This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.Doi Number

A low-cost Q-learning-based approach to handle continuous space problems for decentralized multi-agent robot navigation in cluttered environments

VAHID BABAEI AJABSHIR, MEHMET SERDAR GUZEL, ERKAN BOSTANCI
Computer Engineering Department, Ankara University, 06830 Ankara, Turkey
Corresponding author: Mehmet SERDAR GÜZEL (e-mail: mguzel@ankara.edu.tr).

ABSTRACT This paper addresses the problem of navigating decentralized multi-agent systems in partially cluttered environments and proposes a new machine-learning-based approach to solve it. On the basis of this approach, a new robust and flexible Q-learning-based model is proposed to handle a continuous space problem. As in reinforcement learning (RL) algorithms, Q-learning does not require a model of the environment. Additionally, Q-Learning (QL) has the advantages of being fast and easy to design. However, one disadvantage of QL is that it needs a massive amount of memory, and it grows exponentially with each extra feature introduced to the state space. In this research, we introduce an agent-level decentralized collision avoidance low-cost model for solving a continuous space problem in partially cluttered environments, followed by introducing a method to merge non-overlapping QL features in order to reduce its size significantly by about 70% and make it possible to solve more complicated scenarios with the same memory size. Additionally, another method is proposed for minimizing the sensory data that is used by the controller. A combination of these methods is able to handle swarm navigation low memory cost with at least18 number of robots. These methods can also be adapted for deep q-learning architectures so as to increase their approximation performance and also decrease their learning time process. Experiments reveal that the proposed method also achieves a high degree of accuracy for multi-agent systems in complex scenarios.

INDEX TERMS Adaptive algorithm, Continuous Space Problem, Multi-Agent Systems, Q-Learning.

I. INTRODUCTION

A) Swarm robotics and Q-learning

Robotic researchers extensively use systems with multiple robots in their studies as swarm intelligence. They are inspired by biological studies of animal aggregation that brings impressive and complicated act at group level [1], [2], multi-robot systems can be designed in a way that a desired collective behavior come out from the local individual interactions in between the agents and the environment [3]. This field of study combines swarm intelligence discipline and collective robotics to plan and coordinate robots to complete complex tasks in a timely fashion. In other words, multi-agent systems are aggregations of autonomous agents that, in some ways, resemble the swarm intelligence field [4], [5]. Other essential advantages of multi-robot systems include scalability, robustness, and flexibility that are discussed in [3], [6]–[8]. Robustness is the degree to which a device maintains its functionality in the face of partial failure or other abnormal conditions [9], [10]. As a result, robustness in the sense of Swarm robotics refers to the capacity of the robot swarm to operate even in case of some individual failures or if the environment is unpredictable. They are also scalable since they can be easily changed in number. Swarm robotics may be used to create flexible systems that adapt quickly to changing operating conditions and facilitate creating systems that can work in a wide variety of environments and conditions. There are numerous applications for multiple robot systems in different areas. Collective move in a decentralized manner with a given geometric pattern is investigated in [11], [12]. Mobile autonomous robotic sensors are discussed in [13] for solving coverage problem.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

Swarm robotics is the study of establishing groups of robots that function independently of external infrastructure or centralized control. The collective behavior of robots in a swarm is based on local interactions amongst the robots and also amongst the robots and the environment in which they operate. Swarm intelligence concepts direct and guide the creation of robot swarms. These principles facilitate the development of fault-tolerant, scalable, and versatile systems. When multiple tasks must be completed simultaneously, high redundancy and the absence of a single point of failure are needed. It is theoretically impossible to set up the infrastructure necessary to operate the robots in a centralized manner; swarm robotics appears to be a promising solution. Demining, search and rescue, planetary or underwater discovery, and surveillance are all tasks that may benefit from swarm robotics[14].

Q-Learning [15] is a model-free reinforcement learning algorithm that extracts the policies without directly modeling the environment. In other words, it is an adaptive control utilized where the system model is not known [16], [17]. Thus it is a more simple and more flexible algorithm in comparison with model-based algorithms [18] also; it is proven to converge on optimal Q-function [19]–[21]; on the other hand, experimental researches reveal that it requires more time and inspection samples to learn [22], [23].

RL agents learn by engaging with their complex environment [24]–[26]. The agent observes the state of the environment at each time stage and takes an action that causes the environment to change into a new state. The quality of each exchange in the state is evaluated by a scalar reward signal, and maximizing the cumulative reward is the agent's goal over the process of the interaction. The RL feedback (the reward) is less enlightening than supervised learning, in which the agent is given the appropriate actions to take [27]. However, in supervised learning, where there is no clear feedback on results, the RL feedback is more insightful [28]. Furthermore, the single-agent RL task can be solved using well-understood, verifiably convergent algorithms. This, combined with the simplicity and generality settings, makes RL appealing for multi-agent learning.

Although the individuals in a multi-agent system may be provided with pre-programmed behaviors, they often need to learn new behavior patterns online in order for the agent's or the multi-agent system's output to improve over time [25], [29]. This is generally due to the environment's complexity, which makes a priori design of good agent behaviors challenging, if not impossible. Furthermore, in a changing setting, instinctual behavior can become improper. Overall, for the goal-based navigation problem, it is conceivable to build a decentralized method for collective and collaborative behaviors of multi-agent systems [30].

B) Contribution

There are two informed [31] and non-informed [32] navigation types which refers to whether or not the agents have knowledge of the goal position or not. Our navigation is decentralized, informed and is similar to flocking, however, its main intention is to guide the swarm to the goal point as fast as possible so it does not give extra value or reward for attraction since in our problem this feature does not provide us any advantage. This is because being able to separate from the flock and choosing different paths by some individuals opens the path for others improving the overall performance.

This research utilizes multi-agent method described in [33]. According to which, each robot observes and reacts according to the state of the other robots that needs addition of extra features to state space for each robot. This approach has the advantage of being able to determine local collision-free action for swarms in cluttered workspaces efficiently [34]. This intensifies the main problem of Q-Learning that is its need for large memory. Introducing any new feature into state space increases Q-table's size and consequently increases memory size considerably, sometimes making it impractical to solve large problems. In this paper, we focus on the problem of navigating decentralized multi-agent systems in partially cluttered environments with a Q-learning-based model. We introduce a low-cost method, namely "Controller with compressed state space" or CCSS, to reduce the Q-Table size significantly in order to alleviate Q-learning's need for large memory. Lowering the memory size makes it possible to solve larger problems with the same amount of memory or reduce cost and increase feasibility.

In the experiments, we were able to successfully navigate a group of three robots in three separate scenarios with static and dynamic obstacles using the proposed Q-Learning-based approach. Utilizing the new method, we were able to add a third robot to the robot group; however, memory limitations of a regular computer (our test machine) restricted the addition of extra robots to the environment. This extra memory space could also be employed for further discretization of the state space for smoothening the actions and also making more realistic simulation results.

The proposed method of this paper is designed based on a decentralized strategy. Each robot observes the environment locally and takes the action without relying on a centralized controller for synchronization between robots thus not having any direct information about other robot's intensions. However, it is a good practice to hugely accelerate the learning process, alleviate branching factor problem [35] and gain more confidence on learned policies visiting most of the states in state space several times by sharing the observations of all robots only in learning phase without harming the decentralization concept in exploitation phase. We should bear in mind that sharing the observations is not a necessary task for our method.

Doing further investigation and considering the fact that three robots were not enough to constitute a swarm; we developed a new method for increasing the robot population to any scale,

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

namely "Generalized CCSS" or GCC. GCC prioritizes and reduces the sensory data making it possible to navigate any number of robots by using the same Q-table of the three robots. This is beneficial for both normal and deep Q-learning since it reduces the input feature size significantly. We should consider that while the effects CCSS in deep-q-learning requires further experiments, utilizing GCC on deep Q-learning will most probably be beneficial for swarm robotics in the sense of increasing robot count and decreasing input feature count and learning duration.

The proposed methods are tested and validated by carrying out two different sets of experiments. In the first set of experiments, we show the applicability and prosperity of CCSS by comparing it with the non-CCSS version or namely "Controller with uncompressed state space" or CUSS. The next set of experiments are conducted to validate GCC and show the prosperity of the method in navigating more than three robots in the same environments.

C) Related works

From navigation point of view utilized method in this paper is similar to flocking. It is a form of collective behavior that can be seen in nature. More precisely, It is the planned movement of a bunch of participants in a common direction [30], [32]. Methods using flocking utilize attraction and repulsion and some also use alignment components to decide for their behavior. [32], [36] A more recent decentralized navigation research is done in [37] utilizing Neural networks, however, the proposed method utilizes a centralized unit for local communication and sharing learned features by nearby agents so it is not fully decentralized.

The main subject of this research "Q-Learning" is utilized in diverse kinds of problems, such as inventory management [38], behavior design or navigation in robotics [12], [39], [40] and game play [41]–[43], and it is typically utilized in continuous state space problems by using parametric function approximation methods [40], [44].

Authors in [51] address the problem of coordinating multi-agents in a decentralized fashion. They solve the problem of StarCraft II which requires agent coordination and in order to do so, authors introduced a method named QMIX that learns in a centralized way without any communication constraints and extracts and trains decentralized policies. However, our method does not rely on coordination between agents and despite QMIX, learning phase and running phase both are done in a decentralized fashion.

There are some other researches that try to compress the state space and reduce dimensionality [35], [45]. These researches utilize a method named deep auto-encoders for compressing the state space utilizing neural networks, mainly with the aim of gaining generalization ability rather than memory usage reduction. However, our method does a generalization on terminal states and utilizes expert knowledge for memory usage reduction keeping most of the states intact. There are no data available about memory reduction percentage for these researches for comparison purpose.

The authors in [46] present a method for fitting linear action models to estimate expected outcomes, and then they utilize a strategy for compressing these linear action models, which reduces the amount of space available for learning models, to gain an advantage over Q-learning in adapting to changes in reward function. Another method is state aggregation for information compression (SAIC) [47] aims to compress the communication messages between the agents in a limited centralized multi-agent system. Ref. [48] proposes a compression scheme for shrinking Improved DeepRT (IDeepRT) which is a deep learning technique used to analyze the Cyber-Physical Systems (CPS) data. This is done by utilizing Bayesian matrix factorization in order to reduce the model runtime. In this paper, reinforcement learning is combined with IDeepRT to correct feature space representations in IDeepRT on a continuous basis. Ref. [49] provides a compression strategy for reducing the size of the experience by using an encoder and decoder to shrink the size of the streaming data in order to save bandwidth and power consumption.

For sharing the q-table [35], [50] utilize a normalization step for merging the q-table of each robot after each learning episode which requires extra processing and also extra memory for holding number of times that each agent visited each state, e.g., if we name this table as visit-table in python language, selecting float type (16bits) for q-table and int (12 bits) for visit-table, it increases the memory usage by 75% multiplied by the number of agents, making it even more costly and even impractical for systems with large agent count. On the other hand, it has the advantage of learning in real-time during each episode. However, in our method each robot takes turn to exploit and update the Q-table synchronously at each time-step rather than each episode so it does not need extra normalization and merge phase or extra memory for keeping visit count of each agent and thus, it is faster overall. While this method fits well for many problems it might not be suitable for some real-time problems.

In terms of communication data size both methods only need to send the data about current state, action and the result of the action that will require a small amount of bandwidth. In our example this will be the sum of nine int type size and a real number type size.

D) Paper Overview

The following is a description and overview of the proposed navigation systems: The problem definition and foundation of this work are described in Section 2. The development of a novel low-cost Q-Learning-based controller with compressed state space (CCSS) and a Generalized Q-Learning-based Controller with compressed state space (GCC) is addressed in Section 3, whereas Section 4 focuses on the implementation and evaluation of the proposed navigation systems. Finally, the study is concluded in Section 5.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

II. PROBLEM STATEMENT

Navigating decentralized multi-agent systems in partially cluttered environments is a challenging problem. In our designed scenarios, robots must manage to navigate from a short path to a goal state without colliding with any dynamic (i.e., other robots) or static obstacles, all using copies of the same or very similar controller models depending on the learning strategy. The lack of a centralized control unit for coordinating the robots makes it challenging to avoid collision with dynamic obstacles and also plan for an optimal path. Moreover, the problem's actual state space is continuous and demands a huge Q-Table. This is very important because for this problem, even by discretizing the state space to larger units, the limits for Q-Table size are easily reached. Introducing a few more features make it impossible to utilize Q-learning on practical problems [30]. On the other hand, Q-learning is a model-free algorithm that does not require the model of the environment to be provided priorly.

The formal model of the Markov decision process is utilized for single-agent RL[33].

Definition 1. $\langle X, U, f, \rho \rangle$ Tuple is a finite Morkov decision process where X denotes the finite number of possible environment states. U specifies the finite set of agent actions $f: X \times U \times X \rightarrow [0,1]$ is the probability function for the transition of states, and $\rho: X \times U \times X \rightarrow \mathbb{R}$ is the function for reward.

The environment is defined by the state $x_k \in X$ on every discrete time step k. The agent observes and evaluates the state and takes action $u_k \in U$. As a consequence, the environment state changes to some $x_{k+1} \in X$ according to the transition probabilities given by f: after u_k is executed in x_k, the probability of arriving in x_{k+1} is $f(x_k, u_k, x_{k+1})$. According to the reward function $\rho: r_{k+1} = \rho(x_k, u_k, x_{k+1})$, the robot agent receives a scalar reward $r_{k+1} \in \mathbb{R}$. The immediate impact of action u_k is evaluated by the reward, i.e., the transformation of the environment state from x_k to x_{k+1}. However, it says nothing about the long-term consequences of this action. The reward function is assumed to be bounded.

The f or the transition probability function is replaced by a more straightforward transition function, $\bar{f}: X \times U \rightarrow X$, for deterministic systems. As a result, the reward is entirely dependent on the current state and action: $r_{k+1} = \bar{\rho}(x_k, u_k), \bar{\rho}: X \times U \rightarrow \mathbb{R}$. There are terminal states in several Markov decision processes, which are states that cannot be left once entered. All rewards obtained from a terminal state are zero. Throughout this, the learning process is usually divided into distinct trials (episodes), which are trajectories or set of states, actions and rewards that begin at one state and end in a terminal state.

The agent's behavior is defined by its policy, determining how the agent chooses its actions given the current state. The policy could be either deterministic, $\bar{h}: X \rightarrow U$ or stochastic, $h: X \times U \rightarrow [0,1]$. Policies that do not change over time are called stationary. The agent's purpose is to discover a policy that maximizes the anticipated discounted return from any state x:

$$R^h(x) = E\left\{\sum_{k=0}^{\infty} \gamma^k r_{k+1} | x_0 = x, h\right\} \quad (1)$$

Where under the policy h, the expectation is taken over the probabilistic state transitions, and $\gamma \in [0,1)$ is the discount factor. The long-term reward accumulated by the agent is compactly represented by the return R. The discount factor γ can be thought of as a way of encoding growing uncertainty about future rewards or as a way of constraining a sum that would otherwise grow boundless.

The agent aims to optimize its long-term success while only getting feedback on its short-term, one-step performance (reward). It can do this by calculating the ideal state-action value function (Q-function). The expected return received by the policy h from any state-action pair is given by the Q-function $Q^h: X \times U \rightarrow \mathbb{R}$: $Q^h(x,u) = E\{\sum_{k=0}^{\infty} \gamma^k r_{k+1} | x_0 = x, u_0 = u, h\}$.

Bellman equation is the central element of the proposed Q-Learning algorithm. It helps find the optimal solution for a complex problem by breaking it down into simpler, recursive sub-problems and finding their optimal solutions [49]. The optimal Q-function is denoted by $Q^*(x,u) = max_h Q^h(x,u)$ and the Bellman optimization equation is satisfied using it:

$$Q^*(x,u) = \sum_{x' \in X} f(x,u,x') \left[\rho(x,u,x') + \gamma \max_{u'} Q^*(x',u')\right] \quad \forall x \in X, u \in U \quad (2)$$

According to the equation, the expected immediate reward plus the future discounted reward (optimal value attainable from the next state) is the optimal value of taking u in x. Since X is finite, the expectation is plainly articulated as a sum.

Once Q^* is determined, an optimal policy (i.e., one that gives the maximum return) can be calculated by selecting the action with the highest optimal Q-value in each state: $\bar{h}^*(x) = arg \max_u Q^*(x,u)$

When several actions achieve the highest Q-value, any policy can be selected between them, and the policy remains optimum. In this case, the 'arg' operator is described as returning just one of the equally good solutions, both here and in the sequel. The mentioned policy is said to be greedy since it tries to maximize the Q-Function. As a result, finding an optimal policy requires first evaluating Q^* and thereafter computing a greedy policy in Q^*.

Equation (2) is turned into an interactive approximation procedure [19]. Q-learning begins with an arbitrary Q-function, observes transitions $(x_k, u_k, x_{k+1}, r_{k+1})$, and changes the Q-function with each step.

$$Q_{k+1}(x_k,u_k) = Q_k(x_k,u_k) + \alpha_k \left(r_{k+1} + \gamma \max_{u'} Q_k(x_{k+1},u') - Q_k(x_k,u_k)\right) \quad (3)$$

It uses temporal difference that is the difference between the optimal Q-value of (x_k, u_k) and the current estimate $Q_k(x_k, u_k)$ and the updated estimate $r_{k+1} +$

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

$\gamma \max_{u'} Q_k(x_{k+1}, u')$. This new approximation is based on a sample of the right side of the Bellman equation (2), which has been applied to Q_k in the pair of state-action (x_k, u_k). x' is replaced in our sample by the observed next state x_{k+1}, and observed reward r_{k+1} is put instead of $\rho(x_k, u_k, x')$. The learning rate that is $\alpha \in (0,1]$ is usually time-varying and usually decreases over time.

According to [19], [21], [52], under the following conditions, the Q_k sequence provably converges to Q^*:

First, every state-action pair is experienced infinitely, often asymptotically. Next, for each state-action pair, explicit, distinct values of the Q-function are stored and modified. Additionally, while $\sum_{k=0}^{\infty} \propto_k$ is infinite, the sum expression $\sum_{k=0}^{\infty} \alpha_k^2$ is finite.

The first criterion can be met if, among other conditions, the agent continues to try all actions in all states with greater than zero probabilities. This is known as exploration, and it can be achieved by selecting a random action with probability $\varepsilon \in (0,1)$ and a greedy action of $(1-\varepsilon)$ probability at each step. By doing this, the third criterion, which is the ε-greedy exploration procedure, is obtained. In most cases, the probability ε decreases with time.

The Markov decision method has been generalized to the multi-agent case, sometimes referred to as stochastic game, as follows [33]:

Definition 2. A tuple $\langle X, U_1, \dots, U_n, f, \rho_1, \dots, \rho_n \rangle$ is a stochastic game where a finite set of environmental states is denoted by X. n denotes the number of agents involved. The finite sets of actions accessible by agents are denoted by $U_i, i = 1, \dots, n$, generating the joint action set $U = U_1 \times \cdots \times U_n$, the state transition probability function is $f : X \times U \times X \to [0,1]$, and the reward functions for the agents are denoted as $\rho_i : X \times U \times X \to \mathbb{R}, i = 1, \dots, n$.

The reward functions are assumed to be bounded, and the state transitions are the outcome of all the agents working together in the case of a multi-agent system; that is, $u_k = \left[u_{1,k}^T, \dots, u_{n,k}^T\right]^T, u_k \in U, u_{i,k} \in U_i$ in which T is the vector transpose. The policies $h_i : X \times U_i \to [0,1]$ combine to shape the h joint policy. Since the agents' $r_{i,k+1}$, rewards are contingent on joint action, their results are contingent on the joint policy: $R_i^h(x) = E\{\sum_{k=0}^{\infty} \gamma^k r_{i,k+1} | x_0 = x, h\}$.

Every agent's Q-function is determined by their joint action and policy, $Q_i^h : X \times U \to \mathbb{R}$, with $Q_i^h(x,u) = E\{\sum_{k=0}^{\infty} \gamma^k r_{i,k+1} | x_0 = x, h\}$. The reward functions for all agents in cooperative stochastic games are the same: $\rho_i = \cdots = \rho_n$. What is more, they all have the same goal of maximizing the common return and also the same outputs, $R_1^h = \cdots = R_n^h$.

Q-learning algorithm is given in Algorithm 1:

Algorithm 1 Q-Learning

Require:
- State Set $X = \{x_1, \dots, x_n\}$
- State Set $U = \{u_1, \dots, u_n\}$
- Reward function $\rho : X \times U \to \mathbb{R}$
- Learning rate $\alpha \in (0,1]$
- Discount factor $\gamma \in (0,1]$
- Exploration trade off $\tau \in (0,1]$

```
Procedure Q-Learning (X, U, ρ, α, γ, τ)
    Initialize Q: X × U → ℝ
    While Not Converged do
        Start in state x_k ∈ X
        While State is not terminal do
            Choose u_t ∈ U using ε − greedy policy
            Act u_k, observe reward r_k and new state s_{k+1}
            Utilize equation (3) and get Q_{k+1}(x_k, u_k)
            x_k = x_{k+1}
        End while
    End while
    Return Q
End procedure
```

Table I lists the symbols we use to denote key parameters throughout the study for your convenience.

Table1. **List of Symbols**

X	finite number of possible environment states
U	finite set of agent actions
f	probability function for the transition of states
ρ	reward function
R	anticipated discounted return
h	policy
α	learning rate
γ	discount factor
τ	exploration trade off
Q	quality function
d_g	distance to goal
a_g	relative angle of the goal
a_r	angle of the active robot
c_r	collision feature
g_r	goal feature
d_{r_i}	distance to i'th robot
a_{r_i}	relative angle of the i'th robot
t	time step count
r_g	goal reward
r_s	step reward
r_c	collision reward
s_c	step count

III.SCENARIO DEFINITION

In order to elaborate the problem in details it is better to first start with problem scenarios.

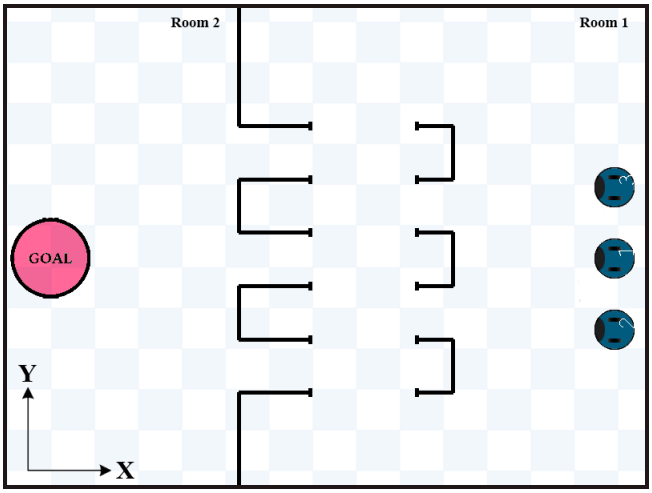
A. SCENARIOS FOR MULTI-AGENT PROBLEM

Figures 1a, 1b, and 1c show three of our sample scenarios in which three robots are expected to traverse a short route from the start point to the goal point. The robots are circular, and the action space consists of four actions, turn left, turn right, step forward and do nothing. This way, in order for a robot to go to a specific point, it first needs to turn towards the point by

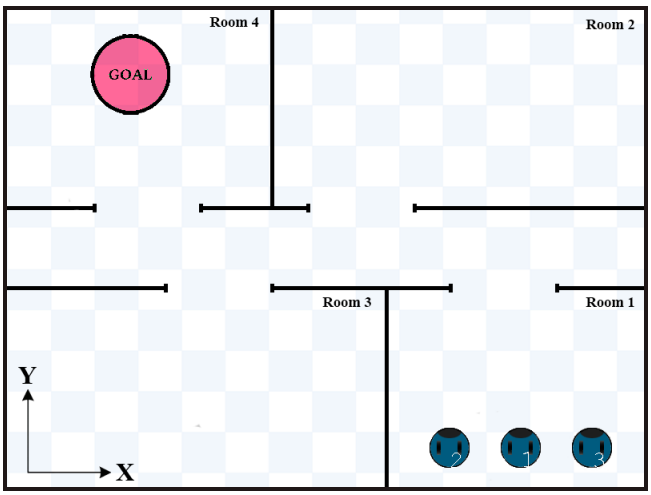
This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

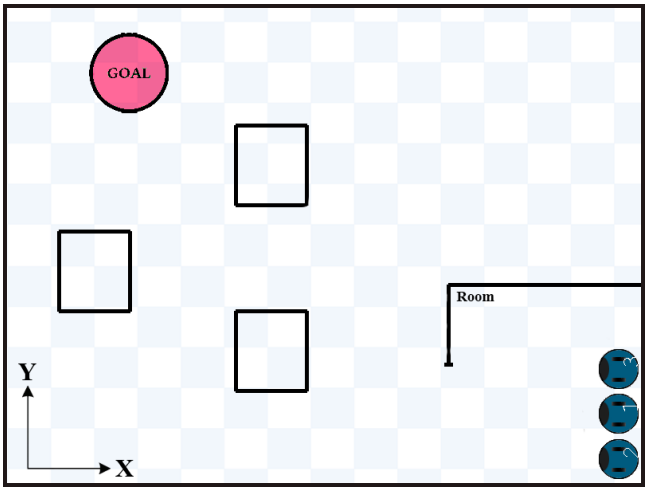
choosing one of the turn-left or turn-right actions, and then it can traverse the direct path by choosing step-forward action. Lines in the environment image represent the obstacles. The robots colliding with the obstacles will fail and will end the running episode and get a large negative reward. Robots reaching the goal point will get a large positive reward and wait for other robots to come. In this case, the episode finishes when all robots reach the goal point, or other robots wander around the environment for a long time, exceeding the maximum step counts. Robots also get a small negative reward in each step. This is for encouraging the robots to reach the goal from a short path. The designed system uses a decentralized controller that is placed on all robots and plans a collision-free short path for each of them at the same time.



(a)



(b)



(c)

FIGURE 1. Test scenarios: (a) a scenario with clattered U-shaped obstacles; (b) a scenario resembling an indoor map of a building. (c) a scenario with a room and cluttered obstacles.

B. DEFINING MODEL FEATURES FOR CUSS

One critical contribution of this paper is the compression of state space size in memory. State space is the set of all possible states of the system; in other words, it is a mathematical model that is represented by variables or sometimes called features. These features should be defined according to the simulation environment and also behaviors we want to design for our controller. Features should cover all information we need for the controller. In other words, we have to make sure we have enough features in which problem can be solved by doing one action in each state. For example, suppose we want our controller to avoid collision with obstacles according to our scenario and environment. In that case, we should include the object's actual position or relative position in our features, or we may handle collisions by giving a sense of position for the controller by adding coordinate features.

While designing the state space, having more than one action for a state means that we need to introduce an extra feature. To elaborate, say we have an environment with only one robot and one goal point. It might seem that it is enough only to introduce x, y position features for a controller to take the robot from any point in the environment to the goal point. Although this might work for robots that can go in any direction at each step but in our case, the robot first has to take action to turn into the right or left direction and then to go forward. This way, only the two x, y features will not be enough for finding the right action; since the controller does not have a sense of the robot's current direction and will try to put more than one action inside one state. Thus, the correct method for doing this will be to add the feature of the robot's direction. Despite the first method, if the robot turns left, it will find itself in a new state and may choose a different action.

According to the environment and behavior specifications, a standard state space for this scenario will have nine features.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

- The distance between the active robot and the goal point (d_g). This is the essential information for the controller in order to guide a robot to the goal.
- The relative angle of the goal point to the active robot (a_g). The relative angle of goal is the angle of line crossing the center of the active robot and goal point. It is the $arctan2(\nabla y, \nabla x)$. Without the angle of the goal point, the robot will not have a sense of relative position, and thus, it cannot find a path to the goal.
- The angle of the robot (a_r). The robot has to have information about its own direction in order to be able to decide whether it has to rotate left/right or not.
- Collision feature (c_r): This feature keeps relevant value for the collision. It has two states.
- Goal feature (g_r): This feature keeps if the robot has reached the goal state or not with two modes.
- The information of the other two robots: These features are repeated for each extra robot.
    - Distance to i'th robot (d_{r_i}): In order to avoid the collision with the other robots, an active robot needs to know the distance from that robot
    - Angle to i'th robot (a_{r_i}): In addition to the distance of the other robot, an active robot needs the relative angle to the other robot. Otherwise, the active robot will not be able to avoid collisions.

The first problem with these features is that except state features, the rest of the features are continuous and are needed to be discretized to be used in Q-Table. While representing the final results, it is possible to generalize the Q-function and make it seem continuous with smooth movements; this would be somehow like supervised learning in which the controller is able to predict values it has never seen before; e.g., using fuzzy inference system on Q-table. However, that is not the focus of this paper. One aspect that this paper wants to address is to propose a discretized model for solving a continuous state space problem of multi-agent robot navigation in a partially cluttered environment, and then compress the Q-table to optimize the memory usage. The proposed model is validated by performing a set of experiments.

Discretization of features refers to the process of converting or partitioning continuous features to discretized intervals, also coming with some degree of loss in accuracy. In other words, a range of values in the environment will be represented with one value making it possible to keep in a finite discretized array. The discretization amount is mainly limited by the size of the computer RAM. Thus, in the discretization process, we first prioritized the features by their importance; by doing some trials and errors, we decided which ones needed to discretize more and require more memory.

For example, maximum value for the distance to goal feature can be 950cm (it is the hypotenuse of the environment minus the robot's radius) and maximum value of the robot's angle can be 360degrees. In our tests we divided the distance-to-goal feature into eighteen parts and robot angle into 8 parts making their step size 52 and 45 respectively. If we had done the opposite the distance-to-goal step size would be about 119cm's which is not proper for solving the problem and in some cases, robots would not be able to pass the obstacles. According to this, by doing some trial and errors the highest to lowest discretization amount is given to distance-to-goal, angle-to-goal, distance-from-other-robots, active-robot's-angle, and angle to other robots, respectively.

$$S(d_1, \dots, d_n) = p.c. \prod_{i=1}^{n} d_i \tag{4}$$

$$\begin{aligned}(d_1^*, \dots, d_n^*) &= \underset{(d_1^*,\dots,d_n^*)\in\mathbb{N}^+}{\operatorname{argmax}} S(d_1^*, \dots, d_n^*) \\ &= \{d_1, \dots, d_n | S(d_1, \dots, d_n) \leq M\}\end{aligned} \tag{5}$$

The more formal definition of discretization process is given in equation (4), (5) and the flowchart in the figure 2. The Q-table size is calculated utilizing the equation (4) in which S is the Q-table size function, $d_1, \dots, d_n$ represent the discretization amount of each feature in the state space, p is the data-type size used in Q-table and c is the total action count defined in the environment. Objective is to maximize the discretization amount while staying under the maximum memory size M utilizing the equation (5).

C. DEFINING MODEL FEATURES FOR CCSS

As stated before, given features will introduce a vast state space, especially in a continuous space problem. After discretization, in search of a way to reduce its size, we found a way to stack non-overlapping states and reduce state space size. Additionally, we reduced the state count introduced with some features. By non-overlapping states, we mean the features which are not used simultaneously in one state. i.e., In this scenario, collision and goal states are non-overlapping and cannot happen at the same time. In addition to that, for our problem, we know that although there are many different positions in the environment in which collision can occur, there is a need for only one collision state in the problem's state space. The reason is that we do not need a differentiation between collision states for our scenario since they have the same negative reward and, the position of the collision is not crucial for these scenarios. All these reasons also apply to the goal feature, so a single state is enough for it too.

Bearing these reasons in mind, we found that instead of introducing a new variable for every feature, we can stack some of them on top of each other, and this way, we can reduce the state space size in memory significantly. Additionally, the feature discretization amount for some features can be reduced to only one. This does not reduce the actual state count of the problem but it will reduce the memory usage. We call this feature stacking.

It should be noted that feature stacking reduces the utilized memory size. That is because the Q-table size would not be

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

defined by multiplication of discretization amount of each feature and instead will be the result of multiplication of sum of stacked features. For example, let's assume that we have only three features in an environment, "A", "B" and "C" that are discretized into "10", "10" and "2" parts respectively with possible "4" actions for the agent. With this setup utilizing CUSS would require a "Q-table" with size "800" states in the memory obtained by the multiplication of "10×10×2×4". However, if it is decided to employ feature stacking of CCSS Q-table "480"states obtained by the multiplication of "10×12×4" will occupy the memory. As a result, utilizing CCSS algorithm reduces the memory size (memory cost) by 40% for this example.

This is crucial in Q-Learning because the main problem of this method is its Q-Table size that, according to problem space, grows polynomial, quadratic, or even faster, by adding features to state space or increasing the discretization amounts. In our experiments, we observed that by adding each extra feature or increasing discretization amount the size of Q-Table proliferates from megabytes to gigabytes or terabytes and makes it almost impossible to be filled or saved. The impact of feature stacking approach on deep Q-learning architectures might be beneficial but it needs further research involving complex experiments based on different scenarios.

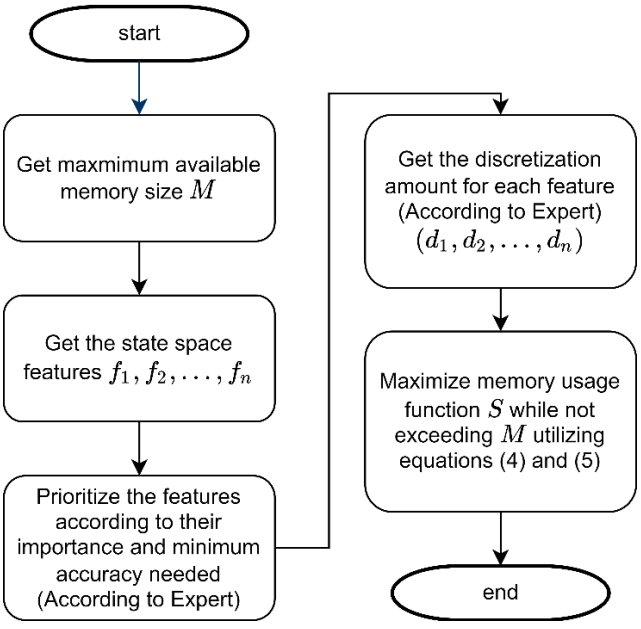


FIGURE 2. Feature discretization flowchart.

Taking these into consideration and analyzing features, we found three non-overlapping "Distance to the goal", "Collision", and "Goal-reached" features. These features are not needed at the same time, i.e., we will not need distance to the goal and goal-reached feature values after the robot collides. Therefore, these features are stacked on top of each other. By feature stacking, we mean putting the features on top of each other, and in practice, it is done by separating ranges of one variable for several features. This can be done by adding an adapter unit that piles up the feature values for adding into Q-Table and vice versa.

After feature stacking in the new state space, there remained five features instead of seven that are:

- d_g Distance-Status which will hold all distance to goal, collision, and goal reached features at the same time,
- a_g The relative angle of goal point to the active robot,
- a_r The angle of the robot.
- Information of the other two robots, these features are repeated for each extra robot.
    - d_{r_i} Distance to i'th robot
    - a_{r_i} Angle to i'th robot

We should pay attention that although these new features significantly reduce the required memory size, they do not affect the learning process and duration. Basically, this is because the actual state space is the same and still finding all states leading to a collision state is necessary. The learning graphs in the experiments section support this assertion.

C. DYNAMIC OBJECT PRIORITIZATION

The next problem we have is applying Q-Learning on a swarm of robots; however, we already know that if we continue using the same method, we have to add two extra features per extra robot. This makes it impractical to utilize Q-Learning on a swarm of robots even by using CCSS to compress the controller's state space. This is because each agent contributes its own variables to the combined state-action space, the complexity of the multi-agent reinforcement learning algorithm grows exponentially with the number of agents. As a result, the dimensionality curse in multi-agent reinforcement learning is more severe than in single-agent RL [33].

For solving this problem, we developed another method to make it possible to handle any number of robots using CCSS. To be able to do so we limited sensory data provided for the Q-Learning algorithm, and prioritize the sensory data given to the controller. The prioritization process determines which two robots' positions should be passed to CCSS. Figure 3 shows the priorities for the active robot, which is the blue robot in the center.



FIGURE 3. robot priorities

The controller on the active robot will be able to see and decide according to only two neighboring robots; so, it has to prioritize them according to their importance. The importance

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

is measured by the probability of causing a collision. According to our robots' movement mechanism, the robots in front are more important than the robots on the sides of the active robot or the robots in the rear. The next prioritization criterion is the distance of the other robots to the active one. The robots in the rear of the active robot are entirely omitted unless there are no other robots in front, even if they are close. That is because, in the short term, no action taken by the active robot can cause a collision with the robots in the rear. The pseudocode of the corresponding algorithm is given in Algorithm 2.

Algorithm 2 Prioritization

Require: $[(d_{r_1}, a_{r_1}),\ldots, (d_{r_n}, a_{r_n})]$ as DA (distance-angle of all nearby dynamic obstacles (robots))

Ensure: to select two most important robots to pass to the controller,

1: **Procedure** Prioritize Robots (*DA*)
2: *A* = **Sort** (*DA*, "*by rear or front*")
3: *B* = **Sort** (*A*, "*by distance*")
4: *C* = **AddAnglePriorityDimension**(*B*)
5: *D* = **Sort** (*C*, "*by angle priority*")
6: *E* = **GetTop2** (*D*)
7: ***Return*** *E*
13: ***End procedure***

Input for the prioritization algorithm is the angle-distance pairs of dynamic obstacles. This list contains a pair for all nearby robots. First, the algorithm sorts the input list according to obstacles' relative angle to the front of the active robot. The robots in the rear are sent to the end of the list. This way, the robots in the front will be favored more than the ones in the back-row, even if they are not as close. Next, the obstacles are sorted according to their distance from the active robot. Following that, the new sorted list is passed to a method to add angle priority dimension. Angle prioritization is done according to figure 3 that is discussed before. Then another sort is done based on this new dimension. Finally, the algorithm will return the top two distance-angle pairs of the obstacles. Utilization of this algorithm and its generalization ability with deep Q-learning for solving large swarm problems will decrease the learning duration process significantly.

C. THE OVERALL FLOW AT A GLANCE

Figure 4 demonstrates the flow of designing a system according to the paper's proposal. The first step in designing a GCC system is to define the problem and detail its specifications in order to extract its features. The next step is to have an expert to analyze the features to find the stackable ones and merge these features. After that, the features are needed to be discretized according to available memory as stated in to the flowchart in figure 4. The designed system so far will be able to solve many problems with lower memory usage in compare with CUSS however, we can put a step further and utilize GCC for enabling the system to handle any number of agents in a multiagent navigation system.

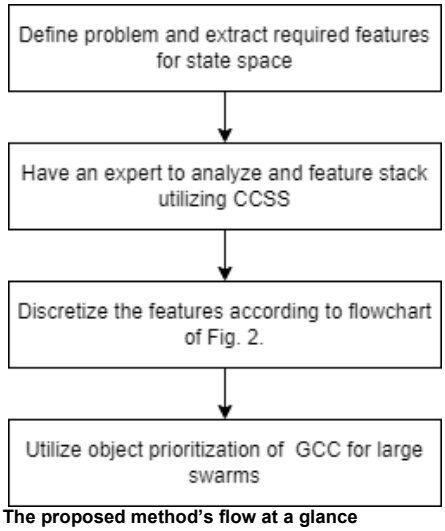


FIGURE 4. The proposed method's flow at a glance

IV. EXPERIMENTS

The proposed methods have been extensively tested in order to show their applicability, effectiveness, and optimization ratio. Experiments are conducted with an Intel Quad-core machine running at 2.8 GHz, with 16 GB of RAM memory in the windows operating system. The system was designed using the python programming language and Pygame [53]. Pygame is a cross-platform Python module set devised for writing video games and simulators and covers computer graphics and sound libraries.

In these sets of experiments, the proposed method described in the previous section was tested and compared with the uncompressed version under three different test scenarios. The scenarios' initial states are shown in Figures 1a, 1b and 1c. Also, three different controllers are designed based on state spaces mentioned in the previous section namely, "Controller with uncompressed state space" (CUSS), "Controller with compressed state space" (CCSS) and "Generalized CCSS"(GCC).

A. MEMORY USAGE COMPARISON

In conducted tests first, we compared the CUSS and CCSS methods' size in memory. Next, we compared and demonstrated the effectiveness of both methods in solving common navigation problems and learning speed. Then, we used the CCSS among with obstacle prioritization algorithm (GCC) for multi-robot navigation on the same scenarios.

In order to use Q-Learning for continuous state space, it is needed to be discretized. In the discretization process, although it would be smoother and more reliable to discretize the environment into more parts, there is a limitation for this. The more we discretize the state space, the more it will take it to learn, and more importantly, it will need more memory for calculations and saving the final model.

Two criteria determine the maximum discretization amount for a problem with continuous state space: the requisite feature count and RAM memory size. Features are determined mainly by the problem specifications; thus, their count cannot be

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

Table 2. **The number of states and memory size of CUSS and CCSS for comparing the needed memory**

	Robot count	Feature Discretization amount $(d_g, a_g, a_r, \begin{pmatrix} d_{r_1}, a_{r_1}, \dots, \\ d_{r_n}, a_{r_n} \end{pmatrix})$	No. of Q-Table elements CUSS	No. of Q-Table elements CCSS	Actual size in memory CUSS	Actual size in memory CCSS	Reduction Ratio
1	3	(6, 6, 4, (6, 4, …))	1,327,104	442,368	5.06MB	1.69MB	66.6%
2	3	(12, 12, 4, (12, 4, …))	21,233,664	6,193,152	81MB	23MB	70.8%
3	3	(18, 18, 8, (18, 8, …))	214,990,848	59,719,680	3280MB	911MB	72.2%
4	3	(24, 24, 12, (24, 12, …))	9,172,942,848	2,484,338,688	34992MB	9477MB	72.9%
5	3	(36, 36, 16, (36, 16, …))	110,075,314,176	29,047,652,352	419840MB	110592MB	73.6%
6	4	(6, 6, 4, (6, 4, …))	31,850,496	10,616,832	121.5MB	40.5MB	66.6%
7	4	(12, 12, 4, (12, 4, …))	1,019,215,872	297,271,296	3888MB	1134MB	70.8%
8	4	(18, 18, 8, (18, 8, …))	123,834,728,448	34,398,535,680	472064MB	117760MB	72.2%

easily changed without changing problem specifications, yet, discretization amount is changeable as far as it does not affect problem resolution. e.g., inadequate discretization in path planning problems may prevent obstacle avoidance. Hence, according to our needs, we found a discretization threshold for each feature by trial and error. It should be borne in mind that the smoothness of the actions also can be attained by different methods, e.g., putting a simple controller unit between the main controller and simulator or robot firmware with a duty of smoothening the actions by dividing one step into smaller steps.

The memory size utilized by the application has a direct relation with the feature count, discretization amount, and data type. After performing a lot of trials, the best discretization values for d_g (distance from goal), a_g (angle to goal), a_r (robot's current angle), d_{r_i} (distance from another robot), a_{r_i} (angle of the other robot) features are found to be 18, 18, 8, 18, and 8, respectively, that give us an acceptable smoothness in performing the actions and achieving successful navigation while not exceeding the available memory size. Additionally, we should bear in mind that the discretization is done according to importance of each feature as it is discussed in the previous section. The g_r (goal reached) and c_r (collision features) both have two states, so they will not change after discretization. With these values at hand, total state count and total memory usage for both methods are given in table 2.

Table 2 shows the size comparison between CUSS and CCSS with different discretizations for multi-agent navigation problems regarding Q-Table element count and actual size in memory. Discretizations are continued until a relatively reasonable amount of RAM for a powerful scientific computer. The features of the environment in the second column are explained in the previous section for both CUSS and CCSS. The results show a considerable amount of 70% reduction in memory usage amount using CCSS. It is worth mentioning that adding more robots was not practical; otherwise, the needed RAM amount would exceed reasonable limits, or the discretization amount would drop so that the problem could not be solved. For example, adding an extra fourth robot to the fourth row would need two extra features, increasing Q-Table size about 144 times needing about 10,566GBs of RAM for our tests.

The first column in Table 2 shows the robot count in the environment. The second column shows the different discretization amounts for each feature of the environment. The next two columns are the number of states calculated according to the discretization amount using CUSS and CCSS methods. The next two columns are the actual size of the Q-Tables in memory calculated in python. The last column is the reduction percentage of the Q-Table using CCSS over CUSS.

By considering preliminary experiment results, learning rate α is set to "0.7" and the discount factor γ is set to "0.9" for conducting final experiments. Performing the tests, we observed that giving an extra positive reward for actions that take the robot nearer to the goal not only made the learning process faster also helped the robot to get out of suboptimal loops faster. Dismissing this extra reward increases the learning process significantly. The robots sticking in a suboptimal loop return to the same state and choose the same action as before for several times, but with this mechanism, the problem does not last long; since after a while, the robot starts to prefer the action that takes it nearer to the goal. On the other hand, this mechanism extends the learning duration of passing local minima (i.e., U-shaped obstacles or empty rooms near the goal). However, according to our experiments, this is usually a negligible time in most scenarios. Another mechanism that we utilized to handle suboptimal loops was keeping the last n step taken by the robot and prevent choosing the same action in the same state in the same learning episode. This mechanism's job is to encourage the robot to explore in these situations.

B. EPISODE EVALUATION AND COMPLEXITY ANALYSYS

To evaluate each running episode and see the progress in learning, we made an equation that gives the highest to lowest reward in the following order. The highest reward is given to an episode in which the robot reaches the goal from the shortest path, and immediately comes the episode in which the

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

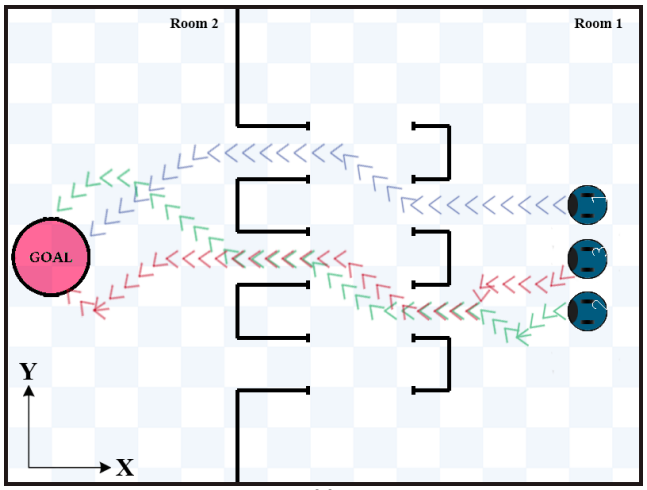
robot reaches the goal from a long path. Then the next high reward is given to an episode in which the robot does not reach the goal but also does not collide. This is because, in our observations, we realized that this case happens rarely and, in most cases, happens when the robot has learned to recognize the obstacles yet, cannot find a way to the goal. This means that it has significant progress in learning the environment; otherwise, it would crash into an obstacle after a while. Following these kinds of episodes is the reward in episodes in which the robot wanders for a while and then crashes with an obstacle. Next and the worst cases are the episodes in which the robots crash shortly after the episode starts. This usually happens at the beginning of the learning. According to these criteria, we come across equation 2, in which t represents the time step count, r_g is the goal reward, r_s represents the step reward, r_c is collision reward and s_c represents the step count. Multiplication of $|r_g/r_s|$ into r_g and multiplication of $|r_c/r_s|$ with r_c intensifies the difference between the mentioned situations and makes them more distinguishable.

$$e_r = \frac{tr_s + r_g * \left|\frac{r_g}{r_s}\right| + r_c * \left|\frac{r_c}{r_s}\right|}{s_c} \quad (6)$$

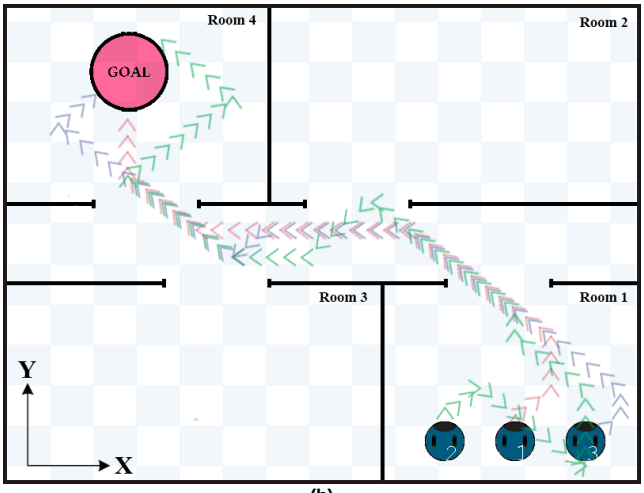
In terms of complexity we should consider that in our problem the agents do not have initial knowledge of topology of the state space, thus it is an unknown state space. Additionally, there are no duplicate actions in this problem and also at first Q-table initialized with zero. According to [54], [55] the worst-case complexity of reaching a goal or terminal state with these configurations is $O(n^3)$.

C. NAVIGATION RESULTS

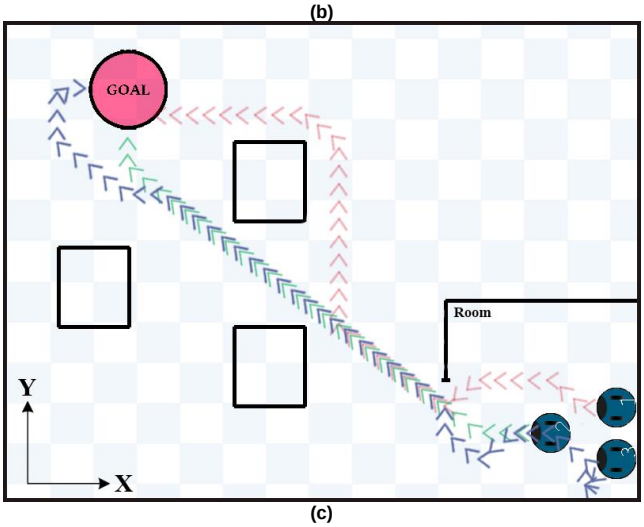
Looking at the results attained from trained models with both CUSS and CCSS, we saw their prosperity in navigating the robot from an admissible short and safe path to the goal. Results for both methods are very similar, so we only demonstrated the results of CCSS. The sample generated path for each scenario is shown in Figures 5a, 5b, and 5c.



(a)



(b)



(c)

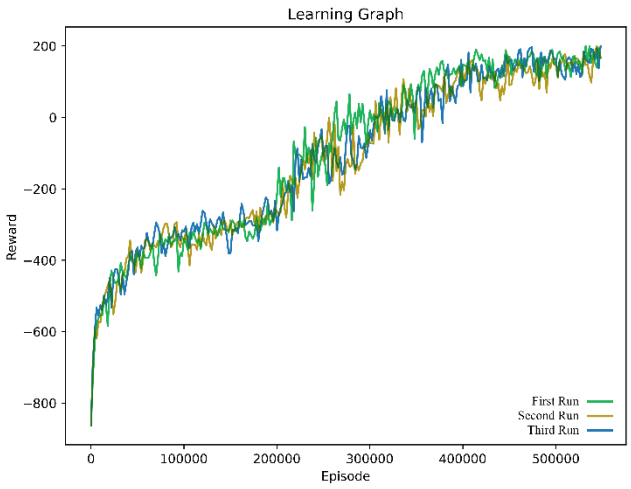
Figure 5. Navigation results of utilizing CCSS and CUSS with three robots: (a) scenario 1; (b) scenario 2; (c) scenario 3.

The paths shown here are the exact paths we get while exploiting the learned model; however, we could not include all the robot turns. Sometimes, the robots select the do-nothing action or turn-left or turn-right actions in each waypoint before continuing to the next waypoint. For more readability, we omitted them. Experiments show that the innate nature of the Bellman equation was finally able to find an acceptable short path, sometimes with negligible deviations.

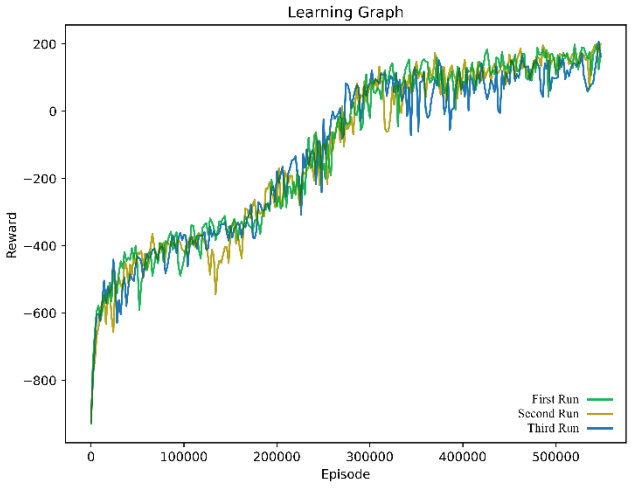
It should also be noted that it is sometimes possible to remove minor deviations from short path by extending the learning duration, however, overextending the learning process makes the robots forget the previously found short paths. This problem is intensified in a multi-agent environment. Excessive exploration can disrupt the other agents, making the exploring agent's learning task more difficult [33]. It should be kept in mind that not all the deviations are due to insufficient learning, e.g., robot 3's turn at the beginning of the path in Figure 5c. This turn is for avoiding collisions.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access



(a)



(b)

Figure 6. Learning graph: (a) learning graph of CUSS (b) learning graph of CCSS for scenario 2; both for three different runs of learning stage;

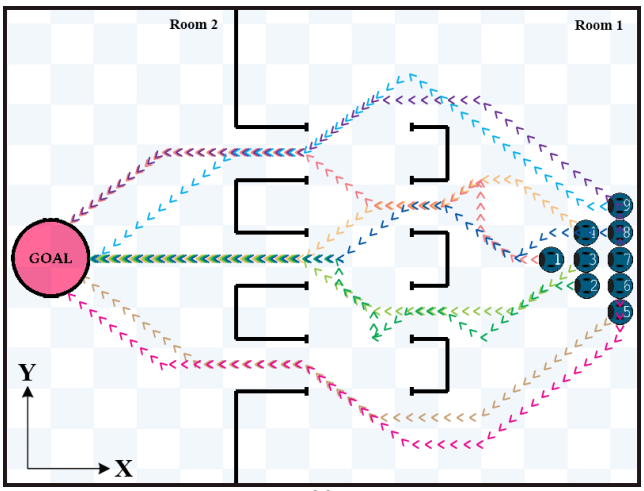
D. LEARNING GRAPHS OF CUSS AND CCSS

Figures 6a and 6b illustrate three different runs of learning stage and their corresponding learning graph for CUSS and CCSS, respectively, for the scenario defined in figure 6b. The horizontal axis of the graph shows the episode number, and the vertical axis shows the reward attained at the respective episode. Graphs are drawn according to equation 2. For drawing these graphs after every 200 episodes, the algorithm is tweaked to only work in exploitation mode for 50 episodes to measure the quality of the trained model so far. The program starts with a high percentage of exploration at first, and then it is reduced step by step as the learning progresses. Each point on the graph is the result of averaging of total reward in these 50 episodes. It can be seen that the learning graphs of both CUSS and CCSS are very similar.

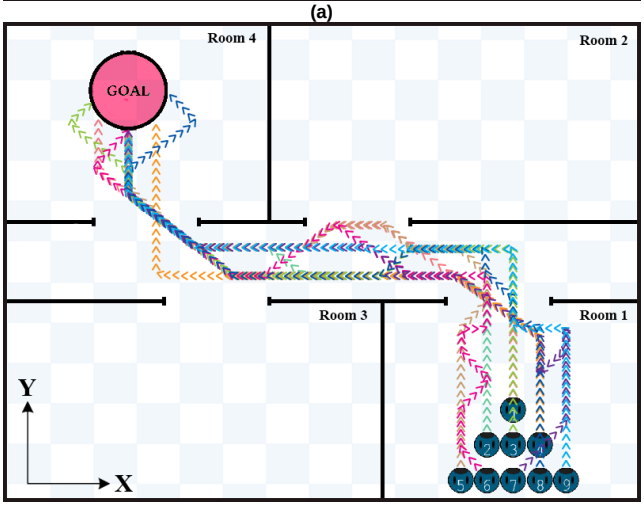
Both methods have nearly identical state spaces, but their storing strategies vary from each other. So, we do not expect their learning duration to be different. Analyzing the graphs in Figure 6, both methods have converged to a reasonable solution after about "400,000" episodes. We should keep in mind that these methods are to some degree stochastic, meaning that the same problem with the same method might find the solution a little faster in one run in comparison with another. However, learning cannot be finished permanently at an early stage or always at the same stage, because, for creating an acceptable model, most of the states are needed to be explored which is encouraged by putting default zero values into Q-Table.

E. NAVIGATION IN SWARMS WITH MORE THAN THREE ROBOTS

In the next set of experiments, we conducted tests for our swarm navigation algorithm using CCSS and (GCC). For this, we utilized the same three scenarios from the previous experiment. In this set of experiments, we chose a swarm consisting of nine and eighteen robots and the results are shared in table 3. For the purpose of demonstration with an image, we chose the test with nine robots because the overlapping trails were more traceable with the eye. The figures below show the sample results:



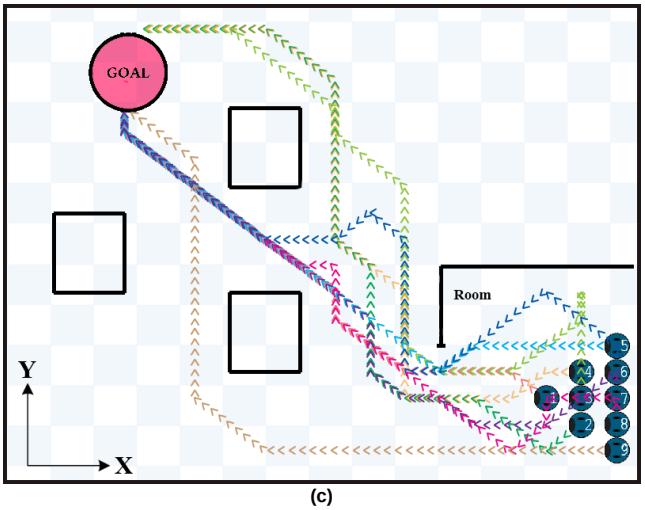
(a)



(b)

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access



(c)

Figure 7. **Navigation results of utilizing GCC with nine robots: (a) scenario 1; (b) scenario 2; (c) scenario 3.**

Despite the three robot experiments in these scenarios, there was not enough space for all the robots around the goal point, so we removed the robots as they reached the goal. Therefore, in 18 robot experiments, several robots might reach the goal from the same point without colliding.

Table 3. **Performance results of the tests done with different numbers of robots**

Scenario No.	Number of robots	Time(sec)	Avg. angular velocity(degree	Avg. distance traveled (cm)	Min. Distance traveled(cm)	Max. Distance traveled(cm)
1	3	12	24	531	508	554
	9	17	18	566	500	645
	18	20	25	605	481	545
2	3	17	20	793	612	1025
	9	21	21	697	579	787
	18	25	23	743	558	730
3	3	14	21	650	508	783
	9	25	19	673	525	783
	18	24	22	716	466	866

As it can be seen from Table 3, generated solutions for all scenarios are in a reasonable and near-optimum range. In most cases, the tests conducted with more robots requires more time to be completed. This is partially due to more dynamic obstacles that cause unexpected deviations from the actual short path. In other words, the presence of more robots makes the environments more indeterministic, and thus, future state prediction becomes more burdensome, especially for robots behind the front line; however, looking at the average values, we can say that it is not a noticeable problem. This is mainly because the number of generated optimal and non-optimal paths are almost equal, dropping the average within the same range as in three robot cases. What is more, analyzing the minimum and maximum distance traveled, adding more robots helped the model learn a better solution, and also, in most cases, the worst case is also better. This is primarily because of more test cases that are available for these test scenarios.

V. CONCLUSION

Q-Learning is an adaptive model-free reinforcement learning algorithm that extracts policies without modeling the environment or needing the model of the environment, making it a more flexible and straightforward algorithm in comparison with model-based algorithms. However, it necessitates a considerable amount of memory, making it unfeasible for large tasks. This paper introduced a new method that compresses the state space size and lowers the memory cost (CCSS) for making it possible to solve more complicated problems, especially the problem of steering decentralized multi-agent systems, relying on Reinforcement Learning. The proposed model achieves high success in multi-agent navigation systems with three agents. Next, another method introduced and employed for reducing the sensory data by utilizing an object prioritization algorithm; utilizing this method integrated on top of the first method (GCC) made navigation possible for more than three agents. The experimental results present the memory reduction percentage, navigation results, and performance measures for experiments, conducted for different complex scenarios. The results reveal the significant memory reduction for multi-agent tasks utilizing the proposed method. Furthermore, conducted tests have revealed that these techniques are successful in solving the navigation problem and identifying near-optimal solutions. It can also be shown from the performance results that increasing the size of the swarm does not affect the average performance. The results of this research also encourage the authors to further investigate proposed methods using deep-Q-learning algorithm specially the “GCC” that potentially can reduce the learning duration and input feature size while increasing the swarm count.

VI. FUTURE WORK

Q-learning has the advantage of not requiring the model of the environment enabling us to design controllers for complicated tasks such as decentralized multi-agent navigation problem, however, there were still many problems that are needed to be addressed in this paper. We can name some of addressed issues as follows: Sharing data between agents in learning phase mainly in order to accelerate the learning, designing a model for coping with decentralized multi-agent navigation problem, coping with limited memory problem and discretizing the features while not exceeding the memory size. For future work we suggest applying the designed system on real robots. Another possibility is applying this method on a different multi-agent task rather than navigation. Additionally,

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

the effects of feature stacking introduced in this paper can be further investigated on a deep Q-learning system.

REFERENCES

[1] G. Flierl, D. Grünbaum, S. Levins, and D. Olson, "From individuals to aggregations: the interplay between behavior and physics," *J. Theor. Biol.*, vol. 196, no. 4, pp. 397–454, 1999.

[2] N. W. F. Bode, A. J. Wood, and D. W. Franks, "Social networks and models for collective motion in animals," *Behav. Ecol. Sociobiol.*, vol. 65, no. 2, pp. 117–130, 2011.

[3] E. Bahgeçi and E. Sahin, "Evolving aggregation behaviors for swarm robotic systems: A systematic case study," in *Proceedings 2005 IEEE Swarm Intelligence Symposium, 2005. SIS 2005.*, 2005, pp. 333–340.

[4] M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo, "Swarm robotics: a review from the swarm engineering perspective," *Swarm Intell.*, vol. 7, no. 1, pp. 1–41, Mar. 2013, doi: 10.1007/s11721-012-0075-2.

[5] M. Chamanbaz *et al.*, "Swarm-enabling technology for multi-robot systems," *Front. Robot. AI*, vol. 4, p. 12, 2017.

[6] S. Camazine, J.-L. Deneubourg, N. R. Franks, J. Sneyd, E. Bonabeau, and G. Theraula, *Self-organization in biological systems*. Princeton university press, 2003.

[7] Y. Mohan and S. G. Ponnambalam, "An extensive review of research in swarm robotics," in *2009 World Congress on Nature & Biologically Inspired Computing (NaBIC)*, 2009, pp. 140–145.

[8] V. Costa, M. Duarte, T. Rodrigues, S. M. Oliveira, and A. L. Christensen, "Design and development of an inexpensive aquatic swarm robotics system," in *Oceans 2016-Shanghai*, 2016, pp. 1–7.

[9] M. Senanayake, I. Senthooran, J. C. Barca, H. Chung, J. Kamruzzaman, and M. Murshed, "Search and tracking algorithms for swarms of robots: A survey," *Rob. Auton. Syst.*, vol. 75, pp. 422–434, 2016.

[10] Y. Tan and Z. Zheng, "Research advance in swarm robotics," *Def. Technol.*, vol. 9, no. 1, pp. 18–39, 2013.

[11] A. V Savkin, C. Wang, A. Baranzadeh, Z. Xi, and H. T. Nguyen, "Distributed formation building algorithms for groups of wheeled mobile robots," *Rob. Auton. Syst.*, vol. 75, pp. 463–474, 2016.

[12] L. Matignon, G. J. Laurent, and N. Le Fort-Piat, "Hysteretic q-learning: an algorithm for decentralized reinforcement learning in cooperative multi-agent teams," in *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2007, pp. 64–69.

[13] T. M. Cheng and A. V Savkin, "Decentralized control for mobile robotic sensor network self-deployment: Barrier and sweep coverage problems," *Robotica*, vol. 29, no. 2, p. 283, 2011.

[14] M. D. et al. Al., "Swarm robotics," *Scholarpedia*, 2014. .

[15] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[16] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[17] D. P. Bertsekas, "Dynamic Programming and Optimal Control, volume 1 of Optimization and Computation Series," *Athena Sci. Belmont, MA, USA, 3rd Ed. Cited*, p. 2, 2005.

[18] C. Jin, Z. Allen-Zhu, S. Bubeck, and M. I. Jordan, "Is Q-learning provably efficient?," in *Advances in Neural Information Processing Systems*, 2018, pp. 4863–4873.

[19] C. J. C. H. Watkins and P. Dayan, "Q-Learning," *Mach. Learn.*, vol. 8, no. 3–4, pp. 279–292, 1992.

[20] V. S. Borkar and S. P. Meyn, "The ODE method for convergence of stochastic approximation and reinforcement learning," *SIAM J. Control Optim.*, vol. 38, no. 2, pp. 447–469, 2000.

[21] T. Jaakkola, M. I. Jordan, and S. P. Singh, "Convergence of stochastic iterative dynamic programming algorithms," in *Advances in neural information processing systems*, 1994, pp. 703–710.

[22] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, 2015, pp. 1889–1897.

[23] M. Deisenroth and C. E. Rasmussen, "PILCO: A model-based and data-efficient approach to policy search," in *Proceedings of the 28th International Conference on machine learning (ICML-11)*, 2011, pp. 465–472.

[24] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement Learning: A Survey," *J. Artif. Intell. Res.*, vol. 4, pp. 237–285, May 1996, doi: 10.1613/jair.301.

[25] G. Weiss, *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press, 1999.

[26] R. S. Sutton and A. G. Barto, "Reinforcement learning: an introduction MIT Press," *Cambridge, MA*, vol. 22447, 1998.

[27] V. Cherkassky and F. M. Mulier, *Learning from data: concepts, theory, and methods*. John Wiley & Sons, 2007.

[28] G. E. Hinton and T. J. Sejnowski, *Unsupervised learning: foundations of neural computation*. MIT press, 1999.

[29] P. Stone and M. Veloso, "Multiagent systems: A survey from a machine learning perspective,"

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

Auton. Robots, vol. 8, no. 3, pp. 345–383, 2000.

[30] M. S. Güzel, V. B. Ajabshir, P. Nattharith, E. C. Gezer, and S. Can, "A novel framework for multi-agent systems using a decentralized strategy," *Robotica*, vol. 37, no. 4, pp. 691–707, 2019.

[31] H. Celikkanat and E. Şahin, "Steering self-organized robot flocks through externally guided individuals," *Neural Comput. Appl.*, vol. 19, no. 6, pp. 849–865, 2010.

[32] E. Ferrante, A. E. Turgut, C. Huepe, A. Stranieri, C. Pinciroli, and M. Dorigo, "Self-organized flocking with a mobile robot swarm: a novel motion control method," *Adapt. Behav.*, vol. 20, no. 6, pp. 460–477, 2012.

[33] L. Buşoniu, R. Babuška, and B. De Schutter, "Multi-agent reinforcement learning: An overview," *Innov. multi-agent Syst. Appl.*, pp. 183–221, 2010.

[34] P. Long, T. Fanl, X. Liao, W. Liu, H. Zhang, and J. Pan, "Towards optimally decentralized multi-robot collision avoidance via deep reinforcement learning," *Proc. - IEEE Int. Conf. Robot. Autom.*, pp. 6252–6259, 2018, doi: 10.1109/ICRA.2018.8461113.

[35] L. Amado and F. Meneguzzi, "Q-Table compression for reinforcement learning," *Knowl. Eng. Rev.*, vol. 33, pp. 1–21, 2018, doi: 10.1017/s0269888918000280.

[36] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 1987, pp. 25–34.

[37] Q. Li, F. Gama, A. Ribeiro, and A. Prorok, "Graph neural networks for decentralized multi-robot path planning," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 11785–11792.

[38] I. Giannoccaro and P. Pontrandolfo, "Inventory management in supply chains: a reinforcement learning approach," *Int. J. Prod. Econ.*, vol. 78, no. 2, pp. 153–161, 2002.

[39] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *Int. J. Rob. Res.*, vol. 32, no. 11, pp. 1238–1274, 2013.

[40] J. D. R. Millán, D. Posenato, and E. Dedieu, "Continuous-action Q-learning," *Mach. Learn.*, vol. 49, no. 2–3, pp. 247–265, 2002.

[41] J. Hu and M. P. Wellman, "Nash Q-learning for general-sum stochastic games," *J. Mach. Learn. Res.*, vol. 4, no. Nov, pp. 1039–1069, 2003.

[42] P. G. Patel, N. Carver, and S. Rahimi, "Tuning computer gaming agents using q-learning," in *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2011, pp. 581–588.

[43] H. Kayakoku, M. S. Guzel, E. Bostanci, I. T. Medeni, and D. Mishra, "A Novel Behavioral Strategy for RoboCode Platform Based on Deep Q-Learning," *Complexity*, vol. 2021, 2021.

[44] H. Kumar, A. Koppel, and A. Ribeiro, "On the sample complexity of actor-critic method for reinforcement learning with function approximation," *arXiv Prepr. arXiv1910.08412*, 2019.

[45] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 1096–1103.

[46] C. Gehring, L. P. Kaelbling, and T. Lozano-Perez, "Adaptable replanning with compressed linear action models for learning from demonstrations," *Proc. 2nd Conf. Robot Learn.*, vol. 87, no. CoRL, pp. 432–442, 2018, [Online]. Available: http://proceedings.mlr.press/v87/gehring18a.html.

[47] A. Mostaani, T. X. Vu, S. Chatzinotas, and B. Ottersten, "Task-Based Information Compression for Multi-Agent Communication Problems with Channel Rate Constraints," vol. 2022, pp. 1–15, 2020, [Online]. Available: http://arxiv.org/abs/2005.14220.

[48] T. Rajalakshmi and S. Sathappan, "Incorporation of Reinforcement Learning and Compression of Deeprt for Accurate and Fast Classification of Cps Data," vol. 7, no. 3, pp. 1366–1375, 2020.

[49] C. Kim, S. Kang, D. Shin, S. Choi, Y. Kim, and H.-J. Yoo, "A 2.1 TFLOPS/W mobile deep RL accelerator with transposable PE array and experience compression," in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, 2019, pp. 136–138.

[50] U. Jaidee and H. Muñoz-Avila, "Classq-l: A q-learning algorithm for adversarial real-time strategy games," 2012.

[51] T. Rashid, M. Samvelyan, C. Schroeder, G. Farquhar, J. Foerster, and S. Whiteson, "Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning," in *International Conference on Machine Learning*, 2018, pp. 4295–4304.

[52] J. N. Tsitsiklis, "Asynchronous stochastic approximation and Q-learning," *Mach. Learn.*, vol. 16, no. 3, pp. 185–202, 1994.

[53] P. Shinners, "Pygame," *Dostupné z http//pygame. org/[Online*, 2011.

[54] S. Koenig and R. G. Simmons, "Complexity analysis of real-time reinforcement learning," in *AAAI*, 1993, pp. 99–107.

[55] R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovász, and C. Rackoff, "Random walks, universal traversal

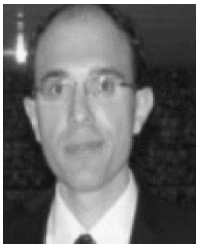
This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3163393, IEEE Access

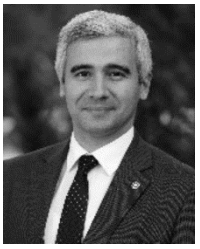
sequences, and the complexity of maze problems," in *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, 1979, pp. 218–223.



VAHID BABAEI AJABSHIR received the B.S. degree in computer engineering from Nabi Akram University, Tabriz, Iran, in 2008 and the M.S. degree in computer engineering from Ankara University, Ankara, Turkey, in 2015. He is currently pursuing the Ph.D. degree in computer engineering at Ankara University, Ankara, Turkey. He is currently working as a senior programmer in Setur tourism company in Turkey. His research interests include Network-on-chip, optimization algorithms, machine learning and robotics.



MEHMET SERDAR GUZEL received the B.Sc. degree from Computer Engineering Department of Baskent University in 2004. He also got his M.Sc. degree from the Computer Engineering Department, Ankara University, Turkey, and the Ph.D. degree from the Mechanical and Systems Engineering Department, Newcastle University, U.K., in 2012. He is currently working as an Associate Professor in Computer Engineering Department of Ankara University. His main research interests include machine learning, swarm intelligence, computer vision, and robotics. He has contributed several international projects and published many academic papers, regarding those fields.



ERKAN BOSTANCI received the B.Sc. degree and the M.Sc. degree in real-time battlefield simulation from the Computer Engineering Department, Ankara University, Turkey, in 2007 and 2009, respectively, and the Ph.D. degree from the School of Computer Science and Electronic Engineering, University of Essex, U.K., in 2014. He joined the Computer Engineering Department, Ankara University, as a Research Assistant. His research interests include different yet closely related aspects of computer science from image processing, computer vision and graphics to artificial intelligence and fuzzy logic, as well as mathematical modeling and statistical analysis. He has been involved in technical committees for several conferences and has been organizing an international conference for several years as well as acting as a reviewer for various journals.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/