TRANSNAV the International Journal on Marine Navigation and Safety of Sea Transportation
http://www.transnav.eu
Volume 19
Number 3
September 2025
DOI: 10.12716/1001.19.03.20

Optimizing the Path of a Mobile Agent in the Environment with Static Obstacles using Reinforcement Learning

A. Sawicki & M. Tomera
Gdynia Maritime University, Gdynia, Poland

ABSTRACT: Path planning for a mobile agent concerns the problem of searching for a collision-free and optimal path between the initial and target positions. The space in which the agent moves contains a number of obstacles modeled by ordered grids, each representing the obstacle location in the space of the agent movement. The final boundary of an obstacles is formed by its actual boundary plus a minimum safe distance, taking into account the size of the agent, which allows to treating the obstacles as points in the environment. In the article, reinforcement learning algorithms were used to determine the path, including numerous design methods: dynamic programming (policy iteration algorithm and value iteration algorithm), Monte Carlo method (Monte Carlo control algorithm), temporal-difference (TD) learning (Q-learning algorithm and Sarsa algorithm), eligibility traces (Q(◎)) algorithm and Sarsa(◎) algorithm), planning and learning (Dyna-Q algorithm), and gradient methods (Q-learning algorithm with Adam optimizer and Sarsa algorithm with Adam optimizer). The reinforcement learning algorithms operate on the principle of determining the agent's policy, which seeks the minimum distance between the initial and target positions of the mobile agent, while avoiding obstacles. These learning procedures differ between, dynamic programming, which requires a good knowledge of the environment model to determine the agent's policy, and other methods which do not require this knowledge. The aim of the reported work was to examine the above named algorithms in terms of their effectiveness and speed of finding the optimal solution. Based on the results of the simulation studies, the most effective methods turned out to be that using gradient methods for optimization, i.e.: Q-learning with Adam optimizer.

1 INTRODUCTION

Path planning is one of the most fundamental issues in autonomous control of robots, cars, ships, and aircrafts. In these control systems, objects need to be moved along the shortest possible path while avoiding obstacles. The basis of path planning is the development of appropriate algorithms, which have evolved from the initial traditional graph-search based methods to modern methods based on optimization. Path planning can be divided into global planning intended to determine optimal routes when a map is available, and local path planning focusing on dealing with obstacles not represented on the map by using localized obstacle avoidance techniques [14]. Depending on whether the obstacles are stationary or moving, local path planning can be further divided into static or dynamic [35].

The algorithms used in global path planning can be further divided into three main groups. The first group consists of traditional algorithms (Dijkstra, A*), while the second group includes optimization algorithms based on artificial intelligence (GA-genetic algorithm, PSO-particle swarm optimization algorithm, etc.), and

the third group consists of algorithms based on RRT (rapidly exploring random tree) path planning.

In the first group, the oldest algorithm is Dijkstra's algorithm, proposed in 1959, which is based on graph search [7]. The algorithm builds a graph model and then traverses most of the subnodes, one by one according to the greedy principle, using the relaxation method to optimize the path selection. The Dijkstra's algorithm gives a good planning result when the volume of map data is small. When it is large, the results become poorer and can even fail to meet the requirements of path planning [31]. A-star (A*) algorithm is a traditional algorithm obtained by improving Dijkstra's algorithm by adding, a heuristic function. This algorithm is based on the use of a raster grid, to discretize the solution search space [3, 26], thus making the basis for drawing graphs of connections between the points. The A-star algorithm traverses the graph and finds paths for a given weighted graph, source node, and destination node [16, 32].

The second group includes artificial intelligence-based optimization algorithms applicable in global route planning. These algorithms determine the optimal traversal path by establishing mathematical models, such as genetic algorithms (GA) [15], differential evolution (DE) [6], ant colony optimization (ACO) [24], firefly algorithm (FA) [1], particle swarm optimization (PSO) [36], whale optimization algorithm (WOA) [29], flower pollination algorithm (FPA) [4], grey wolf optimizer (GWO) [18], sparrow search algorithm (SSA) [34], and cuckoo search algorithm (CSA) [20]. Swarm intelligence algorithms are widely used in optimization computations due to their strong optimization ability and ease of implementation.

The third global path planning group includes the algorithm based on stochastic sampling with rapidly-exploring random tree (RRT) path planning [21]. This algorithm is designed to efficiently search nonconvex and, high-dimensional spaces by randomly building a space-filling tree. The tree is constructed incrementally from samples randomly taken from the search space and is inherently biased to grow toward large, unsearched problem regions. The RRT algorithm was developed by Steven LaValle [13]

The algorithms used in local route planning can be divided into four main groups. The first group consists of algorithms based on Dubins curves, while the second group includes the dynamic window approach (DWA) [15, 24], the third group consists of algorithms based on the artificial potential field (APF) method, and the fourth group consists of reinforcement learning (RL).

In the first group, there are path planning algorithms based on mathematical rules, such as those using Dubins curves [9]. These algorithms make it possible to obtain paths with bounded curvature and derivative of curvature which connect two assigned positions (positions and orientations). The solution is based on the Dubins shortest path approximation and can be used to connect an ordered sequence of points located between the initial and target points [22, 30].

The second group includes the dynamic window approach (DWA). This approach can make the agent avoid unknown obstacles in time and improve the smoothness of the transition path in the moving process. However, it has some drawbacks, such as a single movement state, a long path, and the ease of falling into a local minimum. Therefore, this method is often used in combination with the A* algorithm [17] or the genetic algorithm (GA) [15] or the ant colony optimization (ACO) algorithm [24].

The third group of local path planning includes the artificial potential field (APF) method, which has a simple algorithm, few iterations, high real-time performance and smooth path, but it can easily fall into a local minimum and performs poorly in scenarios with many obstacles [19].

The fourth group includes reinforcement learning (RL) [5], being the object of study in this article. Combining reinforcement learning with artificial neural networks (ANN) creates deep reinforcement learning (DRL) which shows promise in path planning [25]. Reinforcement learning is the algorithm that aims to find the optimal strategy for an agent interacting with an unknown environment. For the actions performed, the agent receives positive, negative or zero reward signals, and the calculated strategy should maximize the cumulative rewards. The agent explores the environment to quickly learn about actions with significant rewards. This definition covers a wide range of problems that can be solved, including path planning [11].

2 MATHEMATICAL MODELING OF REINFORCEMENT LEARNING

Reinforcement learning (RL) is a part of machine learning (ML), being, in turn, a subset of artificial intelligence (AI). The basic components of reinforcement learning are a computational agent and an environment, as shown in Figure 1. Based on the observation of the state st of the environment, the agent takes the action at. The environment is an entity with which the agent can interact through a predefined set of actions $A = \{a_1, a_2, \dots\}$. The set A describes all possible actions, while the set S describes all possible states of the environment $S = \{s_1, s_2, \dots\}$ [28].

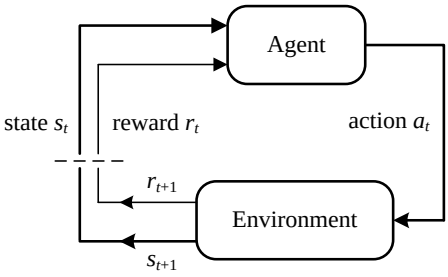

Figure 1. Interaction between agent and environment.

The agent learns by exploring various actions in the environment, receiving feedback in the form of rewards, and adjusting its behaviour to maximize the expected rewards. At any time step t, the agent first observes the current state s_t of the environment and its corresponding reward value r_t. Then, it decides what to do next, based on the information on the state and the reward. The action a_t that the agent intends to perform is introduced into the environment, as a result of which the agent can obtain a new state s_{t+1} and a new reward r_{t+1}. In this way, the agent creates a sequence or

trajectory that starts as follows: s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, … (*State, Action, Reward*, etc.).

The basic task in reinforcement learning is to determine a policy for the agent, denoted as $\pi(a_t|s_t)$, which defines the agent's decision-making strategy. It maps states to actions, indicating what action the agent should take in each state. The goal of the reinforcement learning is to find parameters that define the policy $\pi(a|s)$ and maximize the value of the sum of rewards on the trajectory.

The sequential decision making by the agent is modeled as a Markov decision process (MDP).

2.1 *Markov Decision Process*

Markov decision processes (MDP) allow modeling stochastic environments in which the future state depends solely on the current state and the action taken, which is known as the Markov property. This means that subsequent states in the future do not depend on the past but only on the current value of the state, and, consequently, when the agent makes a decision, it is not necessary to analyze its past history on the trajectory.

The Markov decision process (MDP) provides a formal way to represent the interaction of an agent with its environment, in which the agent takes actions in different states to maximize cumulative rewards. The MDP is described by five variables (S, A, p, R, γ), where [2]:

- S is the set of states (discrete or continuous);
- A is the set of actions (discrete or continuous);
- $p(s_{t+1} | s_t, a_t)$ is the transition function that determines the probability of transitioning to state s_{t+1} after performing action at in state s_t;
- $R(s_{t+1}|s_t, a_t)$ is the reward function assigning a value (reward or penalty) for transitioning from state s_t to s_{t+1} as a result of the action at;
- $\gamma \in [0,1]$ is the discount rate that determines the importance of future rewards relative to the current ones.

Figure 2 shows a graphical model of the Markov decision process, where an edge denotes the transition probability between states. The graphical model contains both rewards, states, and actions, and the transition probabilities are conditioned by both states and actions, hence $p(s_{t+1}|s_t, a_t)$.

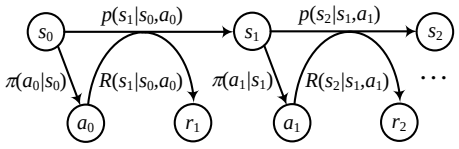

Figure 2. Graphical model of the Markov decision process [10].

2.2 *Expected Return*

Through interactions with the environment, the agent learns from the outcomes and adjusts its policy π to improve its decision-making over time. According to this approach, the agent tries to choose actions such that the sum of discounted rewards it receives in the future is maximized. In particular, it chooses an action that maximizes the expected return G_t, which is the sum of discounted rewards on the trajectory:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-t-1} r_T = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k \tag{1}$$

where T is the final time step, γ is the parameter $0 \le \gamma \le 1$, referred to as the discount rate and used to reduce the weights as the time step t increases, and r_t is the immediate reward received at time t. The value of the reward r depends on the state and the action, or sometimes only on the state, and determines which states and actions are better. The final time step T occurs when the agent's interaction with the environment is decomposed into finite-length sequences, called episodes. Each episode ends with a special state, called the terminate state, after which the return to the standard initial state takes place. Formula (1) also applies to the cases in which $T = 0$ and $\gamma = 0$.

2.3 *Value Functions*

The agent's goal is to find a policy $\pi(a|s) \in \Pi$, that optimizes the state-value function or the action-value function. The state-value function allows the agent to assess how good the state is, in which the agent currently is, while the action-value function allows to assess how good the performance of a given action is in a given state.

The state-value function $\upsilon_\pi(s)$ is the expected return when the agent applies policy π in the given state s

$$\upsilon_\pi(s) = E_\pi[G_t \mid s_t = s] = E_\pi\left[\sum_{k=t+1}^{T} \gamma^{k-t-1} r_k \mid s_t = s\right] \tag{2}$$

where $E_\pi\{\cdot]$ denotes the expected value. The function υ_π bears the name of the state-value function for policy π.

A fundamental property of the value functions used throughout reinforcement learning and dynamic programming is that they satisfy the recurrence relations similar to those described for the expected return given by (2). For any policy π and any state s, the following consistency condition holds between the value of state s and the value of the possible successor states following it [28]:

$$\begin{aligned}\upsilon_\pi(s) &= E_\pi[G_t \mid s_t = s] = E_\pi[r_{t+1} + \gamma G_{t+1} \mid s_t = s] \\ &= \sum_a \pi(s,a) \sum_{s',s} p(s', r \mid s, a)[r + \gamma \upsilon_\pi(s')]\end{aligned} \tag{3}$$

where s' is the next state s_{t+1} and r is the immediate reward of going to the state s_{t+1}. Equation (3) is the Bellman equation for $\upsilon_\pi(s)$. The value of the function $\upsilon_\pi(s)$ is the unique solution of the Bellman equation. From the definition of the state function υ_π its optimal value can be easily determined as

$$\upsilon^*(s) = \max_{\pi \in \Pi} \upsilon_\pi(s) \tag{4}$$

Similarly, the action-value function $q_\pi(s,a)$ is the expected return starting from state s, after taking action a, and then applying policy π

$$q_\pi(s,a) = E_\pi[G_t \mid s_t = s, a_t = a] = E_\pi\left[\sum_{k=t+1}^{T} \gamma^{k-t-1} r_k \mid s_t = s, a_t = a\right] \quad (5)$$

The function q_π bears the name the action-value function for policy π. Similarly to the state-value function υ_π, the optimal value of the action-value function q_π can be defined as

$$q^*(s,a) = \max_{\pi \in \Pi} q_\pi(s,a) = E_\pi\left[r_{t+1} + \gamma \upsilon^*(s_{t+1}) \mid s_t = s, a_t = a\right] \quad (6)$$

Compared to the state-value function υ_π a special feature of the action-value function q_π is that the optimal policy π can be obtained directly from q^*(s,a)

$$\pi^*(s) = \arg\max_{a \in A} q^*(s,a) \quad (7)$$

The optimal state-value function $\upsilon^*(s)$ denotes the expected discounted return when the agent is in a given state s and then applies the policy π^*. The optimal value function $q^*(s,a)$ denotes the expected discounted return when the agent is in a given state s and for a given action a in that state, it applies the policy π^* [10].

2.4 *Bellman's Optimality Equations*

Because υ^* is the state-value function for policy π^*, it must satisfy the self-consistency condition described by Bellman's equation for state values (3). Because it is the optimal state-value function, however, the consistency condition for υ^* can be written in a special form without reference to any particular policy. This is the Bellman optimality equation for υ^*. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state [28]:

$$\upsilon^*(s) = \max_{a \in A} q_\pi^*(s,a) = \max_a E_\pi\left[r_{t+1} + \gamma \upsilon^*(s_{t+1}) \mid s_t = s, a_t = a\right] = \max_a \sum_a \pi(s,a) \sum_{s',r'} p(s', r \mid s, a)[r + \gamma \upsilon_\pi(s')] \quad (8)$$

Equation (8) is the Bellman optimality equation for υ^*. The corresponding Bellman optimality equation for action-value function q^* has the form:

$$q^*(s,a) = E_\pi\left[r_{t+1} + \gamma \max_{a'} q^*(s_{t+1}, a') \mid s_t = s, a_t = a\right] = \sum_{s',r'} p(s', r \mid s, a)[r + \gamma \max_{a'} q^*(s', a')] \quad (9)$$

For Markov decision processes with a finite number of states, the Bellman optimality equation (8) for υ^* has a single solution. This general equation is in fact a system of equations, one for each state, so if there are n states, then there are n equations and n~unknowns. If the dynamics p of the environment is known, then we can solve this system of equations for υ^* using an arbitrary methods for solving systems of nonlinear equations. We can also solve a related set of equations for q^*. However, in this case, problems may arise for very large numbers of states, since the computational complexity of the solution increases by order n^3, where n is the number of states [28]. In addition, the tasks solved in reinforcement learning rely on the interaction of the agent with the environment, of unknown dynamics. Hence, direct methods to solve the system of nonlinear equations cannot be succesfully applied to the Bellman optimality equations (8) and (9). Fortunately, in practice, there are iterative methods to solve problems modeled as Markov decision processes, including dynamic programming, Monte Carlo estimation, and temporal- difference learning, which also allow solving the Bellman optimality equations [8].

3 ALGORITHMS FOR DETERMINING THE OPTIMAL AGENT POLICY

Depending on the level of knowledge of the environment, different methods are used to determine the agent policy π. In the case of full knowledge of the environment model, dynamic programming methods can be used. Otherwise, if the environment model is not known, then such methods as Monte Carlo or temporal-differences learning are available.

3.1 *Dynamic Programming*

Dynamic programming (DP) refers to a set of algorithms that can be used to compute optimal agent policies assuming that an accurate model of the environment, described as a Markov decision process (MDP) with finite sets of states S, actions A, and rewards R is given. The key idea of dynamic programming, as well as reinforcement learning in general, is to use the value functions (υ_π, q_π) to search for good policies π. The updates of the value function are closely related to the Bellman equations, transformed into assignment instructions. Classical methods used in dynamic programming iteratively update the value of one state based on the values of all possible states following it and the probabilities of their occurrence. When the updates no longer cause any changes in the values, the convergence occurs to values that satisfy the Bellman equation.

Policy evaluation refers to iterative computation of a value function for a given policy. Policy improvement is accomplished by computing an improved policy with the given value function for that policy. Combining these two computations yields the policy iteration and the value iteration, the two most popular DP methods. Each of them can be used to reliably compute optimal policies and value functions for finite MDPs, assuming full knowledge of the MDP.

3.1.1 *Policy Iteration Algorithm*

The policy iteration consists of finding the optimal policy via a sequence of monotonically improved policies π_k, based on the evaluation of the value function $\upsilon_{\pi k-1}$ for the previous policy π_{k-1}. Policy iteration consists of two simultaneous, and interacting processes, one that makes the value function consistent with the current policy (policy evaluation), and the other that makes the policy greedy with respect to the current value function (policy improvement). In policy

iteration, these two processes alternate, each ending before the other begins, but this is not necessary. For example, in value iteration, only one policy evaluation iteration is performed between each policy improvement. Algorithm 1 contains the pseudocode for implementing the policy iteration.

Algorithm 1.
Policy Iteration algorithm for estimating $\pi \approx \pi^*$
Setting parameter values: $\gamma = 0.99$ and $\theta = 10^{-12}$.
1. Initialize:
  V(s) $\leftarrow$ 0, for all $s \in S$
  π(s) $\leftarrow$ 0, for all $s \in S$
2. Policy Evaluation
  repeat
    $\Delta \leftarrow 0$
    for each state $s \in$ S:
      $\upsilon \leftarrow$ V(s)
      V(s) $\leftarrow \sum_{s',r} p(s',r \mid s,\pi(s))[r + \gamma V(s')]$
      $\Delta \leftarrow \max(\Delta, \mid \upsilon$ - V(s)$\mid)$
    until $\Delta < \theta$
3. Policy Improvement
  policy-stable $\leftarrow$ *True*
  for each state $s \in$ S:
    old-action $\leftarrow \pi$(s)
    π(s) $\leftarrow \arg\max_a \sum_{s',r} p(s',r \mid s,\pi(s))[r + \gamma V(s')]$
  if *old-action* $\neq \pi$(s) **then**
    policy-stable $\leftarrow$ *False*
  end
end
if *policy-stable* **then**
  stop
  return V $\approx \upsilon$, $\pi \approx \pi'$
else
  go to 2
end

3.1.2 *Value Iteration Algorithm*

The value iteration is obtained by transforming the Bellman optimality equation (8) into an update rule for each state. Each such operation updates the value of one state based on the values of all possible successor states and their probability of occurrence. When the updates do not cause any further changes in the values, the convergence occurs to the values that satisfy the Bellman equation. Based on the state-value function from the Bellman optimality equation (9), the optimal policy is determined. Algorithm 2 shows the subsequent steps performed during the value iteration.

Algorithm 2.
Value Iteration algorithm for estimating $\pi \approx \pi^*$
Setting parameter values: $\gamma = 0.99$ and $\theta = 10^{-12}$.
1. Initialize:
  V(s) $\leftarrow$ 0, for all $s \in$ S
  ◎(s) $\leftarrow$ 0, for all $s \in$ S
2. Value Iteration
  repeat
    $\Delta \leftarrow 0$
    for each state $s \in$ S:
    q(a) $\leftarrow$ 0 (for all $a \in$ A)
    for each action $a \in$ A:
      s′,r $\leftarrow$ p(s′,r$\mid$s,a)
      $\delta \leftarrow$ r + γV(s′)
      q(a) $\leftarrow$ *q(a)* + δ
    end
    $q_{max} \leftarrow \max$ *q(a)*
    $\Delta \leftarrow \max(\Delta, \mid q_{max}$ - V(s)$\mid)$
    V(s) $\leftarrow q_{max}$
  end
until $\Delta < \theta$
3. Policy Calculation
  for each state $s \in$ S:
  q(a) $\leftarrow$ 0, (for all $a \in$ A)
  for each action $a \in$ A:
    s′,r $\leftarrow p(s',r \mid s,a)$
    $\delta \leftarrow r + \gamma V(s')$
    q(a) $\leftarrow$ *q(a)* + δ
  end
  π(s) $\leftarrow$ argmax$_a$*q(a)*
end

3.2 *Monte Carlo methods*

Monte Carlo methods are used to find optimal policies when only sample episodes are given and no other knowledge about the dynamics of the environment. However, they require a great many episodes to be able to determine the optimal policy, making them of little practical use for solving any but the smallest problems.

3.2.1 *Monte Carlo Control Algorithm*

This paper presents the Monte Carlo control method written as a pseudocode in Algorithm 3. This is a method based on the ε-soft-greedy policy, which means that $\pi(a \mid s) > 0$ for all $s \in S$, but gradually moves closer and closer to the deterministic optimal policy. All non-greedy actions are determined by formula (10), for all a $\in$ A.

$$\pi(a_t \mid s_t) = \max_a q((a, s_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon / \mid A \mid & \text{if } a = A^* \\ \varepsilon / \mid A \mid & \text{if } a \neq A^* \end{cases} \quad (10)$$

where $A^* = \max_a Q(s_t,a)$

The greedy strategy refers to a strategy in which the agent always chooses the action with the highest value of the value function. The selected action refers to the greedy action and the agent is said to exploit the environment. The ε-greedy strategy refers to a strategy in which the agent chooses the greedy action with probability 1-ε and a random action with a small probability ε. The random action refers to the non-greedy action and the agent is said to explore the environment.

Algorithm 3.
Monte Carlo Control algorithm
(first visit for ε-soft policies)
Setting parameter value: $\gamma = 0.99$.
Algorithm parameter: small $\varepsilon > 0$
Initialize:
  π(s) $\leftarrow$ an arbitrary ε-soft policy
  Q(s,a) $\in$ R (arbitrarily), for all $s \in$ S, $a \in$ A
  Returns(s,a) $\leftarrow$ empty list, for all $s \in$ S, $a \in$ A
repeat (for each episode):
  repeat (for each step *t* of episode):
    $a_t \leftarrow$ ε- soft-greedy}(s_t,*Q*)
    Generate an episode: $s_0,a_0, r_1,\ldots, s_{T-1}, a_{T-1}, r_T$
  until *s* is terminal
  G $\leftarrow$ 0
  for each step of episode, *t=T-1, T-2,…,0*:
    G $\leftarrow \gamma$G + r_{t+1}
    if first visit the pair s_t, a_t in s_0, a_0, s_1, a_1, s_{t-1}, a_{t-1}
      Append *G* to *Returns*(s_t,a_t)
      Q(s_t,a_t) $\leftarrow$ average(*Returns*(s_t,a_t))
      A* $\leftarrow$ argmax$_a$ *Q*(s_t,*a*)
    end
  end
until stop

3.3 *Temporal-Difference Learning*

Temporal-difference (TD) learning is a combination of the ideas of Monte Carlo methods and dynamic programming (DP). Like Monte Carlo methods, TD methods can learn directly from experience without engaging a model of environment dynamics. Like DP, TD methods update the estimates based in part on other learned estimates, without waiting for the final result. The most popular algorithms that use the idea of learning time differences are Q-learning and Sarsa. Both of these algorithms learn the action-value function.

3.3.1 *Q-learning Algorithm*

Q-learning learns the action-value function $Q(s,a)$, which is a prediction of the return associated with each action $a \in A$, in each state $s \in S$. The temporal-difference (TD) between the target value and the estimated value at different time steps is used to update the value function Q [27]. The most basic method of calculating the temporal-difference TD involves updating the state-value function $V(s)$ as follows:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)] \quad (11)$$

where α is the learning rate, and γ is the discount factor. At each state, the value function V corresponds to the action-value function Q for the action with the highest predicted return, which can be written as

$$V(s_t) \leftarrow Q(s_t, a_t) \quad (12)$$

Since the primary goal is to maximize the obtained return $Q(s_t,a_t)$, the function estimating the value of the state $V(s_{t+1})$ can be written as

$$V(s_{t+1}) \leftarrow \max_{a \in A} Q(s_{t+1}, a) \quad (13)$$

After substituting the dependencies (12) and (13) into formula (11), the single-step update equation in the Q-learning method is obtained

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (14)$$

As shown in [33], this update converges to some fixed value for the Markov decision problems with a finite number of states, for which the function Q is stored in an array. When the function Q converges, then the optimal policy is the action in each state with the highest predicted return.

Algorithm 4 contains a pseudocode based on equation (14), which allows to determine the route for the mobile agent using the Q-learning method.

Algorithm 4. Q-learning algorithm
Setting parameter values: γ= 0.99 and α = 0.01.
Initialize: $Q(s,a) \leftarrow 0$, for all $s \in S$, $a \in A$
repeat (for each episode):
  Initialize state s_0
  repeat (for each step t of episode):
    $a_t \leftarrow \varepsilon$- greedy($s_t$,Q)
    r_t, $s_{t+1} \leftarrow Env(s_t,a_t)$
    $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\ [r_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)]$
    $s_t \leftarrow s_{t+1}$
  until s is terminal
until stop

3.3.2 *Sarsa Algorithm*

This method was proposed by Rummery and Niranjan [23] under the name of Modified Connectionist Q-Learning (MCQ-L). The currently used name for this method, Sarsa, proposed by Richard Sutton [28], is derived from the first letters of the words (state-action-reward-state-action) and is used to determine the table Q containing the probability of taking a specific action a in state s. The update rule in this method is formulated as follows:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)] \quad (15)$$

Algorithm 5 presents the Sarsa algorithm steps to update the Q table based on equation (15).

Algorithm 5. Sarsa algorithm
Setting parameter values: γ= 0.99 and α = 0.01.
Initialize: $Q(s,a) \leftarrow 0$, for all $s \in S$, $a \in A$
repeat (for each episode):
  Initialize state s_0
  $a_0 \leftarrow \max_a Q(s_0, a)$
  repeat (for each step t of episode):
    $a_t \leftarrow \varepsilon$- greedy($s_t$,Q)
    r_t, $s_{t+1} \leftarrow Env(s_t,a_t)$
    $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\ [r_{t+1} + \gamma\ Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]$
    $s_t \leftarrow s_{t+1}$
    $a_t \leftarrow a_{t+1}$
  until s is terminal
until stop

3.4 *Eligibility Traces*

Eligibility traces unify and generalize temporal-difference (TD) learning and Monte Carlo methods. When TD methods are supplemented with eligibility traces, they form a family of methods spanning a spectrum that has Monte Carlo methods at one end (λ=1) and single-step TD methods at the other end (λ=0). In between, there are intermediate methods that are often better than either of the extreme methods. An eligibility trace is a temporary record that maintains a trail of state-action pairs taken over time.

When the eligibility traces are extended with a Q-learning, the obtained combination is known as the Watkins Q(λ) algorithm [also simply refered to as Q(λ) algorithm]. It is similar to the Q-learning algorithm, except that it is augmented with eligibility traces. The eligibility traces are updated in two steps. First, if a non-greedy action is taken, the eligibility traces are set to zero for all state-action pairs. Otherwise, they are decomposed by $\gamma\lambda$. When the eligibility traces are extended with the Sarsa algorithm, the obtained combination is known as the Sarsa(λ) algorithm. Here, the basic algorithm is similar to the Sarsa algorithm, except that the backups are performed n steps later rather than one step later. The algorithms Q(λ) and Sarsa(λ), from [28], are presented here as Algorithm 6 and Algorithm 7, respectively.

Algorithm 6. Q(λ) algorithm
Setting parameter values:γ= 0.99 and α = 0.01.
Initialize:
  $Q(s,a) \leftarrow 0$, for all $s \in S$, $a \in A$
  $E(s,a) \leftarrow 0$, for all $s \in S$, $a \in A$
repeat (for each episode):
  Initialize state s_0
  Initialize action a_0
  repeat (for each step t of episode):
    r_t, $s_{t+1} \leftarrow$ Env(s_t,a_t)

$a_{t+1} \leftarrow$ ε- greedy(s_{t+1},Q)
$a^* \leftarrow \text{argmax}_a Q(s_{t+1},a)$
$\delta \leftarrow r_t+\gamma Q(s_{t+1}, a^*) - Q(s_t,a_t)]$
$E(s_t,a_t) \leftarrow 1$
for all $s \in S$
for all $a \in A$
$Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
if $a_{t+1} == a^*$ **then**
$E(s,a) \leftarrow \gamma\lambda E(s,a)$
else
$E(s,a) \leftarrow 0$
end
end
end
$s_t \leftarrow s_{t+1}$
$a_t \leftarrow a_{t+1}$
until s is terminal
until stop

Algorithm 7. Sarsa(λ) algorithm
Setting parameter values:γ= 0.99 and α = 0.01.
Initialize:
$Q(s,a) \leftarrow 0$, for all $s \in S$, $a \in A$
$E(s,a) \leftarrow 0$, for all $s \in S$, $a \in A$
repeat (for each episode):
Initialize state s_0
Initialize action a_0
repeat (for each step t of episode):
r_t, $s_{t+1} \leftarrow Env(s_t,a_t)$
$a_{t+1} \leftarrow$ ε- greedy(s_{t+1},Q)
$\delta \leftarrow r_t+\gamma Q(s_{t+1}, a_{t+1}) - Q(s_t,a_t)]$
$E(s_t,a_t) \leftarrow 1$
for all $s \in S$
for all $a \in A$
$Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
$E(s,a) \leftarrow \gamma\lambda E(s,a)$
end
end
$s_t \leftarrow s_{t+1}$
$a_t \leftarrow a_{t+1}$
until s is terminal
until stop

3.5 *Planning and Learning*

There are methods that learn a model from experience. Then, the obtained model is used to plan the agent's policy. A good example of this method is the Dyna-Q algorithm, which has a simple architecture that integrates the main functions needed for learning and planning the agent's actions.

3.5.1 *Dyna-Q Algorithm*

Dyna-Q is a combination of planning and temporal-difference (TD) learning. Planning is a process that takes a model of environment as input and creates a policy using simulated experience generated randomly, while direct TD uses real experience obtained from environmental interactions to refine the action-value function Q. The pseudocode for Dyna-Q is shown as Algorithm 8 [28], where *Model-transitions(s,a)* contains the next predicted state, and *Model-rewards(s,a)* contains the received reward for the state-action pair (s, a), with N being the number of planning steps. If the planning step $N = 0$, then the only active algorithm remains the Q-learning algorithm.

Algorithm 8. Dyna-Q algorithm
Setting parameter values:γ= 0.99 and α = 0.01.
Initialize:
$Q(s,a) \leftarrow 0$, for all $s \in S$, $a \in A$
Model-transitions(s,a) ← 0, for all $s \in S$, $a \in A$
Model-rewards (s,a) ← 0, for all $s \in S$, $a \in A$
$N \leftarrow$ constant
repeat (for each episode):
Initialize state s_0
repeat (for each step t of episode):
$a_t \leftarrow$ ε- greedy(s_t,Q)
r_t, $s_{t+1} \leftarrow Env(s_t,a_t)$
$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_t+\gamma \text{max}_a Q(s_{t+1},a) – Q(s_t,a_t)]$
Model-transitions(s_t,a_t) ← s_{t+1}
Model-rewards(s_t,a_t) ← r_t
k ← 0
while k < N **do**
$s_k \leftarrow$ random previously observed state from *Model-transitions(s,a)*
$a_k \leftarrow$ random action previously taken from *Model-transitions(s,a)*
$s_{k+1} \leftarrow$ *Model-transitions*(s_k,a_k)
$r_k \leftarrow$ *Model-rewards*(s_k,a_k)
$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_t+\gamma \text{ max}_a Q(s_{t+1},a) – Q(s_t,a_t)]$
$k \leftarrow k+1$
end
$s_t \leftarrow s_{t+1}$
until s is terminal
until stop

3.6 *Temporal-Difference Learning with Adam Optimizer*

The Adam (Adaptive Moment Estimation) optimizer is an efficient stochastic optimization algorithm that requires only first-order gradients with a small memory footprint. The algorithm computes the first and second moments of the gradients, which are the estimates of the mean and uncentered variance of the gradients, respectively. These moments are used to update the model parameters [12].

Algorithms 9 and 10 contain pseudocodes that represent the subsequent steps to update the Q table parameters using the Q-learning and Sarsa methods with the Adam optimizer. The Adam algorithm introduces two hyperparameters β_1 and β_2, which control the decay rates of the first and second moments, respectively. Typically, these parameters have the values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, which have been found to work very well in practice [12]. The algorithm starts by setting the first m_t and second (v_t) moment variables equal to zero. During each training iteration, the gradients of the model parameters with respect to the loss function are computed. The first and second moments are then updated using the exponential moving averages.

Algorithm 9.
Q-Learning algorithm with Adam optimizer
Setting parameter values: and α = 0.01, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$, γ = 0.99.
Initialize:
$f(s,a;\theta) \leftarrow 0$, for all $s \in S$, $a \in A$
$m_0 \leftarrow 0$, (1^{st} moment vector)
$v_0 \leftarrow 0$, (2^{st} moment vector)
repeat (for each episode):
Initialize timestep: $t \leftarrow 0$
Initialize state: s_0
repeat (for each step t of episode):
$Q(s,a) \leftarrow f(s,a;\theta)$
$a_t \leftarrow$ ε- greedy(s_t,Q)
r_t, $s_{t+1} \leftarrow$ Env(s_t,a_t)
$y \leftarrow r_t+\gamma \text{max}_a Q(s_{t+1},a)$
$\Delta Q \leftarrow Q(s_t,a_t) - y$
$\nabla_\theta f(s,a;\theta) \leftarrow 0$
$\nabla_\theta f(s_t,a_t;\theta) \leftarrow \nabla\theta\, f(st,at;\theta) + \Delta Q$
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f(s,a;\theta)$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$

$\hat{\alpha} \leftarrow \alpha \sqrt{1-\beta_2^t} / (1-\beta_1^t)$

$f(s,a;\theta_t) \leftarrow f(s,a;\theta_{t-1}) - \hat{\alpha}\left(m_t / (\sqrt{v_t} + \varepsilon)\right)$

$s_t \leftarrow s_{t+1}$

until s is terminal

until stop

Algorithm 10.

Sarsa algorithm with Adam optimizer

Setting parameter values: and $\alpha = 0.01$, $\beta 1 = 0.9$, $\beta 2 = 0.999$, $\varepsilon = 10\text{-}8$, $\gamma = 0.99$.

Initialize:

$f(s,a;\theta) \leftarrow 0$, for all $s \in S$, $a \in A$

$Q(s,a) \leftarrow 0$, for all $s \in S$, $a \in A$

$m_0 \leftarrow 0$, (1^{st} moment vector)

$v_0 \leftarrow 0$, (2^{st} moment vector)

repeat (for each episode):

Initialize timestep: $t \leftarrow 0$

Initialize state: s_0

$a_0 \leftarrow \text{argmax}_a Q(s_0,a)$

repeat (for each step t of episode):

$Q(s,a) \leftarrow f(s,a;\theta)$

$r_t, s_{t+1} \leftarrow Env(s_t,a_t)$

$a_{t+1} \leftarrow \varepsilon\text{- greedy}(s_{t+1},Q)$

$y \leftarrow r_t + \gamma\, Q(s_{t+1},a_{t+1})$

$\Delta Q \leftarrow Q(s_t,a_t) - y$

$\nabla_\theta f(s,a;\theta) \leftarrow 0$

$\nabla_\theta f(s_t,a_t;\theta) \leftarrow \nabla_\theta f(s_t,a_t;\theta) + \Delta Q$

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f(s,a;\theta)$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$

$\hat{\alpha} \leftarrow \alpha \sqrt{1-\beta_2^t} / (1-\beta_1^t)$

$f(s,a;\theta_t) \leftarrow f(s,a;\theta_{t-1}) - \hat{\alpha}\left(m_t / (\sqrt{v_t} + \varepsilon)\right)$

$s_t \leftarrow s_{t+1}$

$a_t \leftarrow a_{t+1}$

until s is terminal

until stop

4 RESULTS OF SIMULATION STUDIES

The simulation studies of the developed reinforcement learning algorithms were performed using MATLAB software. The environment in these simulations was a maze consisting of 20x20 = 400 fields, of which 56 fields, i.e. 14% of all fields, were static obstacles. Hence, the number of available environmental states was 344. The exemplary arrangement of the assumed obstacles is shown in Figure3, with obstacle positions marked grey. The initial field was located in the upper left corner of the maze, while the target field was in the lower right corner. A mobile agent can move from its current position to an adjacent position in one of four directions: north, south, east, or west, except for the directions in which it encounters obstacles or leaves the environment. In such a case, the agent maintains its current position. The main goal of the reinforcement learning algorithms was to determine the shortest path for a mobile agent from the initial position (upper left corner of the maze) to the target position (lower right corner of the maze). In the analysed case, the shortest path, consisted of 38 steps, and there are several paths in the maze, which met this condition. In the learning process, for each algorithm, the minimum number of episodes required to find the shortest path was sought. This learning stage was called path planning for a mobile agent. An episode is a single run of the reinforcement learning algorithm, searching for a path between the initial and target points. The learning process is an iterative solution of the optimization problem, in which the total reward obtained in a single episode is maximized. In all algorithms, for which the results are shown in Table 1, the agent, after reaching the target position, received a reward of 50, while in the remaining possible fields, a reward of ◎1. The fields containing static obstacles were inaccessible to the agent.

Figure 3 shows the routes found by the dynamic programming algorithms (policy iteration and value iteration), both routes are 38 steps long. In these algorithms, during the learning stage, the mobile agent could also move into forbidden fields, which were treated as holes. After reaching the target position, the agent received a reward of 50, but on each holes-type fields it got a reward of ◎10, and on the remaining possible fields a reward of 0.

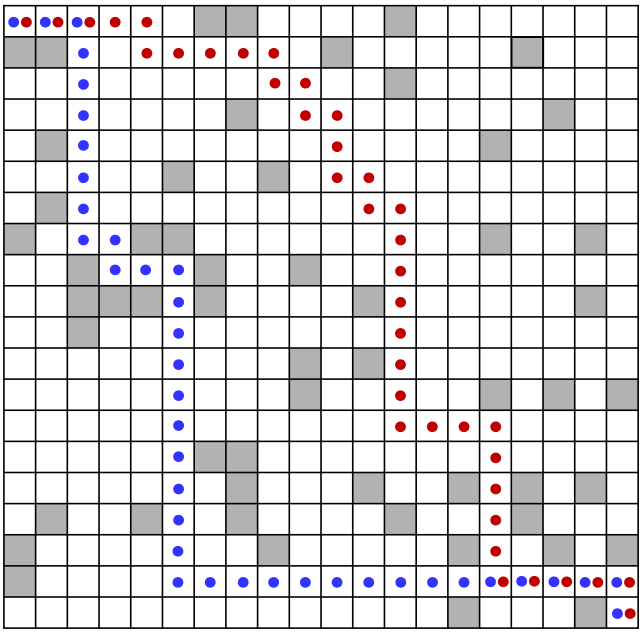

Figure 3. Paths planned by DP algorithms: policy iteration (Algorithm 1; red dots) and value iteration (Algorithm 2; blue dots).

Figure 4 presents the results of training three algorithms: Monte Carlo control, Q-learning and Sarsa. The figure shows the number of steps in individual episodes that the learning algorithm performed in search of a path between the initial and target points in the tested maze. If the learning algorithm did not find a path in the maximum number of steps planned for a single episode, then the episode was ended without reaching the goal. For the Q-learning and Sarsa algorithms, the maximum number of steps in a single episode was constant and amounted to 1000. The situation was different in the Monte Carlo control algorithm, in which the initial value of the maximum number of steps in a single episode was 100000, and only after finding the path to the target point for the first time it was reduced to 10000. Then, after the next three episodes in which it was possible to find a path to the target point, it was reduced to 1000. The minimum number of episodes required to run each learning algorithm before finding the shortest transition path consisting of 38 steps is given in Table 1. The transition routes obtained with the minimum number of episodes required to run the algorithm are shown in Figure 5. Of these three algorithms presented

in Figures 4 and 5, the best result was obtained for the Q-learning algorithm, which required running 7000 episodes in the learning process.

Figure 6 presents the results of training three algorithms: Q(λ), Sarsa(λ) and Dyna -Q. Similarly to Figure 4, the graphs shows the number of steps in individual episodes. For the Q(λ) and Sarsa(λ) algorithms, the values of λ were selected in such way that they gave the minimum number of episodes required to find the shortest path, consisting of 38 steps, between the initial and target points. The paths obtained for each of these algorithms are shown in Figure 7, with the number of episodes given in Table 1. For the Dyna-Q algorithm, the value of N = 10 was selected, which allowed it to obtain similar training results as the other two algorithms: Q(λ) and Sarsa(λ). Out of this group, the best result was obtained for the Sarsa(λ=0.94) algorithm.

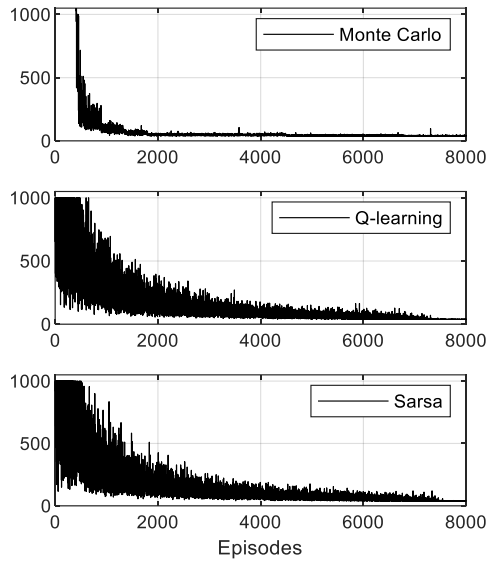

Figure 4. Comparison steps per episodes of algorithms: Monte Carlo control (Algorithm 3), Q-learning (Algorithm 4) and Sarsa (Algorithm 5).

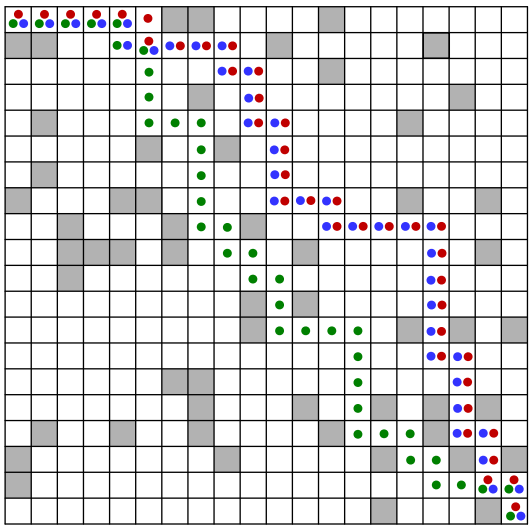

Figure 5. Paths planned by algorithms: Monte Carlo control (green dots), Q-learning (blue dots) and Sarsa (red dots) with the number of Episodes from Table 1.

The results of training the last three analyzed algorithms: Dyna-Q again, but this time with N=30, and Q-learning and Sarsa, both with the Adam optimizer are shown in Figure 8. The graphs shows the number of steps in individual episodes. The transition routes obtained for these algorithms are shown in Figure 9.

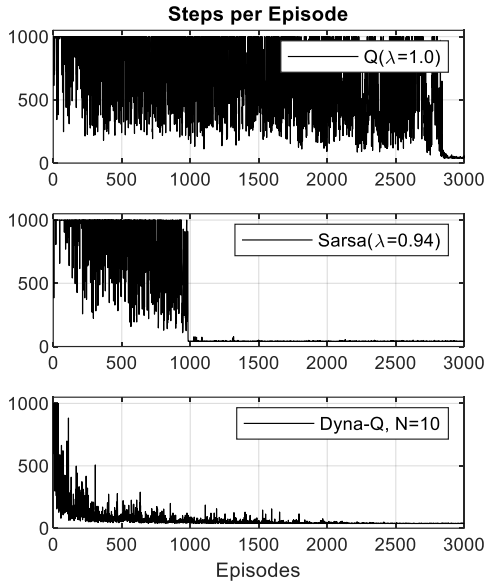

Figure 6. Comparison steps per episodes and sums of rewards per episodes of algorithms: *Q(λ)* (Algorithm 6); Sarsa(λ) (Algorithm 7) and Dyna-Q (N=10; Algorithm 8) .

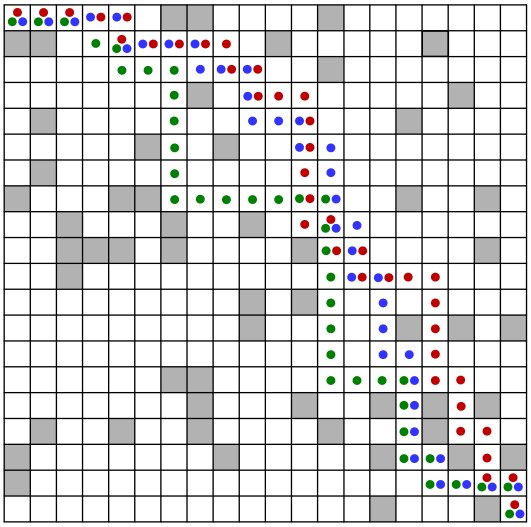

Figure 7. Paths planned by algorithms: Dyna-Q (N=10; green dots), *Q*(λ) (blue dots) and Sarsa(λ) (red dots) with the number of Episodes from Table 1.

It is noteworthy that the best results of all algorithms tested have been obtained for this group of algorithms. Analyzing the results shown in Figure 8, it can be seen that the Dyna-Q algorithm (N=30) finds a transition route close to the optimal one quite quickly, but then these consecutive routes found are still longer than the shortest one and analysing many more episodes was needed to determine the shortest transition route.

The situation is different with the Q-learning and Sarsa algorithms with the Adam optimizer, which after finding the shortest transition route very quickly, also quickly stabilize the result on this solution. In this group of algorithms, the best result was obtained for the Q-learning algorithm with the Adam optimizer, and this algorithm, which required running 210

episodes in the learning process, was considered the best of all algorithms tested.

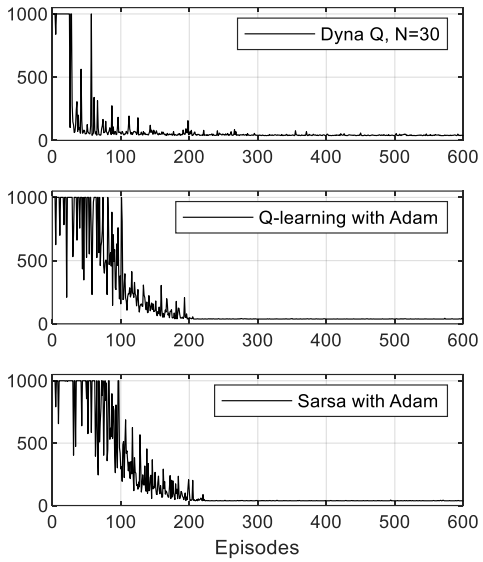

Figure 8. Comparison steps per episodes and sums of rewards per episodes of algorithms: Q(λ) (Algorithm 6); Sarsa(λ) (Algorithm 7) and Dyna-Q (N=10; Algorithm 8) .

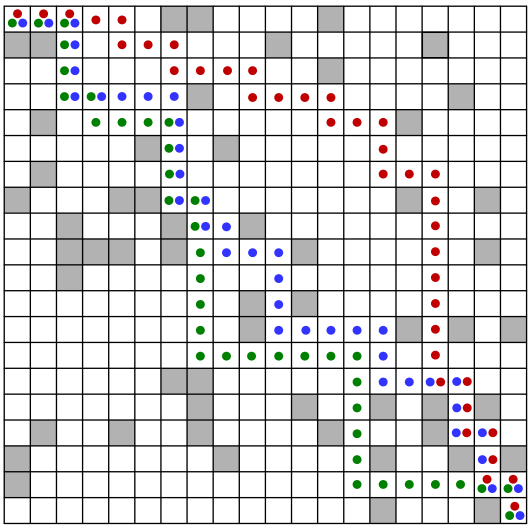

Figure 9. Paths planned by algorithms: Dyna-Q (*N*=10; green dots), Q(λ) (blue dots) and Sarsa(λ) (red dots) with the number of Episodes from Table 1.

Table 1. The performance of the algorithms Monte Carlo control, Q-learning, Sarsa, Q(λ), Sarsa(λ), Dyna-Q, Q-learning with Adam optimizer, Sarsa with Adam

Algorithm	Episodes	Max steps per episode
Monte Carlo control	9000	1000[1]
Q-learning	7000	1000
Sarsa	7450	1000
Q(λ)	2900	1000
Sarsa(λ)	1200	1000
Dyna-Q, (N=10)	2400	1000
Dyna-Q, (N=30)	600	1000
Q-learning with Adam	210	1000
Sarsa with Adam	225	1000

[1] This value has been finally set after finding four times the path to the target position, with the initial value of 100000 - for the first time, and the value of 10000 - for the second, third and fourth times.

5 CONCLUSIONS

The paper evaluates a number of selected reinforcement learning algorithms used to solve the problem of trajectory planning for a mobile agent in the environment with static obstacles. The most important evaluation criterion was the speed of determining the shortest path between the initial and target points. A wide range of algorithms was considered, including dynamic programming, Monte Carlo, temporal-difference learning, planning and learning, and the use of the Adam optimizer.

The principle of operation of the dynamic programming algorithms, which consists in: policy iteration and value iteration is completely different from that of remaining algorithms, as it requires the knowledge of the exact model of the environment to determine the agent's policy allowing for finding the minimum length of the transition route between the initial and target points, which in this case was equal to 38 steps.

The remaining algorithms did not require the knowledge of the exact mathematical model of the environment and they taught the agent a policy that minimized the transition path between the initial and final points by gathering the experience based on the impact on the environment. In this group of algorithms, the simulation studies have shown that the Q-learning and Sarsa algorithms, in which the Adam gradient optimizer was used, were much faster than the others to find the optimal transition path between the initial and target points. The analysis of the obtained simulation results shows that the Q-learning algorithm with the Adam optimizer can be considered the best choice for finding the transition path for a mobile agent in a real environment.

ACKNOWLEDGMENTS

This research was funded by project of Gdynia Maritime University in Poland, No. WE/2025/PZ/03: "New methods of controlling the motion of an autonomous ship in open and restricted waters".

REFERENCES

[1] M. N. Ab Wahab, A. Nazir, A. Khalil, B. Bhatt, M. H. Mohd Noor, M. F. Akbar, and A. S. A. Mohamed, "Optimised path planning using enhanced firefly algorithm for a mobile robot," PLoS ONE, vol. 19, no. 8, e0308264, Aug. 2024, doi: 10.1371/journal.pone.0308264.

[2] R. E. Bellman, "A Markovian decision process," J. Math. Mech., vol. 6, no. 5, pp. 679-684, 1957, doi: 10.1512/iumj.1957.6.56038.

[3] M. Blaich, M. Rosenfelder, M. Schuster, O. Bittel, and J. Reuter, "Fast grid based collision avoidance for vessels using A-star search algorithm," In Proceedings of the 17th International Conference on Methods and Models in Automation and Robotics, MMAR, Aug. 27-30, 2012; pp. 385-390, doi: 10.1109/MMAR.2012.6347858.

[4] P. Brundavani, and Y. Avanija, "Robot path planning and tracking with the flower pollination search optimization algorithm," J. Comput. Allied Intell., vol. 2, no. 4, pp. 70-81, Aug. 2024, doi: 10.69996/jcai.2024020.

[5] C. Chen, X.-Q. Chen, F. Ma, X.-J. Zeng, and J. Wang, "A knowledge-free path planning approach for smart ships

based on reinforcement learning," Ocean Eng., vol. 189, e106299, Aug. 2019, doi: 10.1016/j.oceaneng.2019.106299.
[6] J. Chen, J. Liang, and Y. Tong, "Path planning of mobile robot based on improved differential evolution algorithm," In Proceedings of the 16th International Conference on Control, Automation, Robotics and Vision, ICARCV, Dec. 13-15, 2020, pp. 811-816, doi: 10.1109/ICARCV50220.2020.9305415.
[7] E. W. Dijkstra, "A note on two problems in connexion with graphs," Numer. Math, vol. 1, pp. 269-271, 1959.
[8] H. Dong, Z. Ding, and S. Zhang, Eds. "Deep Reinforcement Learning: Fundamentals, Research and Applications," Springer Nature Book, Jun. 2020, https://deepreinforcementlearningbook.org.
[9] L. E. Dubins, "On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents." Am. J. Math., vol. 79, pp. 497-516, 1957, doi: 10.2307/2372560.
[10] V. Francois-Lavet,P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An Introduction to deep reinforcement learning," Found. Trends Mach. Learn., vol. 11, pp. 219-354, Nov. 2018, doi: 10.48550/arXiv.1811.12560.
[11] R. Jaramillo-Martínez, E. Chavero-Navarrete, and T. Ibarra-Pérez, "Reinforcement-learning-based path planning: A reward function strategy," Applied Sci., vol. 14, e7654, Aug. 2024, doi: 10.3390/app14177654.
[12] D.P. Kingma, and J. L. Ba, "Adam: A method for stochastic optimization," Computing Research Repository, Dec. 2014, https://arxiv.org/abs/1412.6980.
[13] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Annu. Res. Rep., Computer Science Department, Iowa State University. Ames, Iowa, USA, 1998. https://api.semanticscholar.org/CorpusID:14744621.
[14] A. Lazarowska, "Comparison of discrete artificial potential field algorithm and wave-front algorithm for autonomous ship trajectory planning," IEEE Access, vol. 8, pp. 221013-22102, Dec. 2020, doi: 10.1109/ACCESS.2020.3043539.
[15] Y. Li, J. Zhao, Z. Chen, G. Xiong, and S. Liu, "A robot path planning method based on improved genetic algorithm and improved dynamic window approach," Sustainability, vol. 15, no. 5, e4656, Mar. 2023, doi: 10.3390/su15054656.
[16] M. Lin, K. Yuan, C. Shi, and Y. Wang, "Path planning of mobile robot based on improved A* algorithm," In Proceedings of the 29th Chinese Control And Decision Conference, CCDC, May 28-30, 2017, pp. 3570-3576, doi: 10.1109/CCDC.2017.7979125.
[17] L. Liu, J. Yao, D. He, J. Chen, J. Huang, H. Xu, B. Wang, and J. Guo, "Global dynamic path planning fusion algorithm combining jump-A* algorithm and dynamic window approach," IEEE Access, vol. 9, pp. 19632-19638, Jan. 2021, doi: 10.1109/ACCESS.2021.3052865.
[18] L. Liu, L. Li, H Nian, Y. Lu, H. Zhao, and Y. Chen, "Enhanced grey wolf optimization algorithm for mobile robot path planning," Electronics, vol. 12, no. 18. e4026, Sep. 2023, doi: 10.3390/electronics12194026.
[19] J. Luo, Z. X. Wang, and K. L. Pan, "Reliable path planning algorithm based on improved artificial potential field method," IEEE Access, vol. 10, p. 108276-108284, Oct. 2022, doi: 10.1109/ACCESS.2022.3212741.
[20] W. Min, L. Mo, B. Yin, and S. Li," An improved cuckoo search algorithm and its application in robot path planning," Appl. Sci., vol. 14, no. 20, e9572, Oct. 2024, doi: 10.3390/app14209572.
[21] D. K. Muhsen, F. A. Raheem,and A. T. Sadiq, "A Systematic review of rapidly exploring random tree RRT algorithm for single and multiple robots," Cybern. Inf. Technol., vol. 24, pp. 78-101, Sep. 2024, doi: 10.2478/cait-2024-0026.
[22] G. Parlangeli, and G. Indiveri, "Dubins inspired 2D smooth paths with bounded curvature and curvature derivative," IFAC Proceedings Volumes, vol. 43, no. 16, pp. 252-257, 2010, doi: 10.3182/20100906-3-IT-2019.00045.
[23] G. A. Rummery, and M. Niranjan, "On-line Q-learning using connecionist systems," Technical Report, Cambridge University Engineering Department. Cambridge, England. 1994.
[24] D. Shan, S. Zhang, X. Wang, and P. Zhang, "Path-planning strategy: adaptive ant colony optimization combined with an enhanced dynamic window approach," Electronics, vol. 13, no. 5, e825, Feb. 2024, doi: 10.3390/electronics13050825.
[25] R. Singh, J. Ren, and X. Lin, "A review of deep reinforcement learning algorithms for mobile robot path planning," Vehicles, vol. 5, no. 4, pp. 1423-1451, Oct. 2023, doi: 10.3390/vehicles5040078.
[26] P. Sudhakara, and V. Ganapathy, "Trajectory planning of a mobile robot using enhanced A-star algorithm," J. Sci. Technol., vol. 9, no. 41, pp. 1-10, 2016, doi: 10.17485/ijst/2016/v9i41/93816.
[27] R. S. Sutton, "Learning to predict by the methods of temporal differences," Mach. Learn., vol. 3, pp. 9-44, 1988, https://10.1007/BF00115009.
[28] R. S. Sutton, and A. G. Barto, "Reinforcement Learning. An Introduction," 2d edition, MIT Press. Cambridge, MA, USA, 2020.
[29] R. Tang, X. Tang, and H. Zhao, "Enhancing whale optimization algorithm with differential evolution and Lévy flight for robot path planning." Int. J. Adv. Comput., Sci. Appl., vol. 15, no. 6, pp. 401-410, 2024, doi: 10.14569/IJACSA.2024.0150540.
[30] P. Vana, and J. Faigl, "Optimal solution of the generalized Dubins interval problem: Finding the shortest curvature-constrained path through a set of regions," Auton. Robots, vol. 44, pp. 1359-1376, Aug. 2020, doi: 10.1007/s10514-020-09932-x.
[31] S. X. Wang, "The improved Dijkstra's shortest path algorithm and its application". Procedia Eng., vol. 29, pp. 1186-1190, Feb. 2012, doi: 10.1016/j.proeng.2012.01.110.
[32] X. Wang, Z. Liu, and J. Liu, "Mobile robot path planning based on an improved A-star algorithm," In Proceedings of the 2nd International Conference on Computer Graphics, Artificial Intelligence, and Data Processing, ICCAID, May 23, 2023, doi: 10.1117/12.2674526.
[33] Watkins, C. Learning from delayed rewards. Ph.D. dissertation, Cambridge University, Cambridge, U.K., 1989.
[34] Y. Xu, B. Sang, and Y. Zhang, "Application of improved sparrow search algorithm to path planning of mobile robots," Biomimetics, vol. 9, no. 6, e351, Jun. 2024, doi: 10.3390/biomimetics9060351.
[35] G. Zhang, Y. Deng, W. Zhang, and C. Huang, "Novel DVS guidance and path-following control for underactuated ships in presence of multiple static and moving obstacles", Ocean Eng., vol. 170, pp. 100–110, Dec. 2018, doi: 10.1016/j.oceaneng.2018.10.009.
[36] L. Zheng, W. Yu, G. Li, G. Qin, and Y. Luo, "Particle swarm algorithm path-planning method for mobile robots based on artificial potential fields," Sensors, vol. 23, no. 13, e6082, Apr. 2023, doi: 10.3390/s23136082.