Expert Systems With Applications 135 (2019) 212–218

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Deep reinforcement learning applied to the *k*-server problem

Ramon Augusto Sousa Lins*, Adrião Duarte Neto Dória, Jorge Dantas de Melo

Federal University of Rio Grande do Norte, Department of Computer Engineering and automation, Rio Grande do Norte, Natal, Brazil

ARTICLE INFO

Article history:
Received 11 March 2019
Revised 22 May 2019
Accepted 6 June 2019
Available online 7 June 2019

Keywords:
Deep reinforcement learning
Online problem
The *k*-server problem
Combinatorial optimization
Competitive location

ABSTRACT

The reinforcement learning paradigm has been shown to be an effective approach in solving the *k*-server problem. However, this approach is based on the Q-learning algorithm, being subjected to the curse of dimensionality problem, since the action-value function (Q-function) grows exponentially with the increase in the number of states and actions. In this work, a new algorithm based on the deep reinforcement learning paradigm is proposed. For this, the Q-function is defined by a multilayer perceptron neural network that extracts the information of the environment from images that encode the dynamics of the problem. The applicability of the proposed algorithm is illustrated in a case study in which different nodes and servers problem configurations are considered. The agents behavior is analyzed during the training phase and its efficiency is evaluated from performance tests that quantify the quality of the generated server displacement policies. The results obtained provide a new algorithm promising view as an alternative solution to the *k*-server problem.

1. Introduction

In online computation a computer algorithm must decide how to act from the currently available input, without being aware of the entire set of inputs (Allan & El-Yaniv, 1998). Several current issues can be inserted in this type of problem, for example, given the price of Bitcoin, must the crypto currency be sold or bought? How to assign and reassign processes in a parallel processing so that there is load balancing on all processors? How to move a facility into an online transportation service minimizing the total cost involved in the process? Problems like these are complex since after a decision is made, it cannot be revoked, influencing the solution as a whole.

Proposed by Manasse, McGeoch, and Sleator (1988) the *k*-server problem (KSP) is the problem of moving k servers over n nodes on a graph (or metric space) in order to satisfy the requests that appear online (sequentially) minimizing some cost function determined by the problem. Its conceptual simplicity contrasts with its complexity that grows exponentially with the increase in number of nodes and servers, being perhaps the most influential online problem, serving as a propeller for the development of new algorithms (Bansal, Buchbinder, Madry, & Naor, 2015; Gupta, Kamali, & López-Ortiz, 2016).

Initially, the works related to the *k*-server problem focused on special metric spaces such as the uniform metric space, which corresponds to the paging problem in which high-performance algorithms are known (Sleator & Tarjan, 1985). For general metrics, Koutsoupias (2009) proposed the work function algorithm which is suitable for any metric space, obtaining results close to optimal solutions. Despite this, the results obtained by these algorithms are deterministic, and there is a great interest in using randomized algorithms for the *k*-server problem, since these algorithms tend to have a better performance than the deterministic ones (Bansal et al., 2015).

Placed in a context of competitive facility location problem, considering that it serves more than one point (dynamically), in general, the decisions on the location of facilities (servers), include high fixed costs and a long-term return analysis, having a great economic impact due to optimization techniques used, an attractive for its solution. Since the decision depends only on the information available at the time, this problem can be modeled as a Markov decision process and consequently defined as a Reinforcement Learning (RL) problem.

In an RL paradigm an agent observes the state $s_t \in S$ (set of states) of an environment and takes an action $a_t \in A$ (set of actions) based on new choices (exploration) or experiences already acquired (intensification) and receives a reward r_{t+1} at each discrete time step t subject to $t \leq T$, where T is the final time step. The aim is to find a policy $\pi = P(s_t|a_t)$ that maximizes (or minimizes) the expected return signal denoted as $R_t = \sum_{k=0}^{\infty} \gamma^k r_{k+t+1}$, where $\gamma \in [0, 1]$ is the discount factor, a parameter that defines the

* Corresponding author.
E-mail addresses: ramon.lins@dca.ufrn.br (R.A.S. Lins), adriao@dca.ufrn.br (A.D.N. Dória), jorge@dca.ufrn.br (J.D.d. Melo).

https://doi.org/10.1016/j.eswa.2019.06.015
0957-4174/

importance that the reward has in long term calculating the expected total return. In simple terms, solving reinforcement learning problems consists of finding good policies that provide the highest (or lowest) expected future reward.

There are different methods to solving RL problems such as dynamic programming, Monte Carlo, temporal-difference, semi-gradient, policy gradients, actor-critic, etc. The majority of these algorithms involve calculating an action-value function (or a value function). Solving them consists of defining an optimal policy π^* from $Q^*(s,a) = \mathbb{E}[r_{t+1} + \gamma \max_{a'} Q^*(s',a')|s_t = s, a_t = a]$ (Bellman optimality equation), where a' represents the possible actions that can be taken from a new state s_{t+1} denoted as s' (Sutton & Barto, 2018). From the calculated Q^*, the optimal policy is given by $\pi^*(s) = \text{argmax}_a Q^*(s,a)$.

From the authors knowledge, the first time an algorithm based on intelligent optimization techniques was used as a solution to the *k*-server problem was in the work proposed by Junior, Neto, and Melo (2005), who used the reinforcement learning paradigm through the Q-learning algorithm, a temporal-difference method that calculates the Q-function directly from the data (model-free) and independently of the policy (off-policy) used. The algorithm was used in the solution of small instances and compared to influential algorithms of the literature, Harmonic and Work Function (Koutsoupias, 2009). To adapt the algorithm for large instances a hierarchical approach was used in which nodes and servers were separated into clusters. The local policy obtained in each grouping was combined and used in the formation of a global policy. Continuing this approach, Costa, Padilha, Melo, and Neto (2016) paralleled the algorithm and defined a load-balancing parameter to improve clustering. The results obtained showed their feasibility in practical dimensions applications such as some approached by Rudec, Baumgartner, and Manger (2013) and Bertsimas, Jaillet, and Korolko (2019).

Despite this, the hierarchical approach still fall into the curse of dimensionality problem since its storage structure (Q table), algorithm basis, grows exponentially as the number of states and actions increases. From the tabular approach point of view the *k*-server problem presents a storage structure defined as $C_{n,k}.n.k$, where the first term represents the combination of n terms taken k by k from valid states and $n.k$ the total number of possible actions per state. Thus, computing the action-value function for larger instances becomes an extremely costly computational process even in a hierarchical approach, perhaps having as the main limiting factor the quantity of servers, given that the space complexity of the algorithm is $\mathcal{O}(n^k)$ (Costa et al., 2016), which can quickly take the problem to an explosion of dimensionality.

To deal with this type of problem, learning methods based on function approximation can be used as an alternative to the traditional approach. In this work, the *k*-server problem is defined as deep reinforcement learning (Mnih et al., 2015) task through the use of the Q-learning algorithm in conjunction with a Multilayer Perceptron (MLP) neural network. A simple way to integrate a neural network into Q-learning is to use the gradient descent algorithm (Lin, 1993; Riedmiller, 2005; Tesauro, 1995). Currently, the use of deep learning techniques in conjunction with the reinforcement learning paradigm is driven by recent advances (Mnih et al., 2015; Silver et al., 2017), as well as its applicability in several practice areas (Afify et al., 2019; Araque, Corcuera-Platas, Sanchez-Rada, & Iglesias, 2017; Bello, Pham, Le, Norouzi, & Bengio, 2016; Mao, Alizadeh, Menache, & Kandula, 2016).

To convert the Q-learning into a gradient-based method, Q-function updating is replaced by updating the synaptic weights vector w using the approximated action-value function, denoted as $Q(s, a; w)$ or $\tilde{Q}(s,a)$ (implicit parameter) and its gradient calculus. In order to adapt the *k*-server problem to the new proposed algorithm, the dynamics of the problem is represented by images, that encode the states of environment, and rewards that express the minimum cost paths related with actions taken, providing together enough information for the neural network to define a satisfactory displacement policy. With the problem modeled as a visual task, the Q-function, which refers to all possibilities, is replaced by the weight matrix whose solution is generalized and depends on the number of neural network parameters.

The main contributions of this paper are:

- Propose a new algorithm less susceptible to the curse of dimensionality since the decision-making process is done in a generalized way, thus presenting greater scalability for larger instances problem focusing on the increasing of the number of servers.
- Transform the *k*-server model in a visual task problem suitable for approximation methods.
- Validate the proposed algorithm comparing its performance with Q-learning, an algorithm with strong convergence proof (Watkins, 1989); and greedy algorithm whose solution demonstrates reasonable performance (Rudec et al., 2013).
- Provide insight on how the visual task problem can be addressed in a real-life practice situation.
- Show the robustness of the algorithm to randomness, an intrinsic property to the *k*-server problem, an attractive to area, since there is not a good understanding of the problem when randomness is allowed (Bansal et al., 2015).

This article is structured as follows, Section 2 gives a brief formal definition of the *k*-server problem that will be useful in understanding the proposed methodology; in Section 3 the proposed algorithm is formally presented; then in Section 4 its performance is compared with other algorithms, and finally in Section 5 the final considerations are made.

2. The *k*-server problem

Formalizing the problem, consider that $G = (N, E)$ represents a weighted, non-directional and connected graph, where $n = |N|$ is the number of nodes and E, the set of edges that interconnect the nodes. Each edge $\varepsilon \in E$ is associated with a non-negative and symmetric weight. We use $\varepsilon(u, v)$ to represent the weight of nodes $(u, v) \in N$ joined by an edge. Let's use $k^{(t)}$ to represent the distribution of servers (homogeneous) on the graph and k to define the number of servers that must satisfy a sequence of requests that appear online on the nodes along time step t. Assuming that a new request appears only after the current request has been serviced, the *k*-server problem can be defined in the following way:

- The distribution of k servers on the graph is defined as $k^{(t)} = \{k_{1,i}^{(t)}, k_{2,i}^{(t)}, \ldots, k_{l,i}^{(t)}\}$ where $l = \{1, 2, \ldots, k\}$ specifies the server in distribution, and $i \in N$ determine the node in which the server is located. The term $k_{l,i}^{(t)}$ means that in time step t the server l is located at node i of G.
- The sequence of requests $\Sigma = \{\sigma_1^{(0)}, \sigma_2^{(1)}, \ldots, \sigma_j^{(t)}\}$ represents the request $\sigma_j^{(t)}$ that appear at node $j \in N$ along time step $t = \{0, 1, \ldots, T-1\}$, where T is the total number of requests in a sequence.
- The displacement of $k_{l,i}^{(t)}$ to service $\sigma_j^{(t)}$ from a distribution $k^{(t)} = \{k_{1,i}^{(t)}, k_{2,i}^{(t)}, \ldots, k_{l,i}^{(t)}\}$ leads to a new distribution defined as $k^{(t+1)} = \{k_{1,i}^{(t+1)}, k_{2,i}^{(t+1)}, \ldots, k_{l,j}^{(t+1)}\}$.

For a better comprehension the *k*-server problem is demonstrated in an example with $n = 4$ and $k = 2$ (see Fig. 1). The initial configuration ($t = 0$) should be known, for this example the server $l = 1$ is located at node $i = 2$, the server $l = 2$ at $i = 3$ and the request appears at $j = 4$ characterizing the distribution of servers as $k^{(0)} = \{k_{1,2}^{(0)}, k_{2,3}^{(0)}\}$ and request as $\sigma_4^{(0)}$.

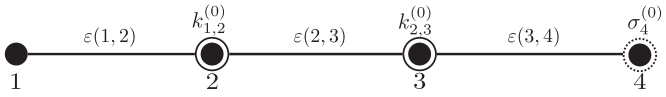


Fig. 1. Initial distribution of servers and request in the k-server problem.

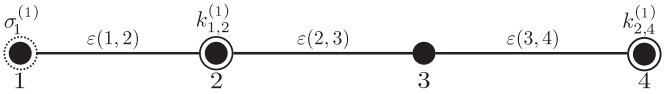


Fig. 2. New distribution after the server is moved and a new request arrives.

If $k_{2,3}^{(0)}$ is displaced to service $\sigma_4^{(0)}$ a new distribution denoted as $k^{(1)} = \{k_{1,2}^{(1)}, k_{2,4}^{(1)}\}$ is characterized and a new request appears at some node, in this case $\sigma_1^{(1)}$ (see Fig. 2).

The server displacement is subjected to the optimization of a cost function according to the problem. A common problem is to minimize the total cost involved in the displacement of the k servers. For the sake of simplicity, the k-server problem is modified so that only one server occupies a node and is moved at a time. Although its formal definition allows multiple servers to be allocated at the same node, this situation is not necessary, so that this modification does not change the cost for any computed solution (Allan & El-Yaniv, 1998).

3. Proposed approach

In this section the k-server problem is modeled as a visual computing problem in a reinforcement learning approach with approximate solution by a multilayer perceptron network.

3.1. Overview

In a traditional reinforcement learning approach, the agent observes the state s_t of an environment, takes an action a_t and receives a reward r_{t+1}. In a deep reinforcement learning approach however, the agent perceives the internal dynamics of the environment from observations $o_t \in \mathcal{O}$ of unknown semantics as shown in Fig. 3.

3.2. Observations

The internal state s_t of the environment is defined by an image $o_t \in \mathbb{R}_+{}^2$ that encodes the distribution of servers $k^{(t)}$ and the request $\sigma_j^{(t)}$ at each step of time t. The number of pixels p must satisfy the condition that $n \leq p$, so that all possible distributions can be encoded. The encoding is done by means of the pixels intensity levels defined in an interval $[L_{min} \leq L \leq L_{max}]$ (grayscale), where white ($L_{max} = 1$) is used to define the servers, medium gray ($L = 0.5$) the request and black ($L_{min} = 0$) the non occupied or surplus pixels. After this, the image is converted into a feature vector $x_t \in \mathbb{R}^{n\times 1}$ whose request numerical value is converted to a negative value and the surplus pixels are removed. This representation is demonstrated in two examples in which the servers are defined as $k^{(0)} = \{k_{1,2}^{(0)}, k_{2,3}^{(0)}\}$ and the request as $\sigma_4^{(0)}$ (see Fig. 4). The dotted edges are used to illustrate that the encoding is not tied to the edge connections, it's a representation that converts the information about the nodes occupied by the servers and by the request into symbols used as input units of the neural network.

Although grayscale representation is used, this modeling is not restricted to this case, this option is made to reduce the computational cost involved in processing colored images.

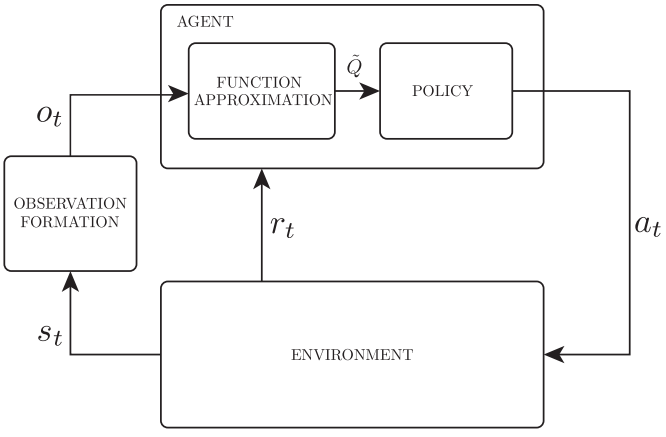
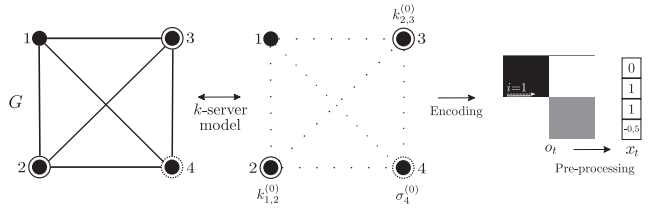
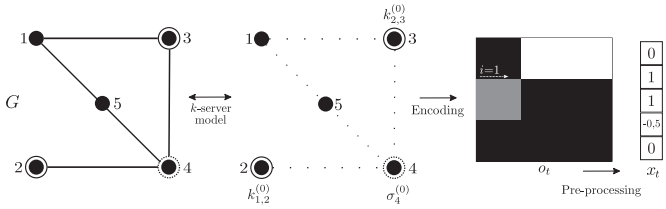


Fig. 3. The agent-environment interaction in a deep reinforcement learning approach.



(a) Problem example with 4 nodes and 2 servers.



(b) Another example with 5 nodes and 2 servers.

Fig. 4. Observation representation.

3.3. Actions and future observations

Once the server distribution and request have been defined, one of the servers $k_{l,i}^{(t)}$ must be moved to answer the request $\sigma_j^{(t)}$ characterizing an action a_t. In this situation the number of allowed actions is equal to k, the number of servers, where $a_t \in \{1, 2, \ldots, k\}$. The distribution of servers and request, associated with an action taken a_{k_l} characterizes an instance of the problem. After the action is performed a new distribution of servers $k^{(t+1)}$ is characterized and another situation happens when a request is in the eminence of happening, in this case the request $\sigma_j^{(t+1)}$ can appear in one of the non occupied nodes characterizing a $(n-k)\times k$ number of possible actions a_{t+1} that defines the future observations x_{t+1}.

3.4. Multilayer artificial neural network

The network consists of a fully connected hidden layer with a sigmoid activation function and a fully connected linear output layer coupled to the number of servers. The linear output is used to represent the entire range of possible real output numbers (see Fig. 5). Since we are using the cost involved in the displacement of the servers as reward, even if the limits of the output are not known, there is a guarantee that their values will be represented.

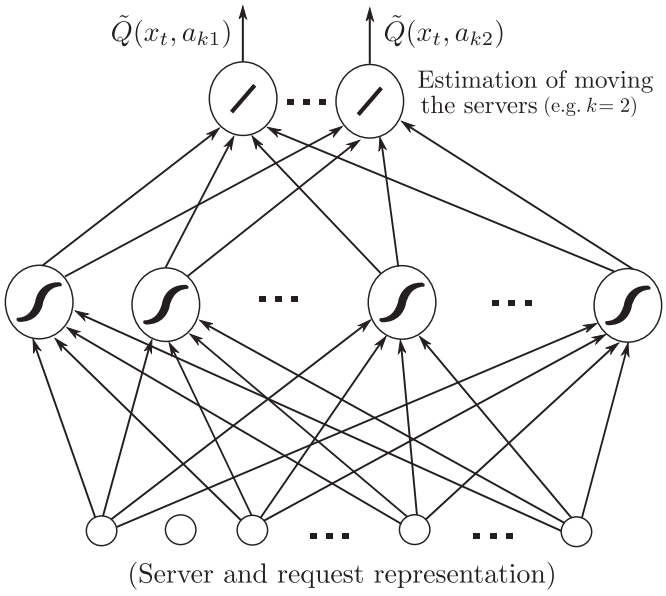


Fig. 5. Neural network architecture.

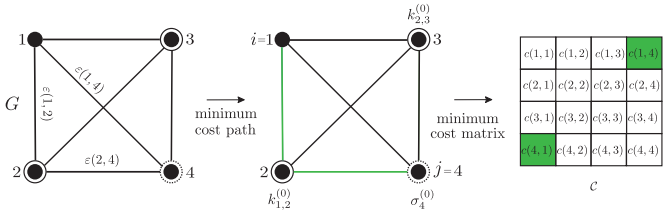


Fig. 6. Reward function.

3.5. Reward and return

For each taken action the environment returns a reward r_{t+1} directly proportional to a cost defined by the minimum cost path $c(i, j)$ between a node i (source) and a node j (destiny) that makes up a minimum cost matrix $\mathcal{C}$ (see Fig. 6) defined as:

$$\mathcal{C} = \begin{cases} c(i,j) & \text{, if } i \neq j \\ 0 & \text{, if } i = j \end{cases} \tag{1}$$

Thus, actions taken through a policy π over time should minimize the expected total return denoted by $min\ \mathbb{E}[R]$, in other words, the agent must learn how to manage the server displacement so that the total cost is as small as possible.

3.6. Agent-environment interaction

The environment internal state is perceived by the agent which, after taking an action, receives the reward (see Fig. 7). The red arrow indicates the chosen server ($l = 2$) to service the request ($j = 4$) and the green edge indicates the minimum cost of moving the server.

3.7. Algorithm

From G the k-server environment is encoded in o_t and converted into x_t. The input vector propagates forward passing through the neurons of the network approximating the Q-function. Next, the policy π maps the action a_{k_l} through the ϵ-greedy algorithm defined by:

$$a_{k_l} = \begin{cases} \underset{a_t}{\text{argmin}} Q(x_t, a_t; w_t) & \text{wit probability } 1-\epsilon \\ \text{random (uniform)} & \text{with probability } \epsilon \end{cases} \tag{2}$$

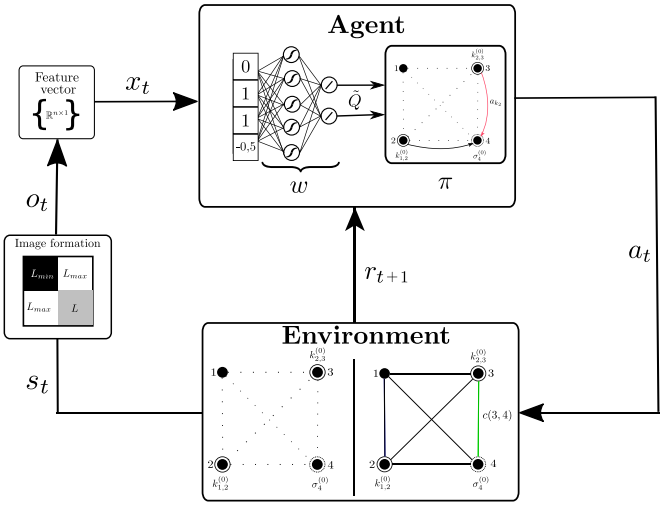


Fig. 7. The k-server problem as a deep reinforcement learning task. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

where ϵ represents the factor of randomness that guarantees the balance of the strategy of exploration-intensification of new solutions, increasing the network generalization power. Once the action is defined, the reward r_{t+1} is received and next state s_{t+1} represented by x_{t+1} is observed. Therefore, the error signal defined as:

$$\Delta\tilde{Q}(x_t, a_t) = \overbrace{\left(c(i,j) + \gamma \min_{a_{t+1}} \underbrace{Q(x_{t+1}, a_{t+1}; w)}_{\tilde{Q}^{future}}\right)}^{\tilde{Q}^{target}} - Q(x_t, a_t; w) \tag{3}$$

must be minimized so that the predicted Q-value, denoted by $Q(x_t, a_t; w)$, is approximated to the target Q-value ($\tilde{Q}^{target}$) according to the Huber objective function (Huber et al., 1964) defined as:

$$L_\delta(\Delta\tilde{Q}) = \begin{cases} \frac{1}{2}(\tilde{Q}^{target} - Q(x_t, a_t; w))^2 & |\Delta\tilde{Q}| \leq \delta \\ \delta\left(|\tilde{Q}^{target} - Q(x_t, a_t; w)| - \frac{1}{2}.\delta\right) & \text{otherwise} \end{cases} \tag{4}$$

where $\delta = 1$ represents the behavior threshold of the objective function. For values below δ, the objective function behaves as a quadratic function, while for larger values it behaves as a linear function. This property provides greater convergence capability than a purely quadratic (mean square error) behavior since the reward is directly linked to the server displacement cost preventing outliers from having a major impact on learning. The weight vector w, initialized according to Glorot and Bengio (2010), is adjusted through the backpropagation algorithm so the error can be minimized (LeCun, Bengio, & Hinton, 2015). In this way, differentiating the objective function in weights terms we have that its gradient is defined as:

$$\nabla_w L_\delta(\Delta\tilde{Q}) = \begin{cases} (\tilde{Q}^{target} - Q(x_t, a_t; w))\nabla_w \tilde{Q}(x_t, a_t) & |\Delta\tilde{Q}| \leq \delta \\ \dfrac{\tilde{Q}^{target} - Q(x_t, a_t; w)}{|\tilde{Q}^{target} - Q(x_t, a_t; w)|} & \text{otherwise} \end{cases} \tag{5}$$

For a better integration between Q-learning algorithm and the neural network the agent experience $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ is stored in a memory structure $\mathcal{M} = \{e_1, \ldots, e_M\}$ with size of M. During the training phase the weights are updated from samples (or batches) randomly withdraw from memory denoted as e_b, where $b \in \{1, 2, \ldots, B\}$, being B the size of batch. The experience replay (Lin, 1993) prevents certain instances from spending

too much time without being visited. Once the weights are modified the actual target state-action value is affected by the modification and there is no guarantee that $Q(s_t, a_t; w) \approx \tilde{Q}^{target}$, a characteristic that causes the network to diverge. To deal with this non-stationary process the weight w that defines the approximate value $Q(s_{t+1}, a_{t+1}; w)$, denoted here as $\tilde{Q}^{future}$, must be kept constant for a period τ, so that $Q(s_{t+1}, a_{t+1}; w) = Q(s_{t+1}, a_{t+1}; w^-)$, increasing the network convergence potential (Mnih et al., 2015). Although part of the gradient (semi-gradient) is used, this does not interfere with the learning being continuous and online (Sutton & Barto, 2018), an intrinsic characteristic to the k-server problem. Thus, the weights are modified at each sample set instead of being updated at each iteration.

The proposed algorithm belongs to the gradient-based algorithms family adapted to the k-server problem and its pseudocode is shown in Algorithm 1.

Algorithm 1 Q-learning with MLP for the k-server problem.

Require: $G, \alpha, \epsilon, \gamma, \tau, M$ and B
Initialize w
Initialize $w^- = w$
Compute $\mathcal{C}$

repeat
Encode s into o
Perform the pre-processing of o into x
Determine the action (ϵ-greedy)

$$a_{k_l} = \begin{cases} \underset{a_t}{\text{argmin}}\, \tilde{Q}(x, a) & \text{with probability } 1 - \epsilon \\ \text{random (uniform)} & \text{with probability } \epsilon \end{cases}$$

Receive r and observe s'
Store the experience e in $\mathcal{M}$
Remove a set of samples $e_b \sim U(\mathcal{M})$
Update w from samples reducing the Huber error:

$$L_\delta(\Delta\tilde{Q}) = \begin{cases} \frac{1}{2}(\tilde{Q}^{target} - \tilde{Q}(x_b, a_b))^2 & |\Delta\tilde{Q}| \le \delta \\ \delta\left(|\tilde{Q}^{target} - \tilde{Q}(x_b, a_b)| - \frac{1}{2}.\delta\right) & \text{otherwise} \end{cases}$$

where $Q^{target} = c(i, j) + \gamma \min_{a'_b} Q(x'_b, a'_b; w^-) - Q(x_b, a_b; w)$
After each τ step $w^- = w$
until the stopping criterion is reached

4. Case study

The codes were elaborated in Matlab and tested on a computer equipped with 4 gigahertz processor (4 cores), 32 gigabyte of memory and Windows 10 operating system.

4.1. Online mobile emergency problem

To test the algorithm we define the k-server problem in a context of intelligent transport systems providing a practical vision on how the proposed algorithm can be applied. The problem is defined as follows: the city population must be assisted in medical emergency cases characterized by situations that may lead to suffering, sequelae or death. Emergencies are attended by mobile service units (equals) denoted by ambulances distributed through different locations in the city. The goal is to minimize the total time associated with emergency care. To simplify the problem, the patient transportation to a health facility is not considered, only the situation of moving the ambulance to attend an emergency.

Formalizing in terms of the k-server problem we have that G represents a city, each node $n \in G$ a different place (region) of emergency in the city, k the amount of ambulances (servers) and Σ the sequence of emergencies (requests) that can arise in any city location. The attendance of an emergency is characterized when an ambulance $k_{l,i}^{(t)}$ is moved to the place of the emergency $\sigma_j^{(t)}$. Associated with the displacement of each ambulance there is a cost $c_t(i, j)$ defined here as being the time spent (undefined unit) in the attendance of each emergency. The goal is to minimize the total time $\sum_t^T c_t(i, j)$ involved in attending a sequence of emergencies.

Table 1
Hyperparameter table.

n, k	M	B	α	γ	ϵ	τ	θ	w
10, 2	10,000	10	0.01	0.9	0.2	10	0.001	64: 2
15, 2							0.01	128: 2
20, 2		30						256: 2
20, 3	100,000					100	0.05	1024: 3
20, 4								1024: 4
20, 5								1024: 5

4.2. Problem settings

In order to verify the proposed algorithm performance, a city, randomly generated (uniform distribution), was defined with 10 regions to be served and 2 ambulances available to attend the emergencies that randomly (uniform distribution) appear at each region. After this, the city was expanded to 15 and 20 regions and the number of ambulances were gradually increased from $k = 2$ to $k = 5$.

We did not try to define the best hyper-parameters (see Table 1) in such a way that after some initial tests they were set with fixed values. To use a sufficient number of requests to obtain a satisfactory approximate policy, the mean value and standard deviation of the Huber error (by batches) below a limit value denoted by θ were used as the stopping criterion. For the Q-learning algorithm the learning rate was defined as $\alpha = 0.1$ and the same values of ϵ and γ were used, having the same number of requests used in the approximate approach as stop criterion.

4.3. Performance analysis

The proposed solution was analyzed in two stages, during the training phase and after, where its performance was compared with the Q-learning and greedy algorithms. The greedy heuristic does not require training and was used to illustrate an agent behavior that aims a short-term return.

The convergence for Q-learning algorithm optimal solution is ensured if all state-action pairs are visited an "infinite" number of times. Considering the high computational cost that this criterion requires, the analysis of neural network learning was performed for the simplest configuration ($n = 10$ and $k = 2$). The learning curves obtained for predicted and target Q-values of the instance $\{k_{1,3}, k_{2,5}, \sigma_4, a_{k_1}\}$ using Q-learning and the neural network are shown in Figs. 8 and 9, respectively. Whenever server 1 was located at node 3, server 2 at node 5, request at node 4 and server 1 was moved to attend the request, Q-values were recorded until the cost function error reached stop criterion (see Fig. 10). The dotted green line (see Fig. 9) represents the predicted Q-value obtained with Q-learning algorithm. It was used to illustrate the similar learning behavior between the algorithms.

The algorithm performance was tested in six different problem configurations. For each one, 100 experiments (distinct) compounded by a random sequence of emergencies were performed. For each sequence, the total time spent $\sum_{t=0}^T c_t(i, j)$ by each agent in the ambulance displacement was calculated. At the end of the experiments the lowest, highest and average time spent were

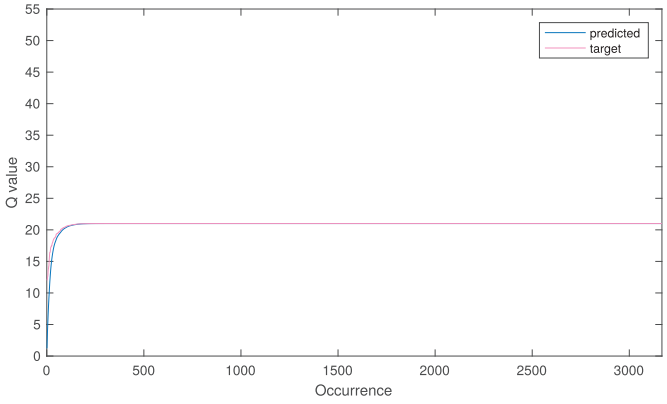


Fig. 8. State-action values with Q-learning.

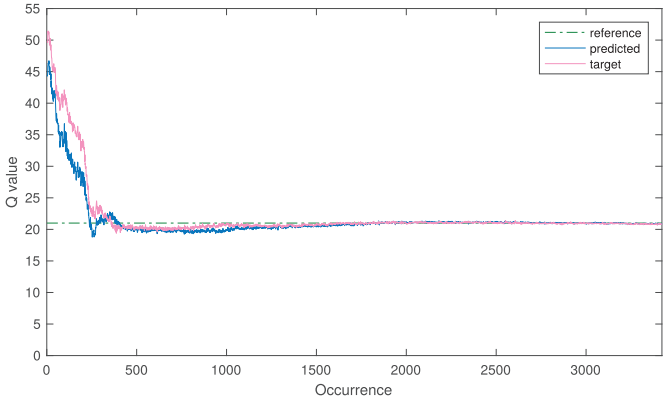


Fig. 9. State-action values with a neural network. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

recorded. Then, a scoring test was carried out among the agents, the ones who shifted the ambulances in less time were considered victorious. In cases they were even, no score was considered. In order to evaluate the performance in short and long term were considered emergency sequences of duration $T = 100$, $T = 1000$ and $T = 10{,}000$.

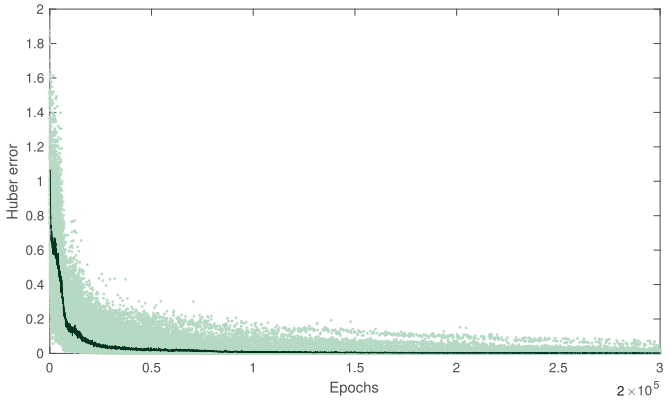


Fig. 10. Loss function.

From the results analysis (see Table 2) in the first problem configuration it was observed that the Q-learning algorithm demonstrated a better displacement policy thus obtaining a greater number of victories. In this case the tabular approach required a storage structure with 900 state-action pairs, while the approximate solution required 770 parameters. As the number of nodes and servers were increased the proposed algorithm obtained a more efficient displacement policy. In the cases with 20 nodes and 4–5 servers, the Q-learning algorithm required a storage structure with 387,600 and 1,550,400 state-action pairs, while the approximate solution required 24,596 and 25,620 parameters, respectively. The storage structure of the neural network was not influenced so much by the exponential growth of the number of instances of the problem as the table used in the traditional approach. The Q table, which has its storage structure directly associated with the increase in the number of state-action pairs, was unable to generate satisfactory displacement policies in a timely manner.

In all cases, it was possible to observe that the agent intelligent behavior in long term was evident with the increase of emergencies attended.

Although the instances discussed here are considered small and medium-sized, larger problems can be addressed through parallelization strategies. The attendance to each region can be considered a localized subproblem, although possible displace-

Table 2
Performance test for different dimensions of the problem.

Algorithms	n, k	$T = 100$				$T = 1000$				$T = 10{,}000$			
		Min	Max	Mean	Wins	Min	Max	Mean	Wins	Min	Max	Mean	Wins
Q-learning	10, 2	291	432	**355.59**	**20**	3273	3745	**3509.69**	**42**	**34,227**	**35,778**	**35,047.11**	**94**
Q-learning with MLP		**284**	432	356.14	12	**3257**	**3738**	3513.74	40	34,258	35,821	35,091.05	6
Greedy		294	**417**	362.52	11	3358	3777	3568.77	10	35,039	36,360	35,704.00	0
Q-learning	15,2	334	510	411.30	27	3836	4324	4058.46	13	40,008	41,319	40,681.76	0
Q-learning with MLP		**319**	**503**	**408.64**	**46**	**3761**	**4282**	**4019.94**	**86**	**39,716**	**40,913**	**40,303.76**	**100**
Greedy		324	522	418.96	15	3913	4356	4144.74	1	40,892	42,020	41,485.92	0
Q-learning	20,2	348	497	419.29	20	3973	4452	4230.58	3	41,731	42,952	42,343.22	0
Q-learning with MLP		**343**	501	**411.78**	**67**	**3887**	**4384**	**4136.27**	**97**	**40,649**	**42,126**	**41,368.35**	**100**
Greedy		373	**496**	428.10	8	4099	4575	4318.00	0	42,510	43,993	43,210.14	0
Q-learning	20,3	**277**	**393**	344.75	33	3263	3761	3509.33	19	34,417	35,767	35,081.79	9
Q-learning with MLP		285	415	**344.31**	**46**	**3248**	**3708**	**3469.00**	**78**	**34,156**	**35,663**	**34,790.00**	**91**
Greedy		301	433	353.51	17	3373	3773	3597.96	0	35,325	36,646	36,003.36	0
Q-learning	20,4	258	384	317.74	13	2963	3386	3206.60	0	31,272	32,507	31,943.31	0
Q-learning with MLP		**226**	**381**	**294.94**	**50**	2814	**3143**	**2980.62**	**89**	**29,250**	**30,566**	**29,819.78**	**100**
Greedy		250	369	302.90	34	**2797**	3214	3055.83	11	30,114	31,131	30,568.75	0
Q-learning	20,5	253	381	314.51	0	2961	3436	3205.79	0	31,500	32,843	32,058.52	0
Q-learning with MLP		199	**320**	**252.25**	**58**	**2351**	**2724**	**2526.08**	**93**	**24,682**	**26,041**	**25,318.13**	**100**
Greedy		199	321	258.26	42	2444	2738	2600.74	7	25,267	26,652	26,072.42	0

Keywords:
Min - Minimum time spent
Max - Maximum time spent
Max - Maximum time spent
Wins - Victories

ments of the servers between the regions are allowed, thus dealing with larger dimensions of the problem as carried out by recent approaches (Bertsimas et al., 2019; Costa et al., 2016; Rudec et al., 2013).

5. Conclusion

In this work, an algorithm based on the deep reinforcement learning paradigm was presented as an alternative solution to the *k*-server problem. To verify the performance of the proposed solution the algorithm was applied to a mobile emergency service simulation. Through the results obtained, the new algorithm demonstrated satisfactorily to be able to perform the displacement of health units in different problem configurations. The scalability of the proposed algorithm was evident in the larger cases of the problem, where emphasis was given to the increase of server quantity. Although an initial state is required to initialize the problem, the solution is not restricted to this condition. Its performance was evaluated considering different possible initial states, a condition that increases problem degree of uncertainty and makes the solution even more robust. In general, the proposed algorithm presented a better performance in the tests than the other algorithms, reducing satisfactorily emergency time response. The results obtained here indicate that the new algorithm can be an effective approach on the solution of the *k*-server problem.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Credit authorship contribution statement

Ramon Augusto Sousa Lins: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Visualization, Writing - review & editing. **Adrião Duarte Neto Dória:** Conceptualization, Methodology, Resources, Writing - review & editing, Supervision, Project administration, Funding acquisition. **Jorge Dantas de Melo:** Conceptualization, Methodology, Visualization, Resources, Writing - review & editing, Supervision.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de NíAvel Superior - Brasil (CAPES) - Finance Code 001.

References

Afify, B., Ray, S., Soeanu, A., Awasthi, A., Debbabi, M., & Allouche, M. (2019). Evolutionary learning algorithm for reliable facility location under disruption. *Expert Systems with Applications, 115*, 223–244.
Allan, B., & El-Yaniv, R. (1998). *Online computation and competitive analysis*.
Araque, O., Corcuera-Platas, I., Sanchez-Rada, J. F., & Iglesias, C. A. (2017). Enhancing deep learning sentiment analysis with ensemble techniques in social applications. *Expert Systems with Applications, 77*, 236–246.
Bansal, N., Buchbinder, N., Madry, A., & Naor, J. S. (2015). A polylogarithmic-competitive algorithm for the k-server problem. *Journal of the ACM, 62*(5), 40.
Bello, I., Pham, H., Le, Q. V., Norouzi, M., & Bengio, S. (2016). *Neural combinatorial optimization with reinforcement learning* arXiv:1611.09940.
Bertsimas, D., Jaillet, P., & Korolko, N. (2019). The k-server problem via a modern optimization lens. *European Journal of Operational Research, 276*(1), 65–78.
Costa, M. L., Padilha, C. A. A., Melo, J. D., & Neto, A. D. D. (2016). Hierarchical reinforcement learning and parallel computing applied to the k-server problem. *IEEE Latin America Transactions, 14*(10), 4351–4357.
Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256).
Gupta, S., Kamali, S., & López-Ortiz, A. (2016). On the advice complexity of the k-server problem under sparse metrics. *Theory of Computing Systems, 59*(3), 476–499.
Huber, P. J. (1964). Robust estimation of a location parameter. *The Annals of Mathematical Statistics, 35*(1), 73–101.
Junior, M. L. L., Neto, A. D., & Melo, J. D. (2005). The k-server problem: a reinforcement learning approach. In *Neural networks, 2005. IJCNN'05. proceedings. 2005 IEEE international joint conference on: 2* (pp. 798–802). IEEE.
Koutsoupias, E. (2009). The k-server problem. *Computer Science Review, 3*(2), 105–118.
LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature, 521*(7553), 436.
Lin, L.-J. (1993). Reinforcement learning for robots using neural networks. *Technical Report*. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.
Manasse, M., McGeoch, L., & Sleator, D. (1988). Competitive algorithms for on-line problems. In *Proceedings of the twentieth annual ACM symposium on theory of computing* (pp. 322–333). ACM.
Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016). Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks* (pp. 50–56). ACM.
Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., et al. (2015). Human-level control through deep reinforcement learning. *Nature, 518*(7540), 529.
Riedmiller, M. (2005). Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European conference on machine learning* (pp. 317–328). Springer.
Rudec, T., Baumgartner, A., & Manger, R. (2013). A fast work function algorithm for solving the k-server problem. *Central European Journal of Operations Research, 21*(1), 187–205.
Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., et al. (2017). Mastering the game of go without human knowledge. *Nature, 550*(7676), 354.
Sleator, D. D., & Tarjan, R. E. (1985). Amortized efficiency of list update and paging rules. *Communications of the ACM, 28*(2), 202–208.
Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.
Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM, 38*(3), 58–68.
Watkins, C. J. C. H. (1989). *Learning from delayed rewards*.