.5	0.5	0.1	0.1	0	**90**
26	0.5	0.5	0.1	0.1	10	**186**
27	0.5	0.5	0.1	0.5	0	**80**
28	0.5	0.5	0.1	0.5	10	**137**
29	0.5	0.5	0.5	0.1	0	**174**
30	0.5	0.5	0.5	0.1	10	**286**
31	0.5	0.5	0.5	0.5	0	**158**
32	0.5	0.5	0.5	0.5	10	**259**

Besides the initial parameters, the obstacle locations can affect the learning performance. The more obstacles that exist between the start and target points, the more episodes are necessary to find the optimal path. Nevertheless, defining the state set by dynamic variables as explained in Section 3.3.7 enables the system to minimize the effect of obstacle locations. The learning performance of the Sarsa(λ) and Q(λ) methods for different initial Q-value matrices are compared using Table 3, which lists the average number of episodes necessary for reward stabilization with parameters α =0.1, γ =0.7, ε =0.15, and λ =0.8. The resulting

experiment was performed in the environment illustrated in Figure 8. The first column of Table 3 lists the results for the Q-value matrix, consisting of zeros. The second column lists the results for the initial Q-value matrix, learned in a different environment with 10 random obstacles. Thus, Table 3 shows that the system can be executed in an unknown environment no matter where the obstacles are when it learns that the Q-values are good enough.

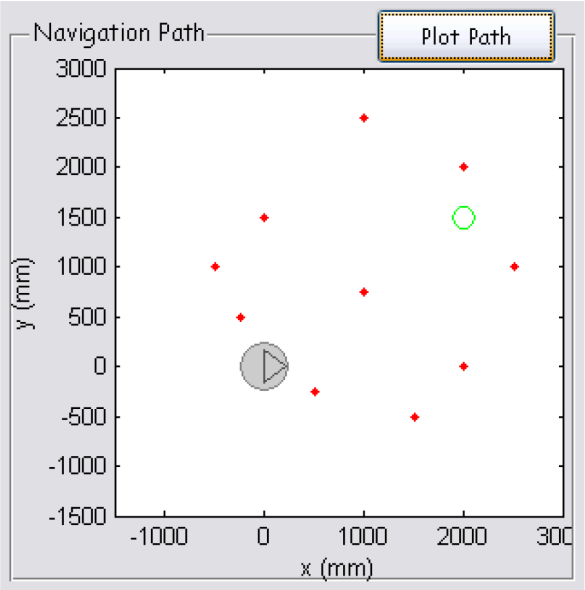


Figure 8. The environment containing 10 obstacles in Tables 1 and 2.

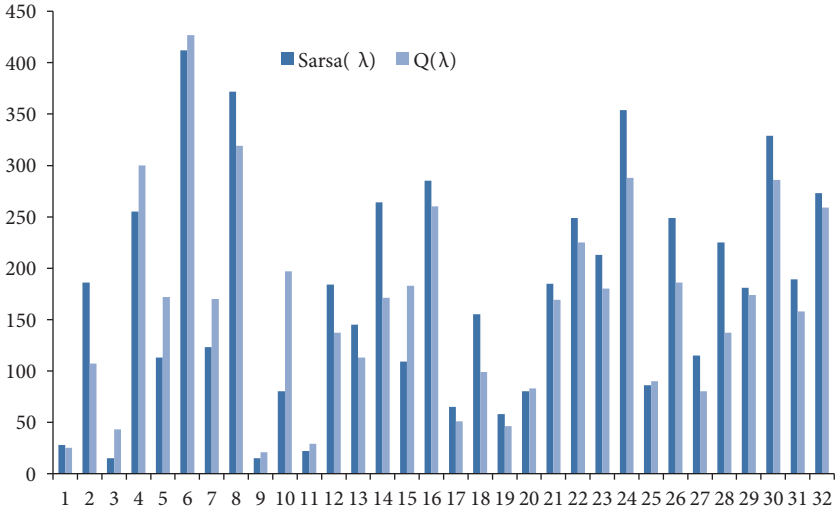


Figure 9. Comparison of Sarsa(λ) and Q(λ) methods depending on Tables 1 and 2. The x-axis shows the number of experiments whose initial parameters are given in Tables 1 and 2. The y-axis maps the number of episodes necessary for the convergence of Q-values.

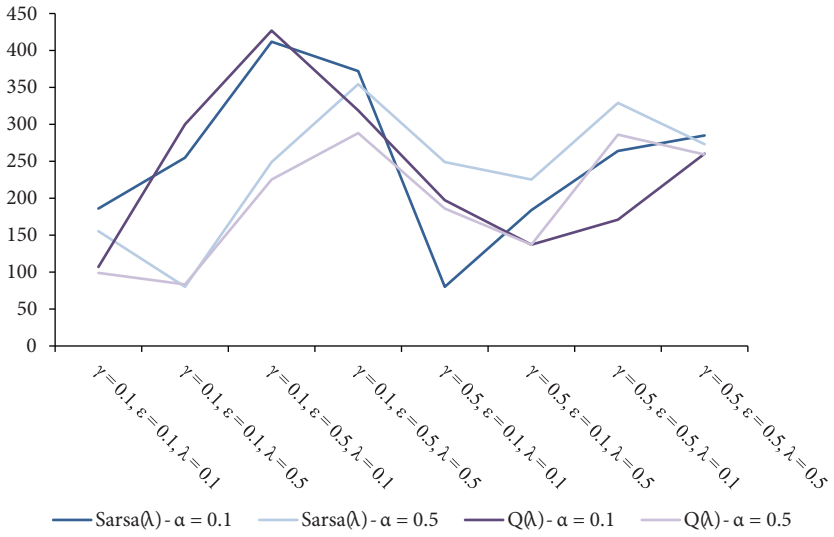


Figure 10. Effects of α values for both Sarsa(λ) and Q(λ), depending on Tables 1 and 2.

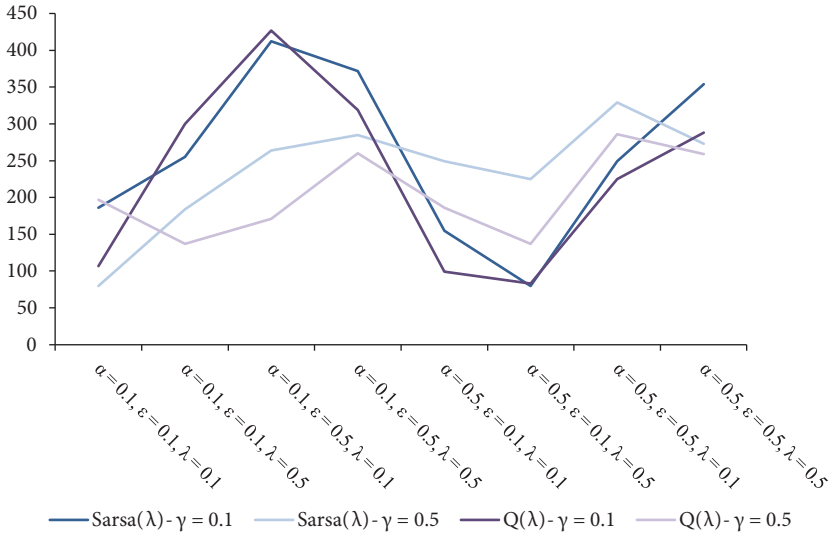


Figure 11. Effects of γ values for both Sarsa(λ) and Q(λ), depending on Tables 1 and 2.

Table 3. The learning performance of Sarsa(λ) and Q(λ) algorithms for different initial Q-value matrices. The results were computed for the environment shown in Figure 8 with initial parameters $\alpha = 0.1$, $\gamma = 0.7$, $\varepsilon = 0.15$, $\lambda = 0.8$.

	Zero matrix as Q-value	Q-value learned in a different environment
Sarsa(λ)	136	100
Q(λ)	107	90

Unfortunately, it is possible to complete an experiment without finding an optimal path. The robot may result in a vicious circle at the end of certain learning processes depending on initial parameter values and the number and locations of the obstacles.

Finally, since the virtual robot cannot capture images of the simulated platform, information about the location of the obstacles in the environment is gathered from the simulator module instead of the sensors.

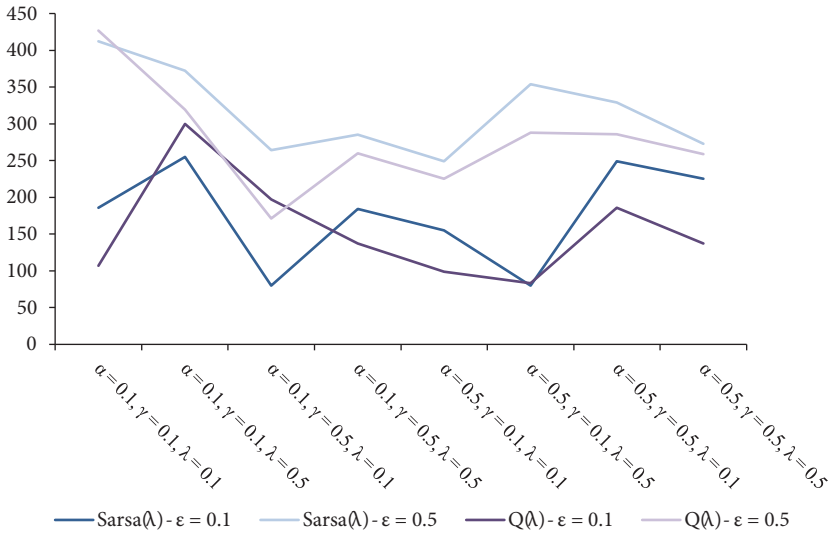


Figure 12. Effects of ε values for both Sarsa(λ) and Q(λ), depending on Tables 1 and 2.

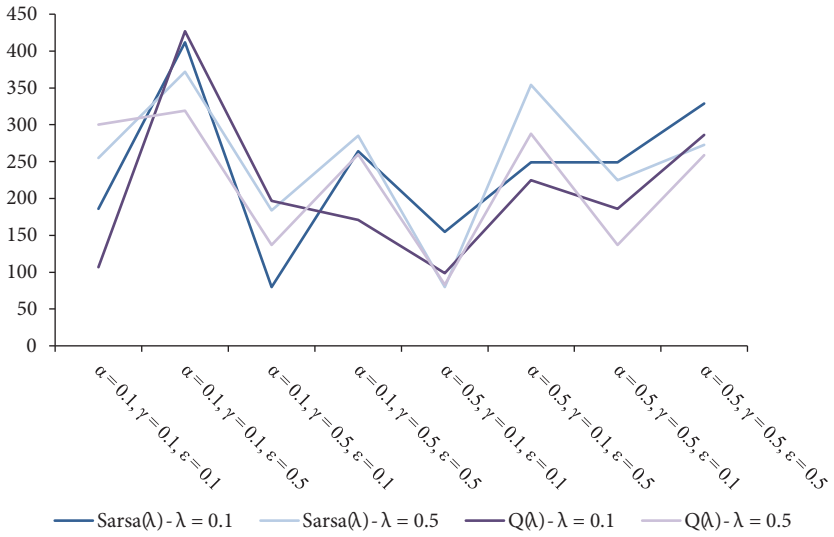


Figure 13. Effects of λ values for both Sarsa(λ) and Q(λ), depending on Tables 1 and 2

4.1. Experiments on Robotino

Studying on a real platform causes certain additional problems that are not encountered on a simulated platform. For instance, it is not a problem for the simulated agent to make decisions in rest as well as between actions selected from a set of motions for specific amounts, but this situation causes the real robot to move jerkily. In order to avoid that jerky transition, the robot is designed to keep its motion of the previously selected action during the decision-making process for the next action.

Another problem of the real experimental platform concerns accelerating the motors of the omnidirectional wheels on the robot to the specified velocity. The PI control is used to solve this problem. When the system orders the robot to move with a certain velocity v, it takes time to accelerate the velocity from 0 to v. This acceleration should be accomplished as fast and smoothly as possible to avoid big delays and robot oscillations. The Robotino uses closed-loop PID control to adjust the speed of motors of its three omnidirectional wheels,

and the EA09 I/O board updates PID control loops. Robotino MATLAB API provides functionality to tune the PID parameters of these motors.

In this study, the built-in controller was used for the Robotino to reach the desired speed. The range of values of PID parameters is from 0 to 255, depending on the user manual. These values are scaled by the microcontroller firmware to match the PID controller implementation. Indeed, there are some methods to tune PID parameters to their optimal values in the closed-loop. In this system, the parameters are set as $K_P = 60$, $K_I = 1.5$, and $K_D = 0$ after a set of trials and observations of motor behaviors. Thus, a PI controller was preferred.

Additionally, the real movement amounts of the robot were different from what we initially expected due to different factors such as communication delays and the position of the three wheels, which were placed at 120° relative to each other. Besides, it is not possible to estimate how much the robot moves during the acceleration process, so it is necessary to obtain information about robot position using the odometer on the robot. The odometer calculates the position of the robot by measuring wheel rotations. However, using an odometer to learn the position of the robot raises certain other questions, namely how accurate the odometer is or whether the velocity of the robot affects odometer accuracy or not.

Certain empirical results of the Robotino with different linear and angular velocities for varied movement amounts were measured. Since the action set of the proposed system consists of motions through the x-axis, θ, and $-\theta$, motion through the y-axis was not measured. For all measurements, the initial position and velocities of Robotino are set as $x = 0$, $y = 0$, $\theta = 0$, $v = 0$, and $w = 0$, and, as mentioned above, PI parameters of the robot motors are set to $K_P = 60$ and $K_I = 1.5$. Finally, the robot keeps capturing 5 image frames per second during the experiments, just to eliminate one of the causes of delays not to be used in the built-in controller.

Movement amount and linear and angular velocities can be changed by the user. Therefore, it is important to be sure that the odometer uncertainty is acceptable for all velocities and movement amounts. Generally, it is decided that the error should be at most 0.02 mm per 1 mm of movement and 0.4° per 1° of turn. After the measurements, the average odometry errors of the Robotino were calculated as 0.011 mm per 1 mm and 0.005° per 1° in linear and rotational motions, respectively. Thus, the odometry accuracy could be accepted as sufficient for use in the system.

After the learning phase of the system is executed, the user can test how well the system learned what to do in both simulated and real platforms. In the test phase, the learned Q-values are used to decide actions. If the Q-value matrix sufficiently converges to optimal, the robot is expected to reach the goal by avoiding obstacles. Figure 14 shows an example execution of the system on the Robotino. The path of the robot is illustrated as a black line.

Figure 5 illustrates the resulting system interface at the termination of the system execution demonstrated in Figure 14. As can be seen in the figures, after a successful learning phase, the system can navigate the robot in the test phase regardless of executing on the simulated or real platform. Even if the environment used in the test phase is different than the one in the learning phase, the robot can reach the target successfully, since states were defined by dynamic variables.

5. Discussion and conclusion

This paper mainly discusses the implementation and performance comparison of two TD(λ)-based RL algorithms: Sarsa(λ), which is an *on*-policy method, and Q(λ), which is *off*-policy. While implementing the proposed system, it is essential to define the state and action sets in order to perform successful learning, since continuous environments have infinite possible states and actions. Thus, discretizing the continuous space de-

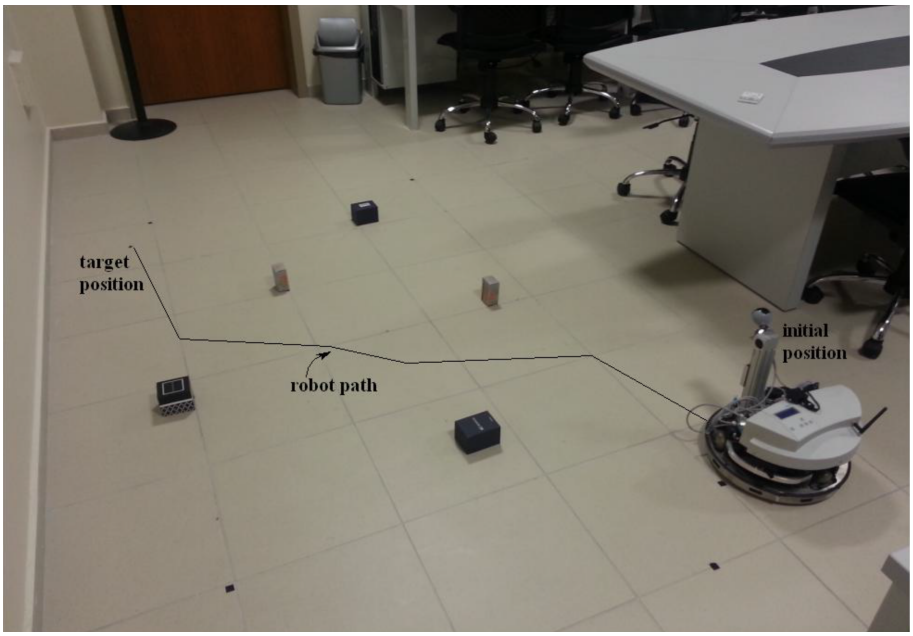


Figure 14. The Robotino can reach the target by avoiding obstacles in the test phase.

termines the performance of the implemented RL algorithms. The proposed system defines a state set using dynamic variables so that after the system learns how to behave in an environment, it can be successful in different environments where the target and obstacles are located in different points. Another decision criterion is how to describe the reward function, which is the response of the environment to the actions of the intelligent agent.

Although the implemented system gives promising results, it can be enhanced to increase its learning speed and performance. For instance, Q-values in the algorithms used by the system are represented in tabular form, which requires large space in the memory and complex mathematical calculations. Instead of tabular form, it is possible to integrate a supervised learning algorithm to represent Q-values in order to reduce memory requirements and provide faster convergence to optimal policy.

Acknowledgment

This study was supported by the Scientific Research Fund of Fatih University (Project No: P50061202_ B).

References

[1] Sebag M. A tour of machine learning: an AI perspective. AI Commun 2014; 27: 11-23.

[2] Bellman RE. A Markov decision process. J Math Mech 1957; 6: 679-684.

[3] Watkins CJCH. Learning from delayed rewards. PhD, Cambridge University, Cambridge, UK, 1989.

[4] Rummery GA, Niranjan M. On-line Q-learning Using Connectionist Systems. Cambridge, UK: Cambridge University Engineering Department, 1994.

[5] Muhammad J, Bucak IO. An improved Q-learning algorithm for an autonomous mobile robot navigation problem. In: IEEE 2013 International Conference on Technological Advances in Electrical, Electronics and Computer Engineering; 9–11 May 2013; Konya, Turkey. New York, NY, USA: IEEE. pp. 239–243.

[6] Yun SC, Parasuraman S, Ganapathy V. Mobile robot navigation: neural Q-learning. Adv Comput Inf 2013; 178: 259-268.

[7] Hwang KS, Lo CY. Policy improvement by a model-free dyna architecture. IEEE T Neural Networ 2013; 24: 776-788.

[8] Fard SMH, Hamzeh A, Hashemi S. Using reinforcement learning to find an optimal set of features. Comput Math Appl 2013; 66: 1892-1904.

[9] Zuo L, Xu X, Liu CM, Huang ZH. A hierarchical reinforcement learning approach for optimal path tracking of wheeled mobile robots. Neural Comput Appl 2013; 23: 1873-1883.

[10] Konar A, Chakraborty IG, Singh SJ, Jain LC, Nagar AK. A deterministic improved Q-learning for path planning of a mobile robot. IEEE T Syst Man Cy A 2013; 43: 1141-1153.

[11] Rolla VG, Curado M. A reinforcement learning-based routing for delay tolerant networks. Eng Appl Artif Intel 2013; 26: 2243-2250.

[12] Geramifard A, Redding J, How JP. Intelligent cooperative control architecture: a framework for performance improvement using safe learning. J Intell Robot Syst 2013; 72: 83-103.

[13] Maravall D, de Lope J, Domínguez R. Coordination of communication in robot teams by reinforcement learning. Robot Auton Syst 2013; 61: 661-666.

[14] Martín JA, de Lope J. $E_x < \alpha >$: an effective algorithm for continuous actions reinforcement learning problems. In: IEEE 2009 35th Annual Conference on Industrial Electronics; 3–5 November 2009; Porto, Portugal. New York, NY, USA: IEEE. pp. 2063-2068.

[15] Khriji L, Touati F, Benhmed K, Al-Yahmedi A. Mobile robot navigation based on Q-learning technique. Int J Adv Robot Syst 2011; 8: 45-51.

[16] McCallum RA. Instance-based state identification for reinforcement learning. Adv Neural In 1995; 8: 377-384.

[17] Zhumatiy V, Gomez F, Hutter M, Schmidhuber J. Metric state space reinforcement learning for a vision-capable mobile robot. In: Arai T, Pfeifer R, Balch T, Yokoi H, editors. Intelligent Autonomous Systems 9. Amsterdam, the Netherlands : IOS Press, 2006. pp. 272-282.

[18] Maček K, Petrović I, Perić N. A reinforcement learning approach to obstacle avoidance of mobile robots. In: IEEE 2002 7th International Workshop on Advanced Motion Control; 3–5 June 2002; Maribor, Slovenia. New York, NY, USA: IEEE. pp. 462-466.

[19] Altuntaş N, İmal E, Emanet N, Öztürk CN. Reinforcement learning based mobile robot navigation. In: ISCSE 2013 3rd International Symposium on Computing in Science and Engineering; 24–25 October 2013; Kuşadası, Turkey. İzmir, Turkey: Gediz University Publications. pp. 285-289.

[20] Sutton RS, Barto AG. Reinforcement Learning: an Introduction. Cambridge, MA, USA: MIT Press, 2005.

[21] Russell SJ, Norvig T. Artificial Intelligence: A Modern Approach. 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[22] Xu X, Lian CQ, Zuo L, He HB. Kernel-based approximate dynamic programming for real-time online learning control: an experimental study. IEEE T Contr Syst T 2014; 22: 146-156.

[23] Ni Z, He HB, Wen JY, Xu X. Goal representation heuristic dynamic programming on maze navigation. IEEE T Neural Networ 2013; 24: 2038-2050.

[24] Hwang KS, Jiang WC, Chen YJ. Adaptive model learning based on dyna-Q learning. Cybernet Syst 2013; 44: 641-662.

[25] Bellman RE. Dynamic Programming. Princeton, NJ, USA: Princeton University Press, 1957.

[26] Wang YH, Li THS, Lin CJ. Backward Q-learning: the combination of Sarsa algorithm and Q-learning. Eng Appl Artif Intel 2013; 26: 2184-2193.

[27] Chen XG, Gao Y, Fan SG. Temporal difference learning with piecewise linear basis. Chinese J Electron 2014; 23: 49-54.

[28] Chen XG, Gao Y, Wang, RL. Online selective kernel-based temporal difference learning. IEEE T Neural Networ 2013; 24: 1944-1956.

[29] Kober J, Bagnell JA, Peters J. Reinforcement learning in robotics: a survey. Int J Robot Res 2013; 32: 1238-1274.

[30] Lopez-Guede JM, Fernandez-Gauna, B, Gra na M. State-action value function modeled by ELM in reinforcement learning for hose control problems. Int J Uncertain Fuzz 2013; 21: 99-116.

[31] Miljković Z, Mitić M, Lazarević M, Babić B. Neural network reinforcement learning for visual control of robot manipulators. Expert Syst Appl 2013; 40: 1721-1736.